

# CHAPTER4

# TENSORRT模型部署优化

主 讲：韩君  
单 位：早稻田大学  
公众号：自动驾驶之心

# 主要内容

1 模型部署基础知识

2 模型部署的几大误区

3 模型量化

4 模型剪枝

5 层融合



01

## 模型部署的 基础知识

Goal: 理解FLOPS和TOPS是什么，CPU/GPU中的计算FLOPS/TOPS的公式，  
以及CUDA Core和Tensor Core的区别

# FLOPS

FLOPS:

(Floating point number operations per second)<sup>(\*)</sup>

- 指的是一秒钟可以处理的浮点小数点运算的次数
- 衡量计算机硬件性能、计算能力的一个单位

各种FLOPS	全称	量级
MFLOPS	Mega FLOPS	$10^6$ FLOPS
GFLOPS	Giga FLOPS	$10^9$ FLOPS
TFLOPS	Tera FLOPS	$10^{12}$ FLOPS
PFLOPS	Peta FLOPS	$10^{15}$ FLOPS

常见的FLOPS

TOPS:

(Tera operations per second)

- 指的是一秒钟可以处理的整型运算的次数
- 衡量计算机硬件性能、计算能力的一个单位

Peak FP64 <sup>1</sup>	9.7 TFLOPS
Peak FP64 Tensor Core <sup>1</sup>	19.5 TFLOPS
Peak FP32 <sup>1</sup>	19.5 TFLOPS
Peak FP16 <sup>1</sup>	78 TFLOPS
Peak BF16 <sup>1</sup>	39 TFLOPS
Peak TF32 Tensor Core <sup>1</sup>	156 TFLOPS   312 TFLOPS <sup>2</sup>
Peak FP16 Tensor Core <sup>1</sup>	312 TFLOPS   624 TFLOPS <sup>2</sup>
Peak BF16 Tensor Core <sup>1</sup>	312 TFLOPS   624 TFLOPS <sup>2</sup>
Peak INT8 Tensor Core <sup>1</sup>	624 TOPS   1,248 TOPS <sup>2</sup>
Peak INT4 Tensor Core <sup>1</sup>	1,248 TOPS   2,496 TOPS <sup>2</sup>

NVIDIA Ampere架构的TESLA A100的性能<sup>[1]</sup>

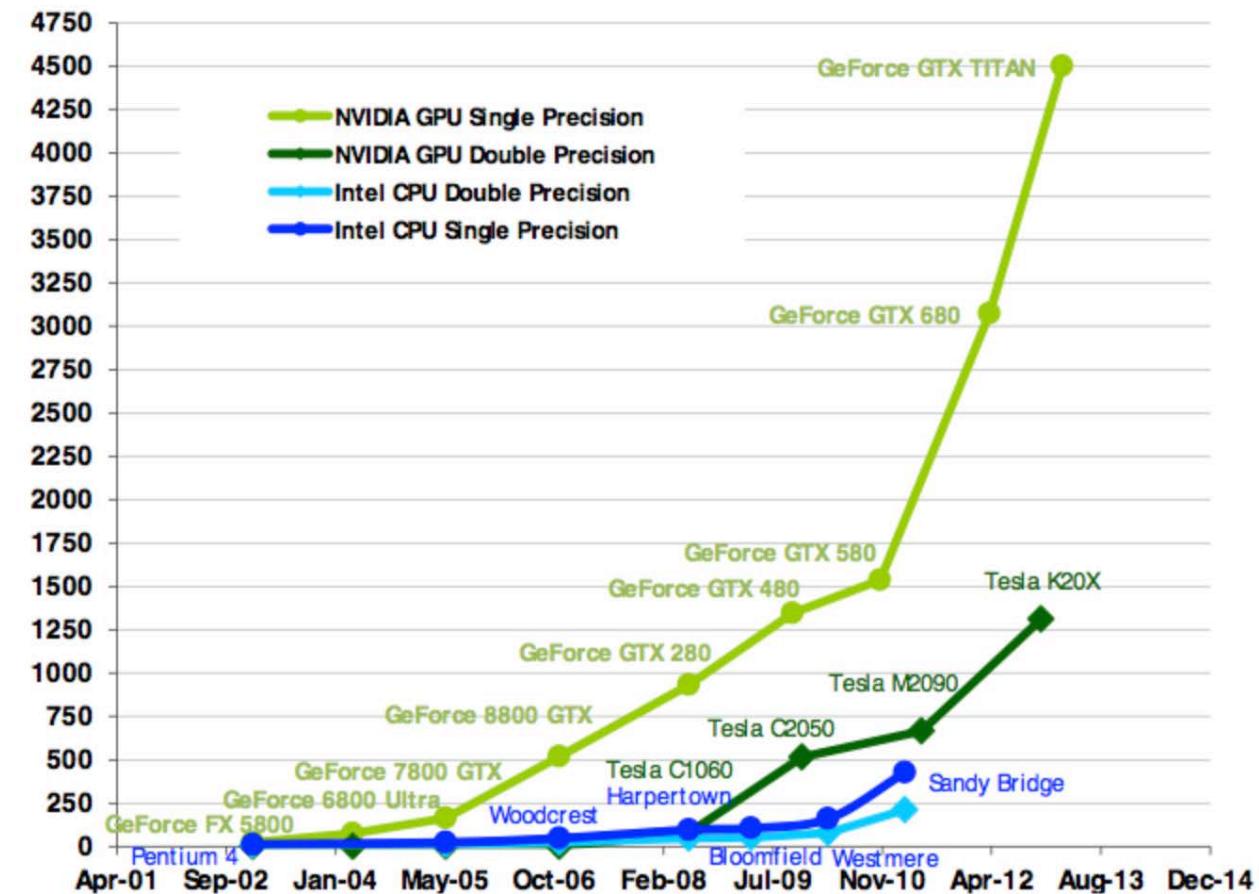
<sup>[1]</sup>AI Chips: A100 GPU with Nvidia Ampere architecture

# HPC的排行，CPU/GPU比较

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power [kW]
1	Frontier - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,730,112	1,102.00	1,685.65	21,100
2	Supercomputer Fugaku - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
3	LUMI - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	1,110,144	151.90	214.35	2,942
4	Summit - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148.60	200.79	10,096
5	Sierra - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LNLL United States	1,572,480	94.64	125.71	7,438
6	Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC National Supercomputing Center in Wuxi China	10,649,600	93.01	125.44	15,371
7	Perlmutter - HPE Cray EX235n, AMD EPYC 7763 64C 2.45GHz, NVIDIA A100 SXM4 40 GB, Slingshot-10, HPE DOE/SC/LBNL/NERSC United States	761,856	70.87	93.75	2,589
8	Selene - NVIDIA DGX A100, AMD EPYC 7742 64C 2.25GHz, NVIDIA A100, Mellanox HDR Infiniband, Nvidia NVIDIA Corporation United States	555,520	63.46	79.22	2,646
9	Tianhe-2A - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000, NUDT National Super Computer Center in Guangzhou China	4,981,760	61.44	100.68	18,482
10	Adastra - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE Grand Equipment National de Calcul Intensif - Centre Informatique National de l'Enseignement Supérieur	319,072	46.10	61.61	921

HPC历年的排名<sup>[1]</sup>

Theoretical  
GFLOP/s



CPU与GPU在FLOPS上的性能比<sup>[2]</sup>

# FLOPs

FLOPs:  
(Floating point number operations)

是衡量模型大小的一个指标，大家在CVPR的paper或者Github里经常能够看到的就是这个信息

name	pretrain	resolution	acc@1	acc@5	#params	FLOPs	FPS	22K model
Swin-T	ImageNet-1K	224x224	81.2	95.5	28M	4.5G	755	-
Swin-S	ImageNet-1K	224x224	83.2	96.2	50M	8.7G	437	-
Swin-B	ImageNet-1K	224x224	83.5	96.5	88M	15.4G	278	-
Swin-B	ImageNet-1K	384x384	84.5	97.0	88M	47.1G	85	-
Swin-T	ImageNet-22K	224x224	80.9	96.0	28M	4.5G	755	<a href="#">github/baidu/config</a>
Swin-S	ImageNet-22K	224x224	83.2	97.0	50M	8.7G	437	<a href="#">github/baidu/config</a>
Swin-B	ImageNet-22K	224x224	85.2	97.5	88M	15.4G	278	<a href="#">github/baidu/config</a>
Swin-B	ImageNet-22K	384x384	86.4	98.0	88M	47.1G	85	<a href="#">github/baidu</a>
Swin-L	ImageNet-22K	224x224	86.3	97.9	197M	34.5G	141	<a href="#">github/baidu/config</a>
Swin-L	ImageNet-22K	384x384	87.3	98.2	197M	103.9G	42	<a href="#">github/baidu</a>

Swin Transformer中的FLOPs<sup>[1]</sup>

Table 7. EfficientNetV2 Performance Results on ImageNet (Russakovsky et al., 2015) – Infer-time is measured on V100 GPU FP16 with batch size 16 using the same codebase (Wightman, 2021); Train-time is the total training time normalized for 32 TPU cores. Models marked with 21k are pretrained on ImageNet21k with 13M images, and others are directly trained on ImageNet ILSVRC2012 with 1.28M images from scratch. All EfficientNetV2 models are trained with our improved method of progressive learning.

Model	Top-1 Acc.	Params	FLOPs	Infer-time(ms)	Train-time (hours)
ConvNets & Hybrid	EfficientNet-B3 (Tan & Le, 2019a)	81.5%	12M	1.9B	19
	EfficientNet-B4 (Tan & Le, 2019a)	82.9%	19M	4.2B	30
	EfficientNet-B5 (Tan & Le, 2019a)	83.7%	30M	10B	60
	EfficientNet-B6 (Tan & Le, 2019a)	84.3%	43M	19B	97
	EfficientNet-B7 (Tan & Le, 2019a)	84.7%	66M	38B	170
	RegNetY-8GF (Radosavovic et al., 2020)	81.7%	39M	8B	21
	RegNetY-16GF (Radosavovic et al., 2020)	82.9%	84M	16B	32
	ResNeSt-101 (Zhang et al., 2020)	83.0%	48M	13B	31
	ResNeSt-200 (Zhang et al., 2020)	83.9%	70M	36B	76
	ResNeSt-269 (Zhang et al., 2020)	84.5%	111M	78B	160
	TResNet-L (Ridnik et al., 2020)	83.8%	56M	-	45
	TResNet-XL (Ridnik et al., 2020)	84.3%	78M	-	66
	EfficientNet-X (Li et al., 2021)	84.7%	73M	91B	-
	NFNet-F0 (Brock et al., 2021)	83.6%	72M	12B	30
	NFNet-F1 (Brock et al., 2021)	84.7%	133M	36B	70
	NFNet-F2 (Brock et al., 2021)	85.1%	194M	63B	124
	NFNet-F3 (Brock et al., 2021)	85.7%	255M	115B	203
	NFNet-F4 (Brock et al., 2021)	85.9%	316M	215B	309
Vision Transformers	LambdaResNet-420-hybrid (Bello, 2021)	84.9%	125M	-	67
	BotNet-T7-hybrid (Srinivas et al., 2021)	84.7%	75M	46B	-
	BiT-M-R152x2 (21k) (Kolesnikov et al., 2020)	85.2%	236M	135B	500
	ViT-B/32 (Dosovitskiy et al., 2021)	73.4%	88M	13B	13
	ViT-B/16 (Dosovitskiy et al., 2021)	74.9%	87M	56B	68
	DeiT-B (ViT+reg) (Touvron et al., 2021)	81.8%	86M	18B	19
	DeiT-B-384 (ViT+reg) (Touvron et al., 2021)	83.1%	86M	56B	68
ConvNets (ours)	T2T-ViT-19 (Yuan et al., 2021)	81.4%	39M	8.4B	-
	T2T-ViT-24 (Yuan et al., 2021)	82.2%	64M	13B	-
	ViT-B/16 (21k) (Dosovitskiy et al., 2021)	84.6%	87M	56B	68
	ViT-L/16 (21k) (Dosovitskiy et al., 2021)	85.3%	304M	192B	195
	EfficientNetV2-S	83.9%	22M	8.8B	24
	EfficientNetV2-M	85.1%	54M	24B	57
	EfficientNetV2-L	85.7%	120M	53B	98

We do not include models pretrained on non-public Instagram/JFT images, or models with extra distillation or ensemble.

EfficientNetV2中的FLOPs<sup>[2]</sup>

# FLOPS是如何计算的

跟FLOPS有关的概念

- Core数量
- 时钟频率
- 每个时钟周期可以处理的FLOPS

$$\text{FLOPS} = \text{频率} * \text{core数量} * \text{每个时钟周期可以处理的FLOPS}$$

比如说：

Intel i7 Haswell架构 (8核， 频率3.0GHz)。

思考一下：这个16 FLOPS/clk是怎么算的？

那么它的FLOPS在**双精度**的时候就是：

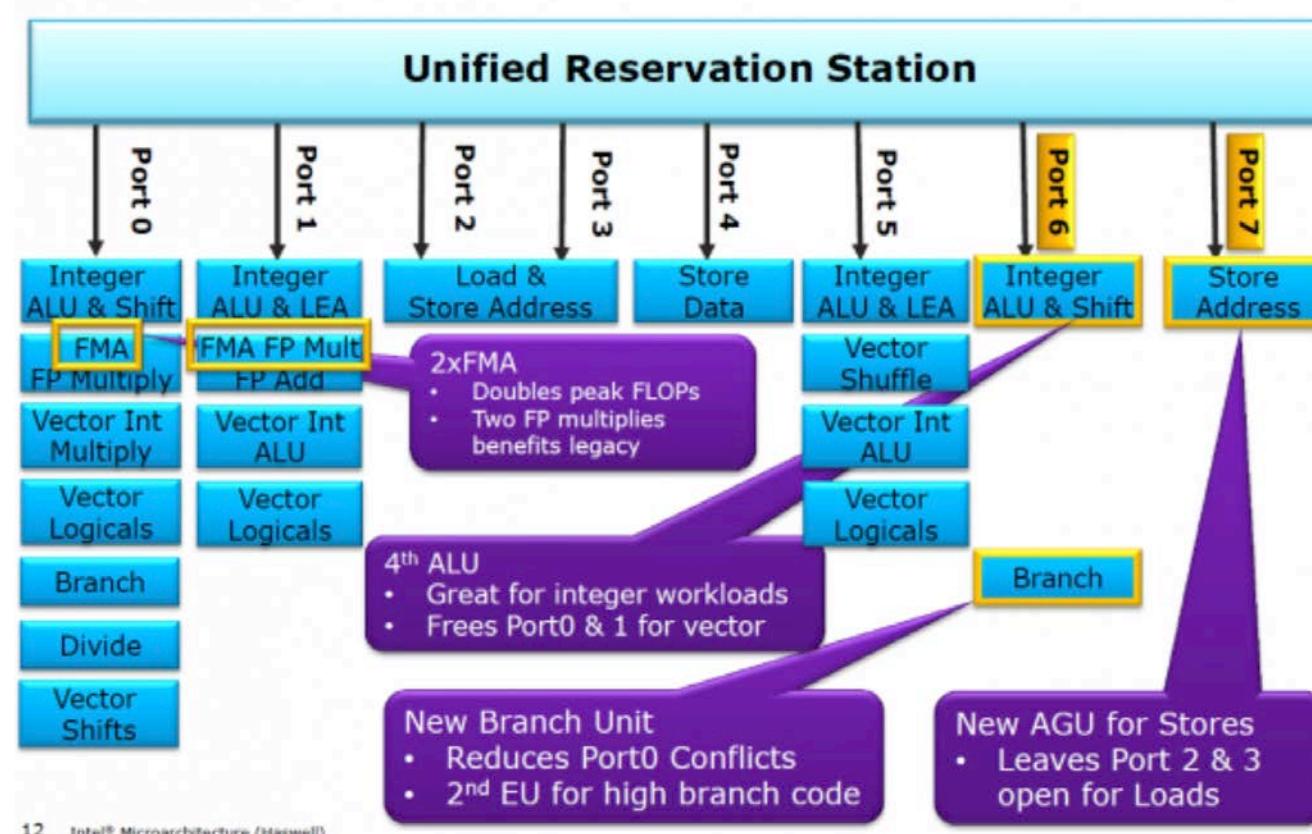
$$3.0 * 10^9 \text{ Hz} * 8 \text{ core} * 16 \text{ FLOPS/clk} = 0.38 \text{ TFLOPS}$$

那么它的FLOPS在**单精度**的时候就是：

$$3.0 * 10^9 \text{ Hz} * 8 \text{ core} * 32 \text{ FLOPS/clk} = 0.76 \text{ TFLOPS}$$

有两个FMA，以及支持AVX-256指令集

## Haswell Execution Units



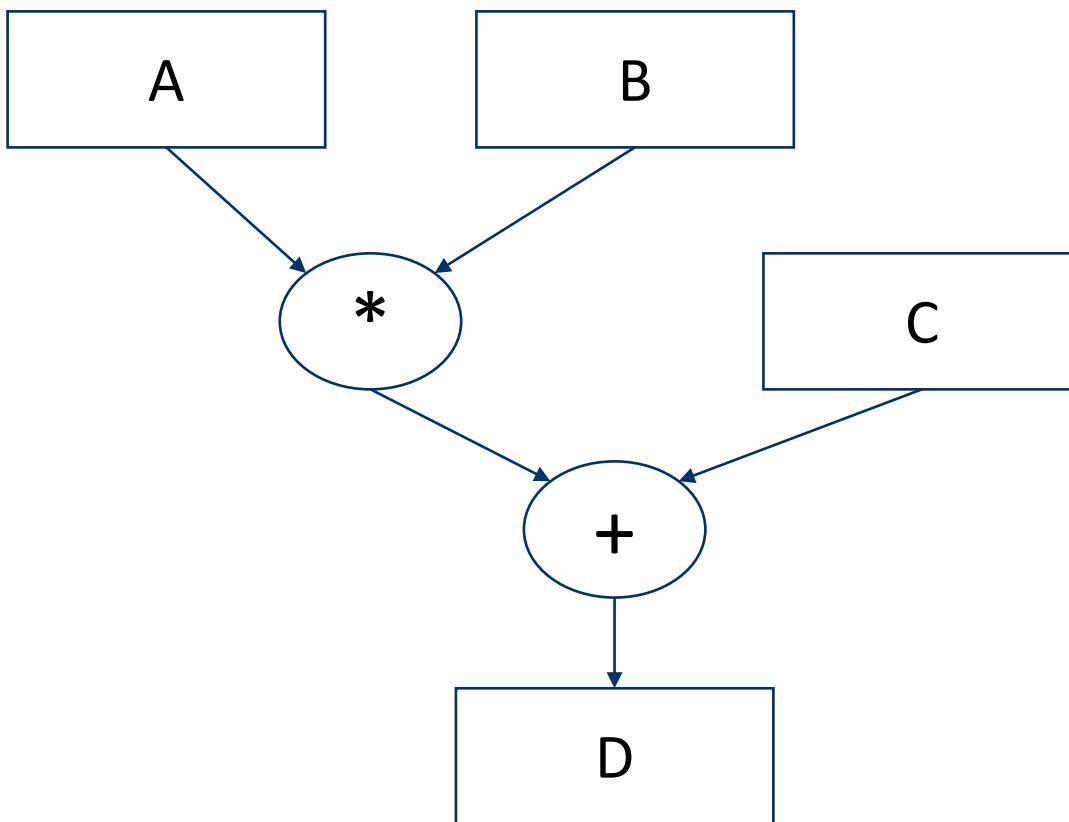
Intel Haswell计算单元架构图[1]

[1]Intel Processor Architecture

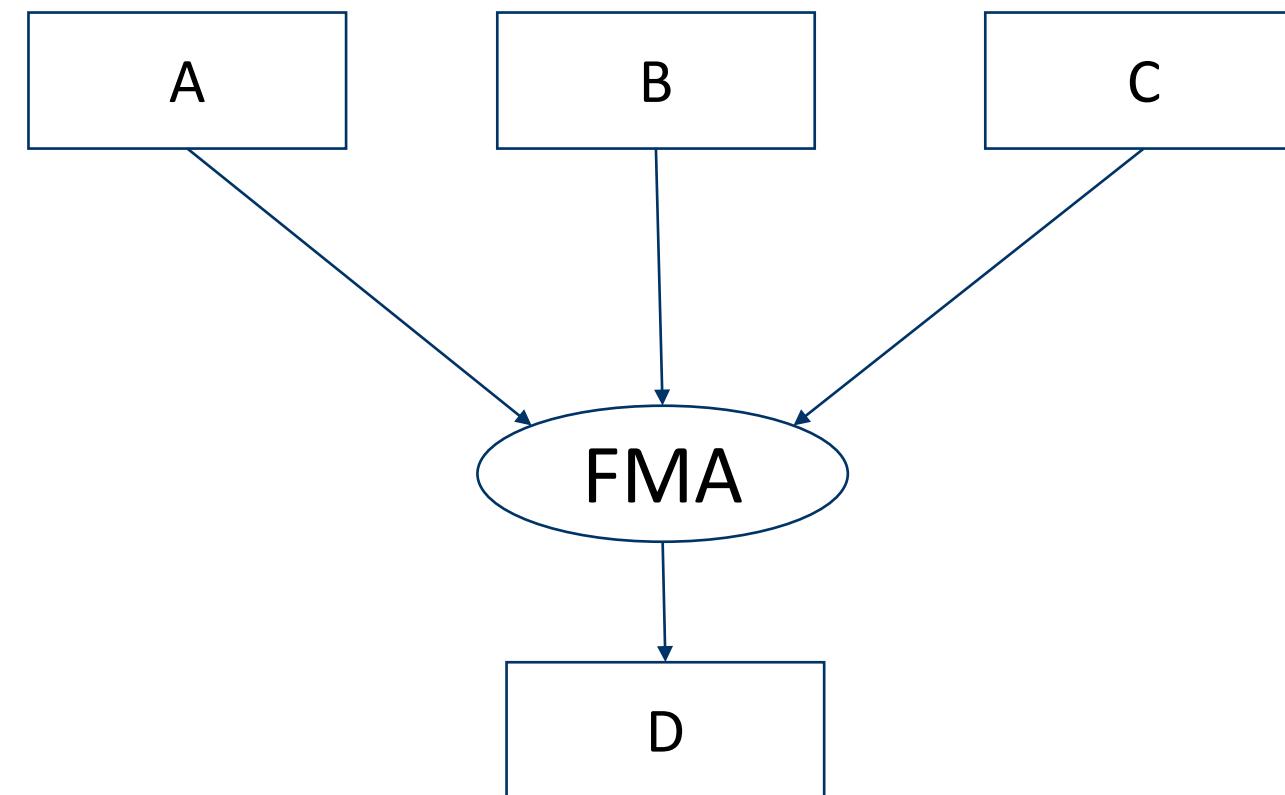
# FMA计算

## Fused Multiply Accumulation

$$D = A * B + C$$



没有FMA，乘法加法**分开算**  
计算 $D = A * B + C$ 需要**两个**时钟周期



有FMA，乘法加法**一起算**  
计算 $D = A * B + C$ 需要**一个**时钟周期

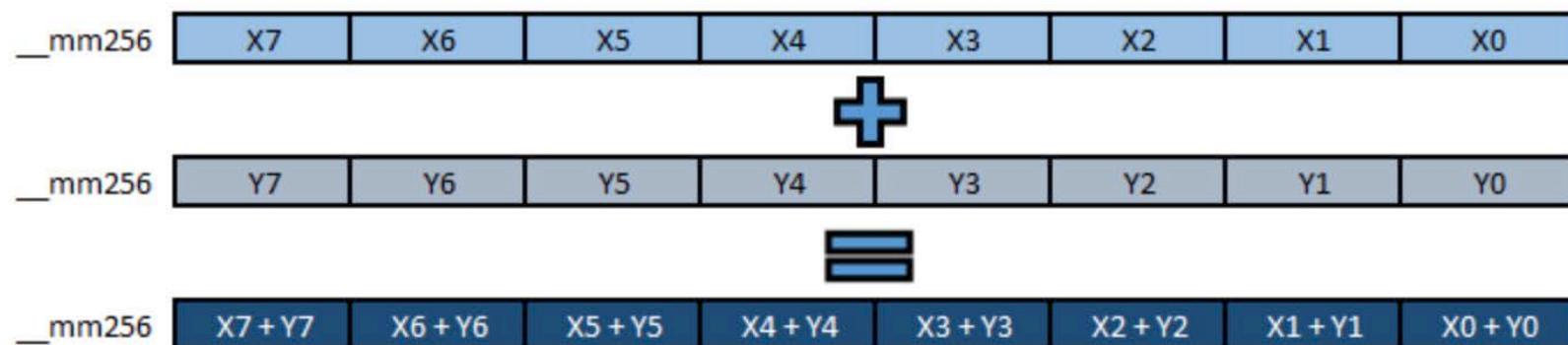
# AVX-256指令集

## AVX Data Types (16 YMM Registers)

<code>_mm256</code>	Float	8x 32-bit float							
<code>_mm256d</code>	Double	4x 64-bit double							

AVX-256的数据类型<sup>[2]</sup>

## AVX Operation



AVX-256的SIMD操作<sup>[2]</sup>

[2]What is SSE and AVX?

## FLOPS是如何计算的

Intel i7 Haswell架构 (8核, 频率3.0GHz)。

那么它的FLOPS在**双精度**的时候就是:

$$3.0 * 10^9 \text{ Hz} * 8 \text{ core} * 16 \text{ FLOPS/clk} = 0.38 \text{ TFLOPS}$$

16 FLOPS/clk

2 FMA \* 4个FP64的SIMD运算 \* 2 乘加融合= 16 FLOPS/clk

Intel i7 Haswell架构 (8核, 频率3.0GHz)。

那么它的FLOPS在**单精度**的时候就是:

$$3.0 * 10^9 \text{ Hz} * 8 \text{ core} * 32 \text{ FLOPS/clk} = 0.76 \text{ TFLOPS}$$

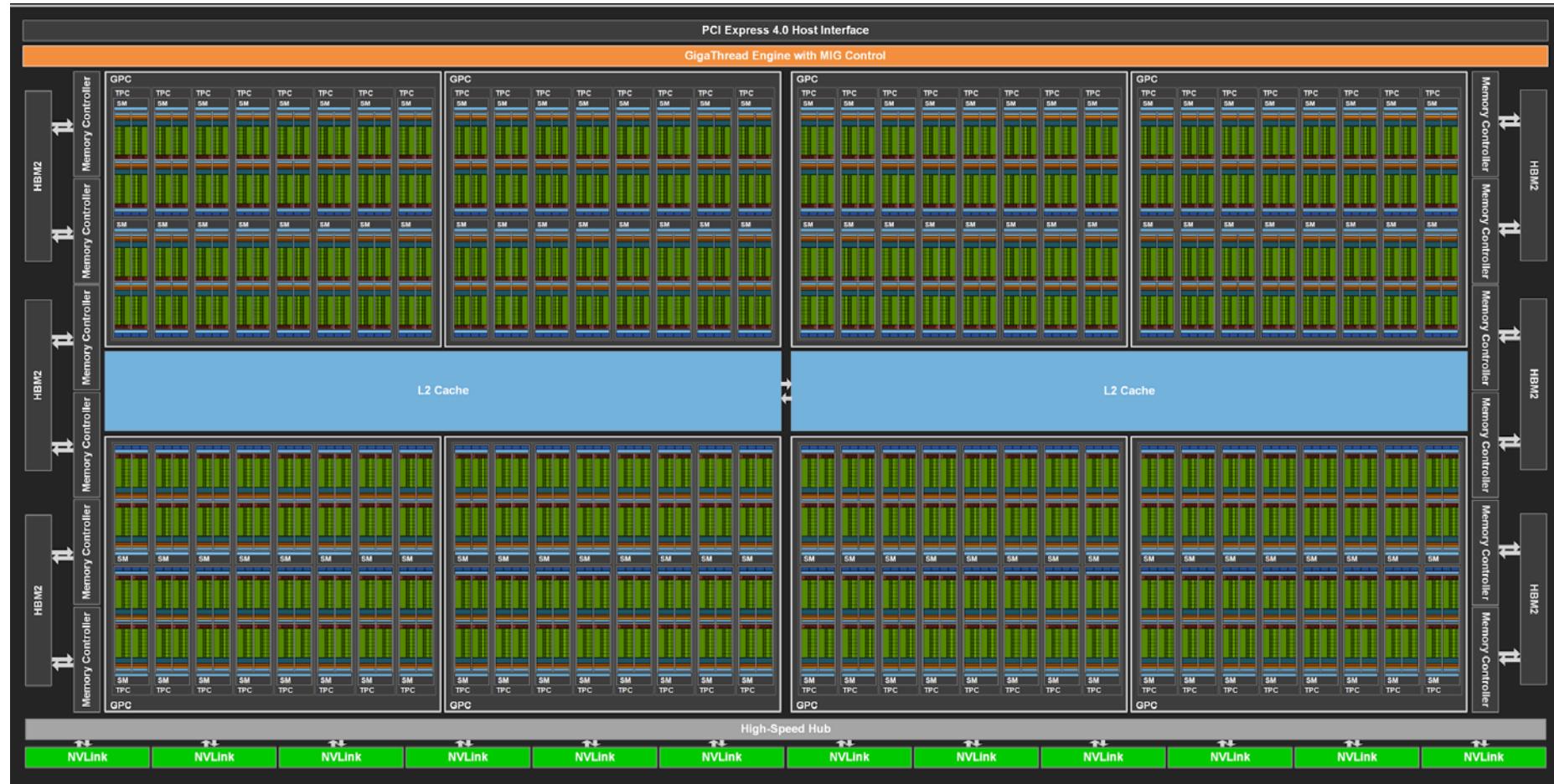
32 FLOPS/clk

2 FMA \* 8个FP32的SIMD运算 \* 2 乘加融合= 32 FLOPS/clk

# FLOPS在GPU中是如何计算的?

跟CPU不同的地方

- 没有AVX这种东西
- 但是有大量的core来提高吞吐量
- 有Tensor Core来优化矩阵运算



NVIDIA TESLA A100, Ampere Architecture, 一共108个SM<sup>[1]</sup>

# Ampere SM Architecture



一个SM里面有

- 64个处理INT32的CUDA Core
- 64个处理FP32的CUDA Core
- 32个处理FP64的CUDA Core
- 4个处理矩阵计算的Tensor Core

思考一下: 这些值是怎么算出来的?

	FP64	FP32	TF32	FP16	INT8	INT4	INT1
CUDA Core	32	64	-	256	256	-	-
Tensor Core	64		512	1024	2048	4096	16484

每一种精度在一个SM中，一个clk可以完成的计算数量

<sup>[1]</sup>AI Chips: A100 GPU with Nvidia Ampere architecture

# Ampere SM Architecture

思考一下: 这些值是怎么算出来的?

Peak FP64 <sup>1</sup>	9.7 TFLOPS
Peak FP64 Tensor Core <sup>1</sup>	19.5 TFLOPS
Peak FP32 <sup>1</sup>	19.5 TFLOPS
Peak FP16 <sup>1</sup>	78 TFLOPS
Peak BF16 <sup>1</sup>	39 TFLOPS
Peak TF32 Tensor Core <sup>1</sup>	156 TFLOPS   312 TFLOPS <sup>2</sup>
Peak FP16 Tensor Core <sup>1</sup>	312 TFLOPS   624 TFLOPS <sup>2</sup>
Peak BF16 Tensor Core <sup>1</sup>	312 TFLOPS   624 TFLOPS <sup>2</sup>
Peak INT8 Tensor Core <sup>1</sup>	624 TOPS   1,248 TOPS <sup>2</sup>
Peak INT4 Tensor Core <sup>1</sup>	1,248 TOPS   2,496 TOPS <sup>2</sup>

Ampere架构的各个精度的Throughput<sup>[1]</sup>

[1]深度了解 NVIDIA Ampere 架构

# Ampere SM Architecture

Peak FP64<sup>1</sup>

9.7 TFLOPS



	FP64	FP32	TF32	FP16	INT8	INT4	INT1
CUDA Core	32	64	-	256	256	-	-
Tensor Core	64		512	1024	2048	4096	16484

使用CUDA Core的**FP64**最好理解  
一共32个计算FP64的CUDA Core, 每一个clk计算一个FP64

- 频率: 1.41 GHz
- SM数量: 108
- 一个SM中计算FP32的CUDA core的数量: 32
- 一个CUDA core一个时钟周期可以处理的FP64: 1
- 乘加 : 2

$$\text{Throughput} = 1.41 \text{ GHz} * 108 * 32 * 1 * 2 = 9.7 \text{ TFLOPS}$$

<sup>[1]</sup>AI Chips: A100 GPU with Nvidia Ampere architecture

# Ampere SM Architecture

Peak FP32<sup>1</sup>

19.5 TFLOPS



	FP64	FP32	TF32	FP16	INT8	INT4	INT1
CUDA Core	32	64	-	256	256	-	-
Tensor Core	64		512	1024	2048	4096	16484

使用CUDA Core的FP32跟FP64差不多  
一共64个计算FP32的CUDA Core, 每一个clk计算一个FP32

- 频率: 1.41 GHz
- SM数量: 108
- 一个SM中计算FP32的CUDA core的数量: 64
- 一个CUDA core一个时钟周期可以处理的FP32: 1
- 乘加: 2

$$\text{Throughput} = 1.41 \text{ GHz} * 108 * 64 * 1 * 2 = 19.4 \text{ TFLOPS}$$

<sup>[1]</sup>AI Chips: A100 GPU with Nvidia Ampere architecture

# Ampere SM Architecture

Peak FP16<sup>1</sup>

78 TFLOPS



	FP64	FP32	TF32	FP16	INT8	INT4	INT1
CUDA Core	32	64	-	256	256	-	-
Tensor Core	64		512	1024	2048	4096	16484

Ampere中没有专门针对FP16的CUDA core，而是将FP32的CUDA Core和FP64的CUDA Core一起使用来计算FP16。我们苟且说一个SM中计算FP16的CUDA core的数量是: 256 ( = 32 \* 2 + 16 \* 4 )

- 频率: 1.41 GHz
- SM数量: 108
- 一个SM中计算FP16的CUDA core的数量: 256
- 一个CUDA core一个时钟周期可以处理的FP16: 1
- 乘加: 2

$$\text{Throughput} = 1.41 \text{ GHz} * 108 * 256 * 1 * 2 = 78 \text{ TFLOPS}$$

<sup>[1]</sup>AI Chips: A100 GPU with Nvidia Ampere architecture

# Ampere SM Architecture



	FP64	FP32	TF32	FP16	INT8	INT4	INT1
CUDA Core	32	64	-	256	256	-	-
Tensor Core	64		512	1024	2048	4096	16484

Ampere中没有专门针对INT8的CUDA core，而是用INT32的CUDA Core计算INT8。我们苟且说一个SM中计算INT8的CUDA core的数量是: 256 ( = 64 \* 4 )

- 频率: 1.41 GHz
- SM数量: 108
- 一个SM中计算INT8的CUDA core的数量: 256
- 一个CUDA core一个时钟周期可以处理的INT8: 1
- 乘加 : 2

$$\text{Throughput} = 1.41 \text{ GHz} * 108 * 256 * 1 * 2 = 78 \text{ TOPS}$$

<sup>[1]</sup>AI Chips: A100 GPU with Nvidia Ampere architecture

# Ampere SM Architecture

Peak FP16 Tensor Core<sup>1</sup>

312 TFLOPS | 624 TFLOPS<sup>2</sup>



	FP64	FP32	TF32	FP16	INT8	INT4	INT1
CUDA Core	32	64	-	256	256	-	-
Tensor Core	64		512	1024	2048	4096	16484

我们来分析Tensor Core

Ampere架构使用的是第三代Tensor Core，可以一个clk完成一个 $1024 (= 256 * 4)$ 个FP16运算。准确来说是4x8的矩阵与8x8的矩阵的FMA

- 频率: 1.41 GHz
- SM数量: 108
- 一个SM中计算FP16的Tensor core的数量: 4
- 一个Tensor core一个时钟周期可以处理的FP16: 256
- 乘加: 2

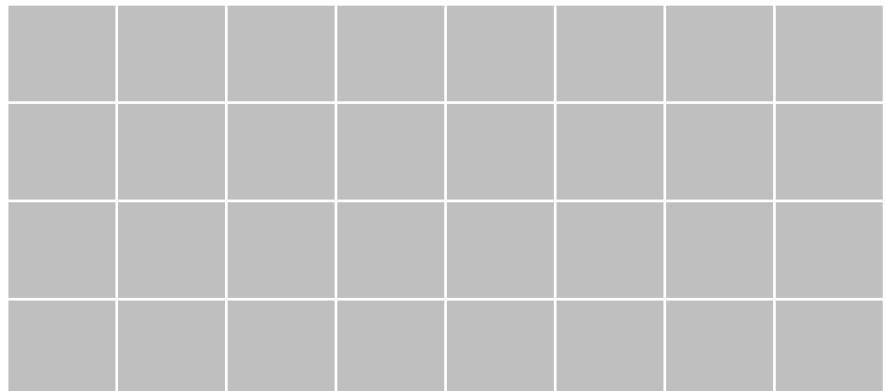
$$\text{Throughput} = 1.41 \text{ GHz} * 108 * 4 * 256 * 2 = 312 \text{ TFLOPS}$$

<sup>[1]</sup>AI Chips: A100 GPU with Nvidia Ampere architecture

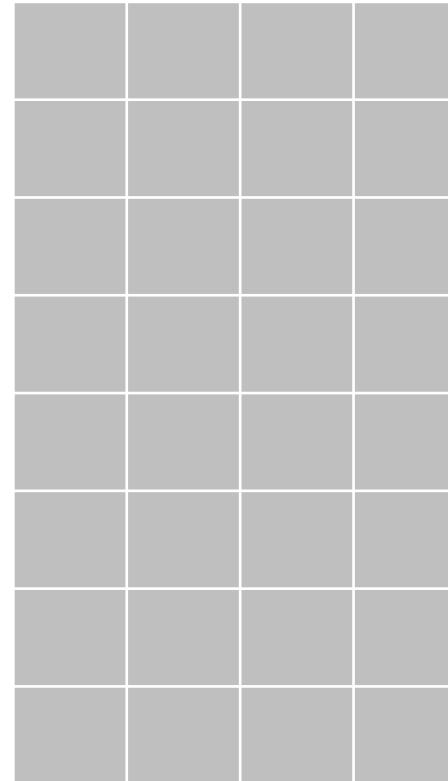
# CUDA Core vs Tensor Core

使用一个CUDA Core  
计算  $C = A * B$

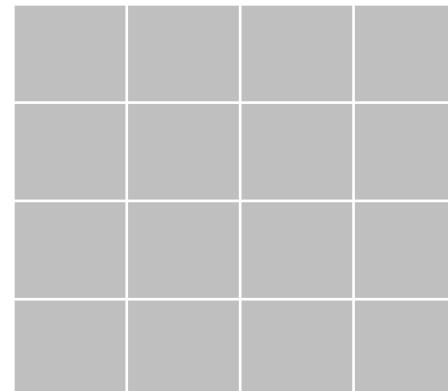
A(4 x 8)



B(8 x 4)



C(4 x 4)



# CUDA Core vs Tensor Core

使用一个CUDA Core  
计算  $C = A * B$

A(4 x 8)


B(8 x 4)


C(4 x 4)


# CUDA Core vs Tensor Core

使用一个CUDA Core  
计算  $C = A * B$

A(4 x 8)

a(0,0)	a(0,1)	a(0,2)	a(0,3)	a(0,4)	a(0,5)	a(0,6)	a(0,7)

B(8 x 4)

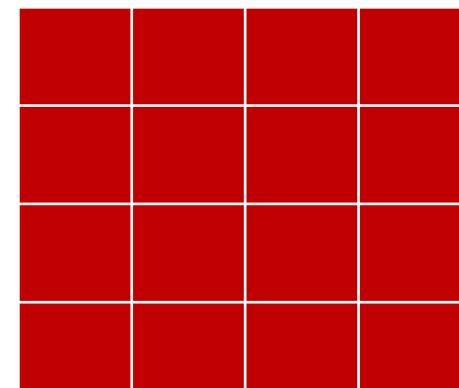
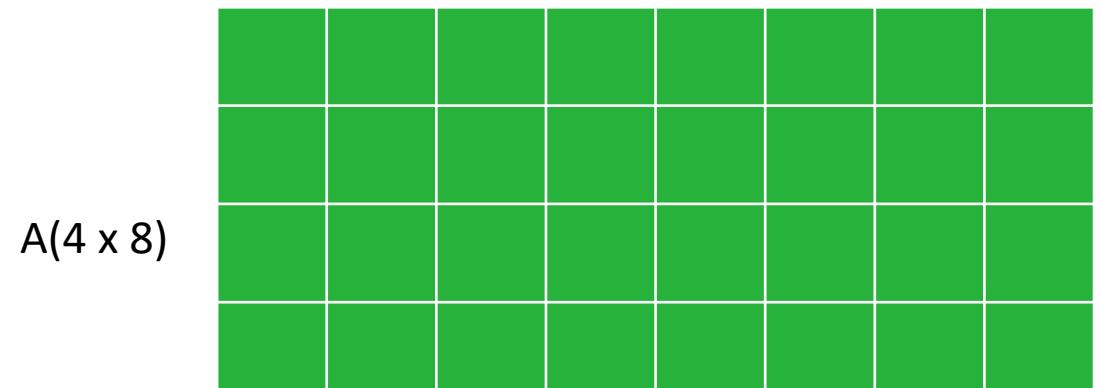
b(0,0)			
b(1,0)			
b(2,0)			
b(3,0)			
b(4,0)			
b(5,0)			
b(6,0)			
b(7,0)			

C(4 x 4)

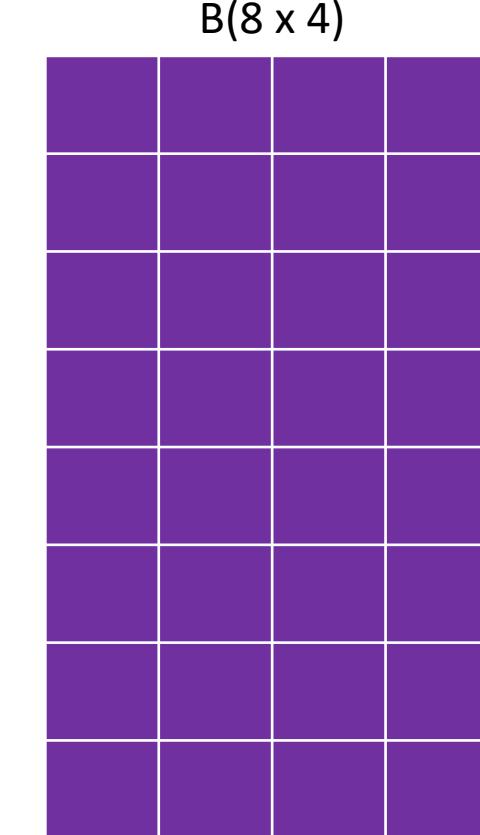
$$\begin{aligned} c(0, 0) \\ &= a(0, 0) * b(0, 0) \\ &+ a(0, 1) * b(1, 0) \\ &+ a(0, 2) * b(2, 0) \\ &+ a(0, 3) * b(3, 0) \\ &+ a(0, 4) * b(4, 0) \\ &+ a(0, 5) * b(5, 0) \\ &+ a(0, 6) * b(6, 0) \\ &+ a(0, 7) * b(7, 0) \end{aligned}$$

如果使用CUDA Core的话，  
需要8次FMA，所以需要8  
个clk才可以完成一个c(0,0)

# CUDA Core vs Tensor Core



使用一个CUDA Core  
计算  $C = A * B$



要完成 $4 \times 8$ 与 $8 \times 4$ 的计算，  
需要  $8 * 16 = 128$ 个clk才  
可以完成

当然，如果我们有16个  
CUDA core的话，这些计  
算并行，实际上是8个clk。  
为了与Tensor Core比较，  
这里只用一个CUDA Core

# CUDA Core vs Tensor Core

使用 1st Gen. Tensor Core  
计算  $C = A * B$

A(4 x 8)


B(8 x 4)


C(4 x 4)

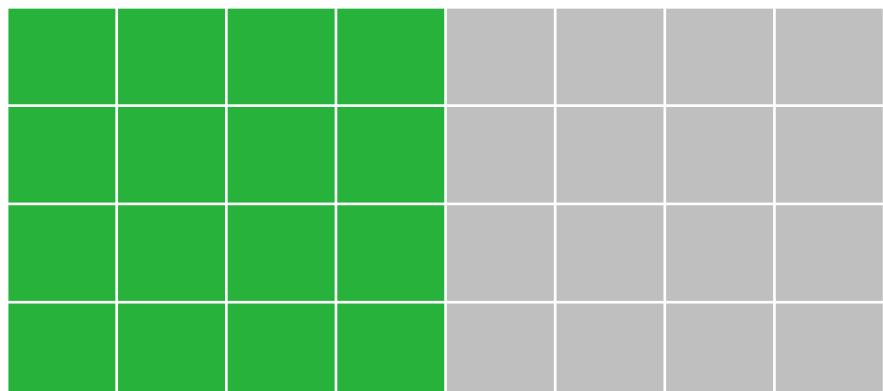

# CUDA Core vs Tensor Core

使用1st Gen. Tensor Core  
计算  $C = A * B$

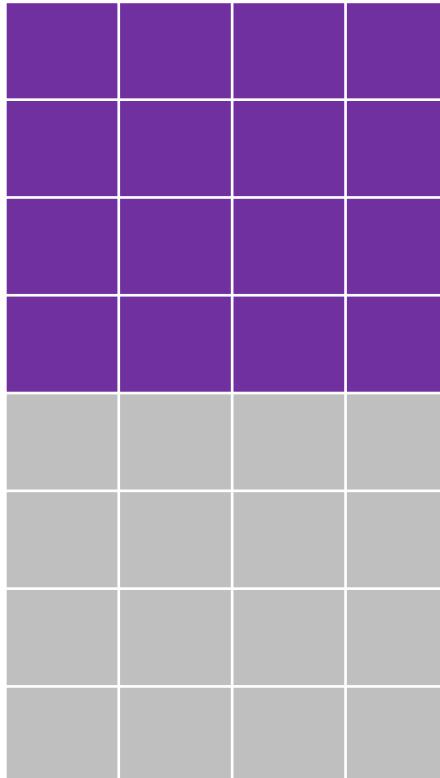
$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix}_{\text{FP16 or FP32}} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix}_{\text{FP16}} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}_{\text{FP16 or FP32}}$$

Tensor Core的矩阵运算处理<sup>[1]</sup>

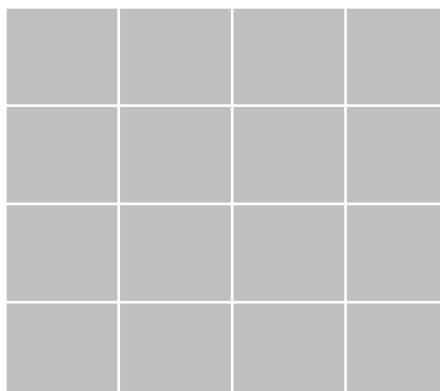
A(4 x 8)



B(8 x 4)



C(4 x 4)



Tensor Core不是1个1个的  
对FP16进行处理，而是4x4  
个FP16一起处理

第一个clk先做A和B的前半段，  
结果先存着

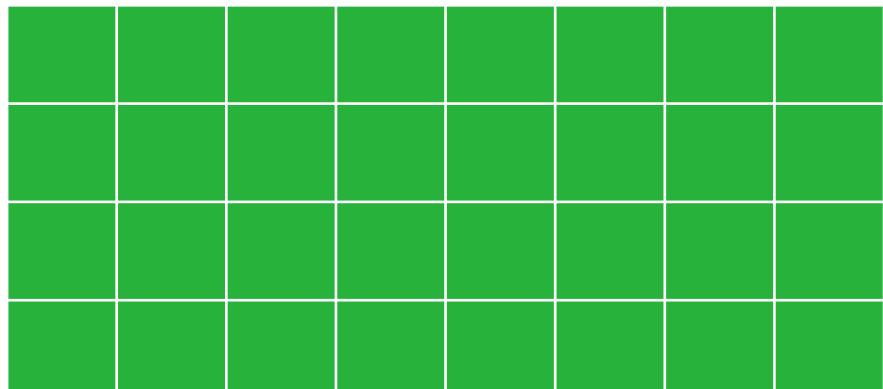
# CUDA Core vs Tensor Core

使用1st Gen. Tensor Core  
计算  $C = A * B$

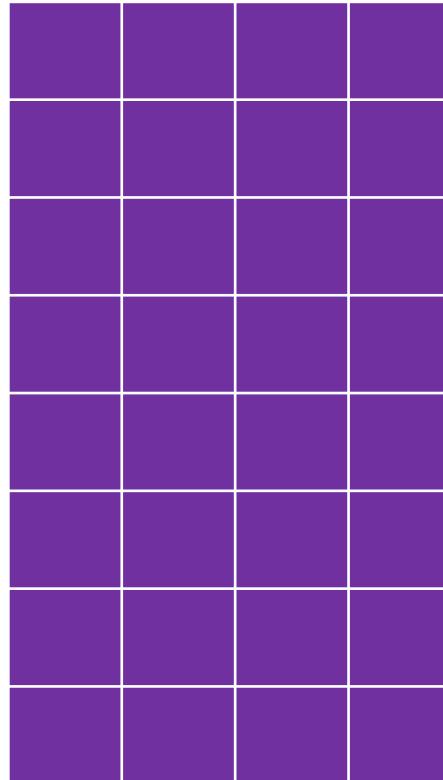
$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix}_{\text{FP16 or FP32}} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix}_{\text{FP16}} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}_{\text{FP16 or FP32}}$$

Tensor Core的矩阵运算处理<sup>[1]</sup>

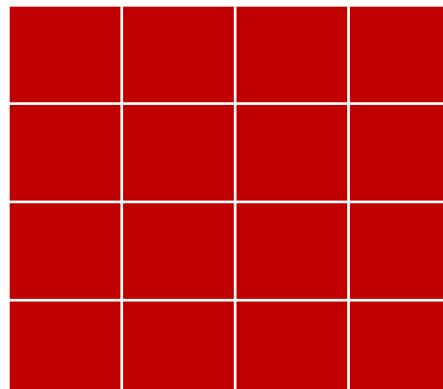
A(4 x 8)



B(8 x 4)



C(4 x 4)



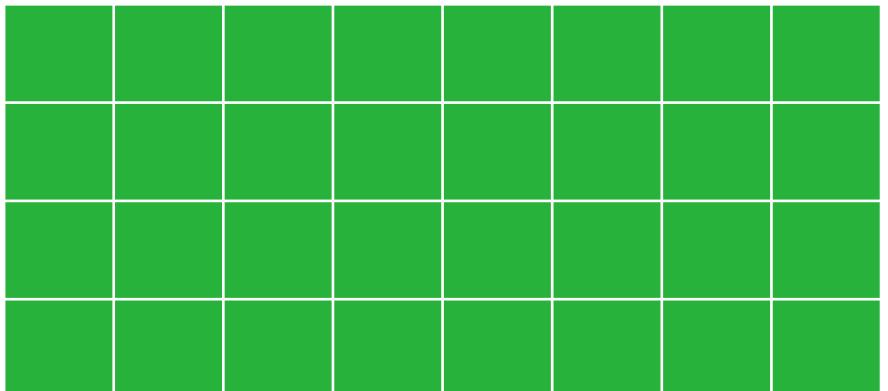
第二个clk再处理A和B的后半段，最后和前半段结果做个累加，完成计算。所以说Tensor Core处理4x8\*8x4的计算只需要 $1 + 1 = 2$ 个clk

<sup>[1]</sup>Programming Tensor Cores in CUDA 9

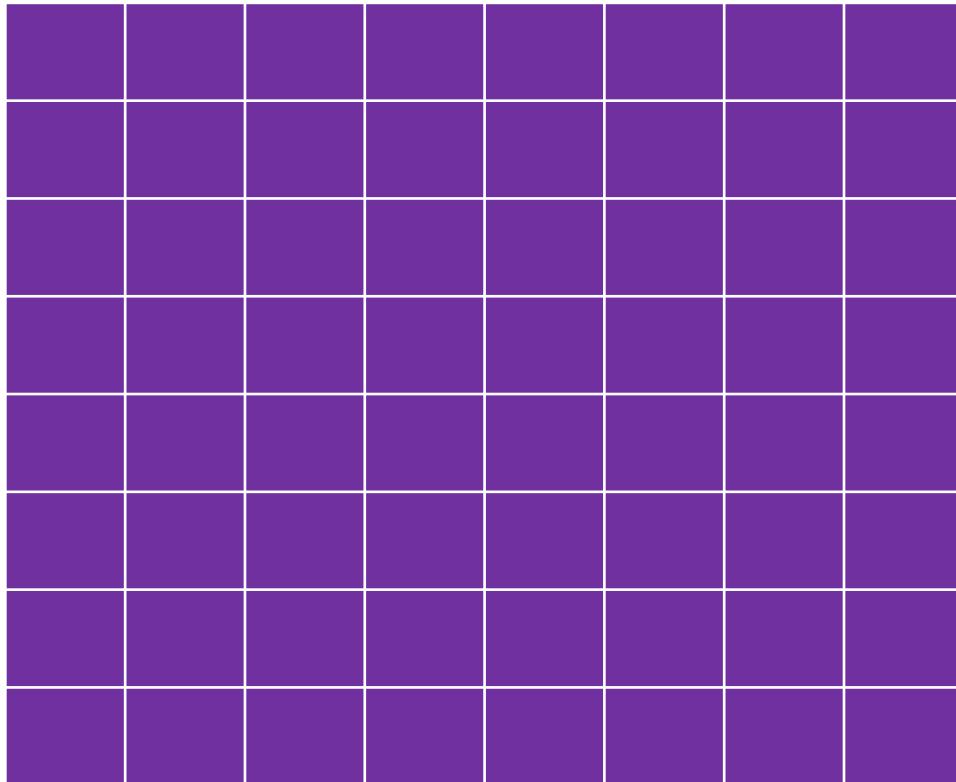
# CUDA Core vs Tensor Core

现在的Ampere架构中的Tensor Core已经是3rd Gen.了，可以1clk处理 $4 \times 8 * 8 \times 8$ 的操作，也就是说1clk可以处理**256个FP16**

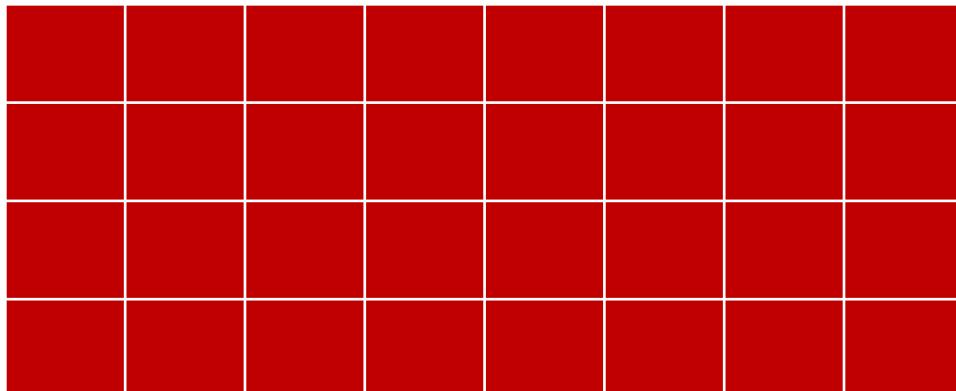
A( $4 \times 8$ )



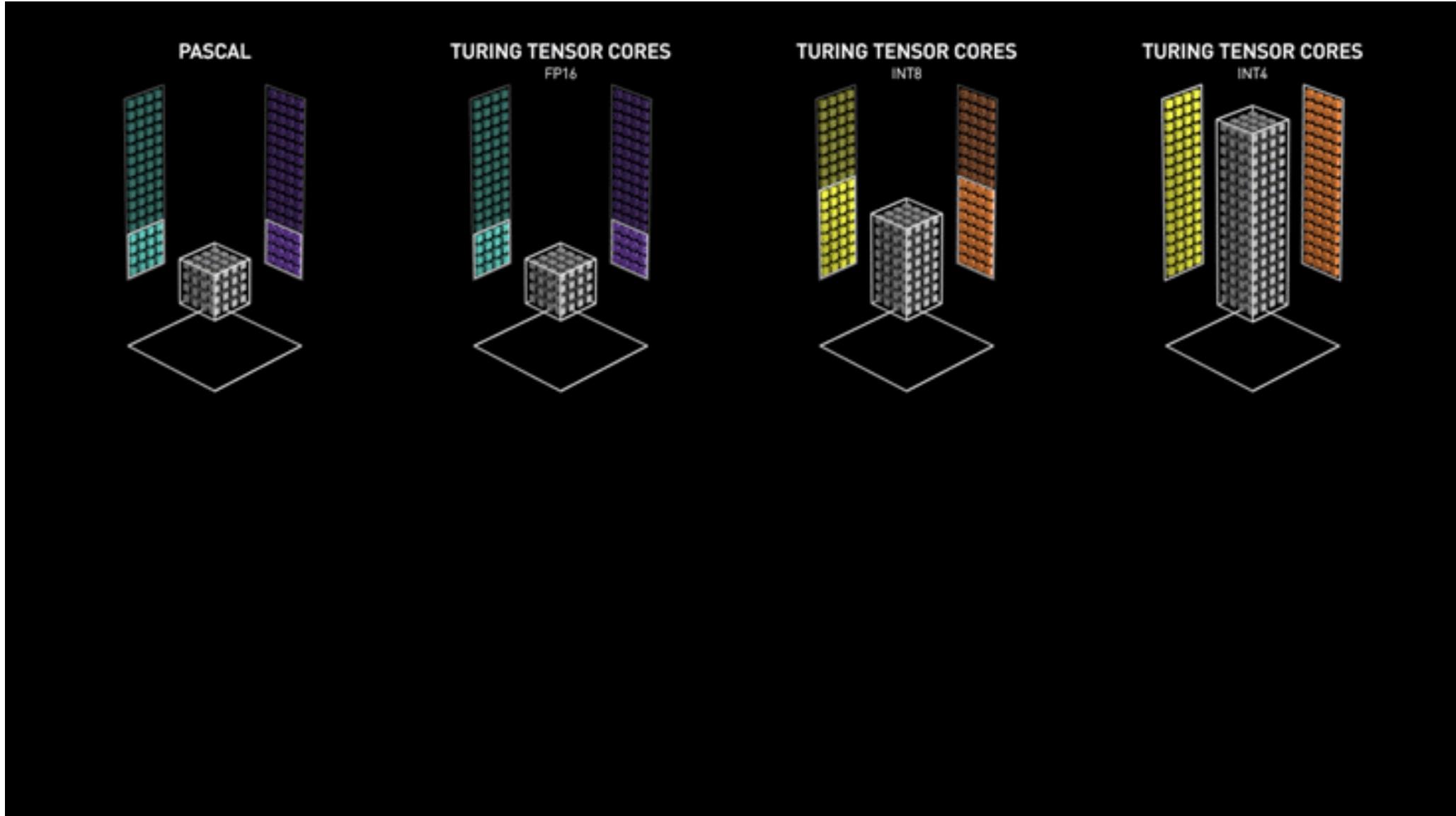
B( $8 \times 8$ )



C( $4 \times 8$ )



# CUDA Core vs Tensor Core



CUDA Core vs Tensor Core<sup>[1]</sup>

<sup>[1]</sup>[YouTube: About NVIDIA Tensor Core](#)

# Ampere SM Architecture

Peak FP16 Tensor Core<sup>1</sup>

312 TFLOPS | 624 TFLOPS<sup>2</sup>



	FP64	FP32	TF32	FP16	INT8	INT4	INT1
CUDA Core	32	64	-	256	256	-	-
Tensor Core	64		512	1024	2048	4096	16484

我们来分析Tensor Core

Ampere架构使用的是第三代Tensor Core，可以一个clk完成一个 $1024 (= 256 * 4)$ 个FP16运算。准确来说是4x8的矩阵与8x8的矩阵的FMA

- 频率: 1.41 GHz
- SM数量: 108
- 一个SM中计算FP16的Tensor core的数量: 4
- 一个Tensor core一个时钟周期可以处理的FP16: 256
- 乘加: 2

$$\text{Throughput} = 1.41 \text{ GHz} * 108 * 4 * 256 * 2 = 312 \text{ TFLOPS}$$

<sup>[1]</sup>AI Chips: A100 GPU with Nvidia Ampere architecture

# Ampere SM Architecture

Peak INT8 Tensor Core<sup>1</sup>

624 TOPS | 1,248 TOPS<sup>2</sup>



Ampere SM Block diagram<sup>[1]</sup>

	FP64	FP32	TF32	FP16	INT8	INT4	INT1
CUDA Core	32	64	-	256	256	-	-
Tensor Core	64		512	1024	2048	4096	16484

## 我们来分析Tensor Core

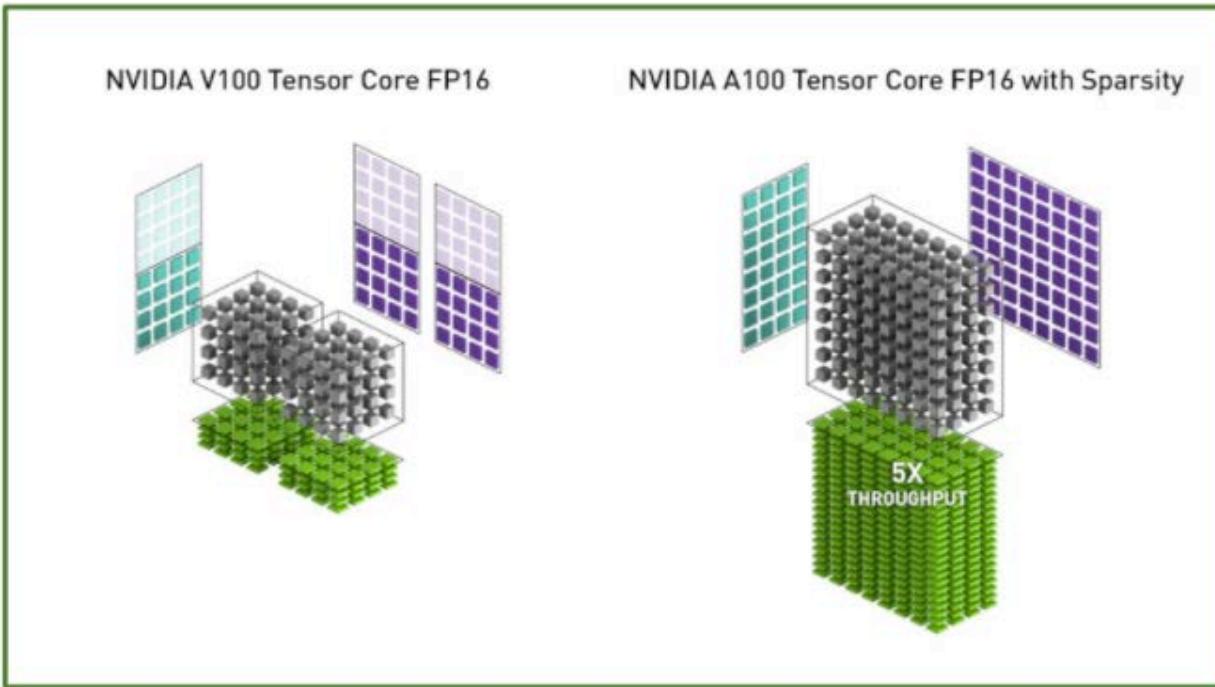
Ampere架构使用的是第三代Tensor Core，可以一个clk完成一个 $2048 (= 256 * 2 * 4)$ 个INT8运算。准确来说是4x8的矩阵与8x8的矩阵的FMA

- 频率: 1.41 GHz
- SM数量: 108
- 一个SM中计算INT8的Tensor core的数量: 4
- 一个Tensor core一个时钟周期可以处理的INT8: 512
- 乘加: 2

$$\text{Throughput} = 1.41 \text{ GHz} * 108 * 4 * 512 * 2 = 624 \text{ TOPS}$$

<sup>[1]</sup>AI Chips: A100 GPU with Nvidia Ampere architecture

# 3<sup>rd</sup> generation Tensor Core

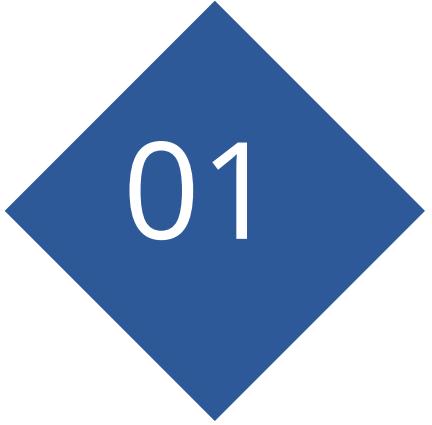


Volta架构Tensor Core (1<sup>st</sup> Gen.)  
与Ampere架构的Tensor Core(3<sup>rd</sup> Gen.)<sup>[1]</sup>

Peak FP64 <sup>1</sup>	9.7 TFLOPS
Peak FP64 Tensor Core <sup>1</sup>	19.5 TFLOPS
Peak FP32 <sup>1</sup>	19.5 TFLOPS
Peak FP16 <sup>1</sup>	78 TFLOPS
Peak BF16 <sup>1</sup>	39 TFLOPS
Peak TF32 Tensor Core <sup>1</sup>	156 TFLOPS   312 TFLOPS <sup>2</sup>
Peak FP16 Tensor Core <sup>1</sup>	312 TFLOPS   624 TFLOPS <sup>2</sup>
Peak BF16 Tensor Core <sup>1</sup>	312 TFLOPS   624 TFLOPS <sup>2</sup>
Peak INT8 Tensor Core <sup>1</sup>	624 TOPS   1,248 TOPS <sup>2</sup>
Peak INT4 Tensor Core <sup>1</sup>	1,248 TOPS   2,496 TOPS <sup>2</sup>

Ampere架构的各个精度的Throughput<sup>[1]</sup>

[1]深度了解 NVIDIA Ampere 架构



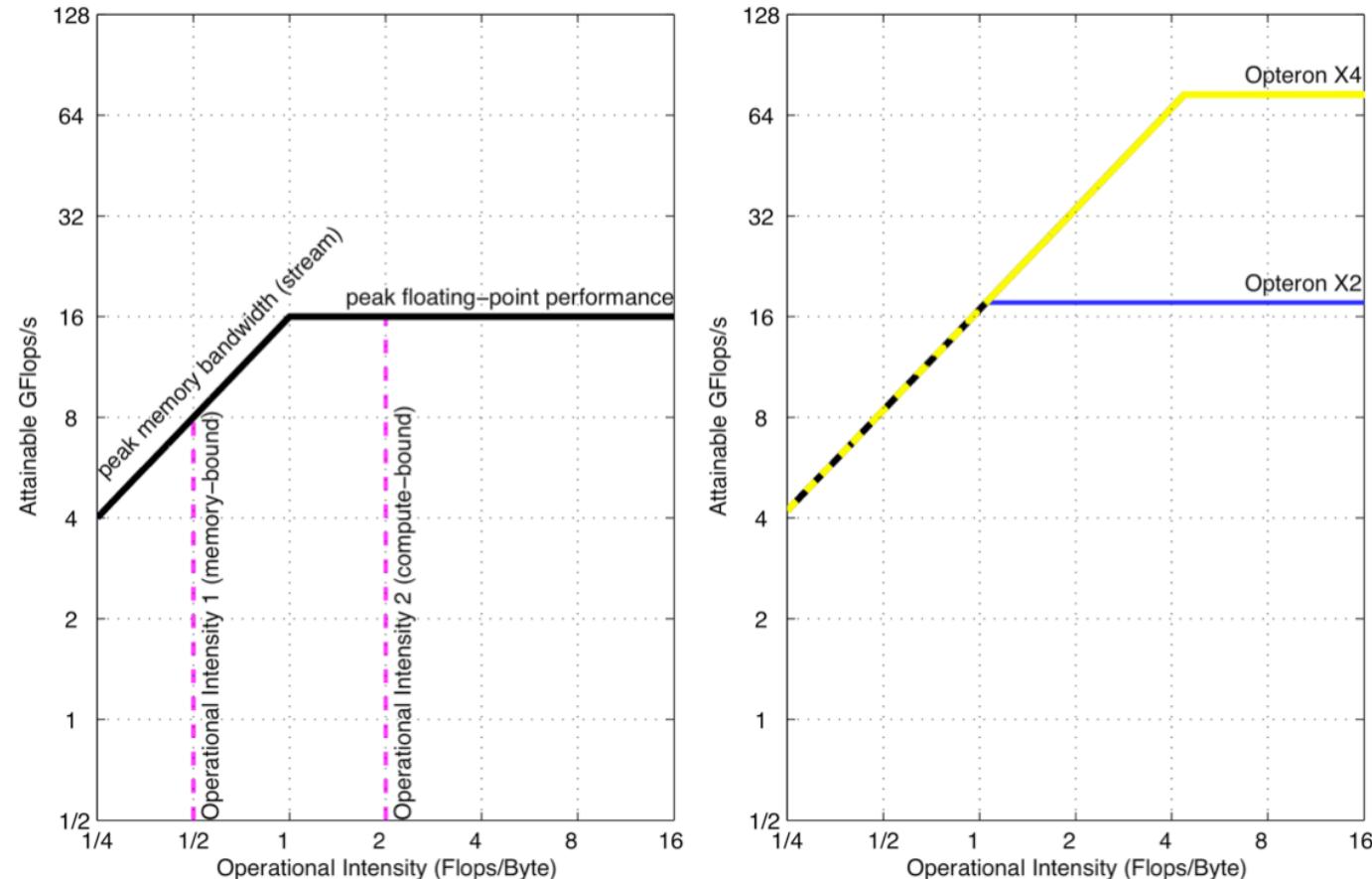
01

## Roofline model

Goal: 理解什么叫做Roofline model, memory bound, compute bound,  
以及各个layer的计算密度的分类

## Roofline model

一个衡量计算机软件/硬件性能的一个分析模型。是David Patterson带领的UC Berkeley的团队与2008年发表的paper中提出的概念。对今后大家使用TensorRT部署模型分析性能是很有帮助



不同的AMD架构的计算峰值、计算密度、带宽分析(左Opteron X2, 右Opteron X4)<sup>[1]</sup>

## Roofline model

为什么需要学习Roofline model?

- 分析 $3 \times 3$  conv,  $5 \times 5$  conv,  $7 \times 7$  conv,  $9 \times 9$  conv,  $11 \times 11$  conv的计算效率
- $1 \times 1$  conv的计算效率
- depthwise conv的计算效率
- 分析目前计算的瓶颈(bottleneck)
- 分析模型的可以优化的上限
- ....

## 回顾一下

- 计算量
- 计算峰值
- 参数量
- 访存量
- 带宽

# 回顾一下

- 计算量
- 计算峰值
- 参数量
- 访存量
- 带宽

单位是FLOPs，表示模型中有多少个floating point operations。是衡量模型大小的标准

name	pretrain	resolution	acc@1	acc@5	#params	FLOPs	FPS	22K model
Swin-T	ImageNet-1K	224x224	81.2	95.5	28M	4.5G	755	-
Swin-S	ImageNet-1K	224x224	83.2	96.2	50M	8.7G	437	-
Swin-B	ImageNet-1K	224x224	83.5	96.5	88M	15.4G	278	-
Swin-B	ImageNet-1K	384x384	84.5	97.0	88M	47.1G	85	-
Swin-T	ImageNet-22K	224x224	80.9	96.0	28M	4.5G	755	<a href="#">github/baidu/config</a>
Swin-S	ImageNet-22K	224x224	83.2	97.0	50M	8.7G	437	<a href="#">github/baidu/config</a>
Swin-B	ImageNet-22K	224x224	85.2	97.5	88M	15.4G	278	<a href="#">github/baidu/config</a>
Swin-B	ImageNet-22K	384x384	86.4	98.0	88M	47.1G	85	<a href="#">github/baidu</a>
Swin-L	ImageNet-22K	224x224	86.3	97.9	197M	34.5G	141	<a href="#">github/baidu/config</a>
Swin-L	ImageNet-22K	384x384	87.3	98.2	197M	103.9G	42	<a href="#">github/baidu</a>

Swin Transformer中的FLOPs<sup>[1]</sup>

## 回顾一下

- 计算量
- **计算峰值**
- 参数量
- 访存量
- 带宽

单位是FLOPS (也可以是FLOP/s),  
表示计算机每秒可以执行的  
*floating point operations*。是衡量  
计算机性能的标准

Peak FP64 <sup>1</sup>	9.7 TFLOPS
Peak FP64 Tensor Core <sup>1</sup>	19.5 TFLOPS
Peak FP32 <sup>1</sup>	19.5 TFLOPS
Peak FP16 <sup>1</sup>	78 TFLOPS
Peak BF16 <sup>1</sup>	39 TFLOPS
Peak TF32 Tensor Core <sup>1</sup>	156 TFLOPS   312 TFLOPS <sup>2</sup>
Peak FP16 Tensor Core <sup>1</sup>	312 TFLOPS   624 TFLOPS <sup>2</sup>
Peak BF16 Tensor Core <sup>1</sup>	312 TFLOPS   624 TFLOPS <sup>2</sup>
Peak INT8 Tensor Core <sup>1</sup>	624 TOPS   1,248 TOPS <sup>2</sup>
Peak INT4 Tensor Core <sup>1</sup>	1,248 TOPS   2,496 TOPS <sup>2</sup>

NVIDIA Ampere架构的TESLA A100的性能<sup>[1]</sup>

<sup>[1]</sup>[AI Chips: A100 GPU with Nvidia Ampere architecture](#)

# 回顾一下

- 计算量
- 计算峰值
- **参数量**
- 访存量
- 带宽

单位是Byte，表示模型中所有的weights(主要在conv和FC中)的量。是衡量模型大小的标准

name	pretrain	resolution	acc@1	acc@5	#params	FLOPs	FPS	22K model
Swin-T	ImageNet-1K	224x224	81.2	95.5	28M	4.5G	755	-
Swin-S	ImageNet-1K	224x224	83.2	96.2	50M	8.7G	437	-
Swin-B	ImageNet-1K	224x224	83.5	96.5	88M	15.4G	278	-
Swin-B	ImageNet-1K	384x384	84.5	97.0	88M	47.1G	85	-
Swin-T	ImageNet-22K	224x224	80.9	96.0	28M	4.5G	755	<a href="#">github/baidu/config</a>
Swin-S	ImageNet-22K	224x224	83.2	97.0	50M	8.7G	437	<a href="#">github/baidu/config</a>
Swin-B	ImageNet-22K	224x224	85.2	97.5	88M	15.4G	278	<a href="#">github/baidu/config</a>
Swin-B	ImageNet-22K	384x384	86.4	98.0	88M	47.1G	85	<a href="#">github/baidu</a>
Swin-L	ImageNet-22K	224x224	86.3	97.9	197M	34.5G	141	<a href="#">github/baidu/config</a>
Swin-L	ImageNet-22K	384x384	87.3	98.2	197M	103.9G	42	<a href="#">github/baidu</a>

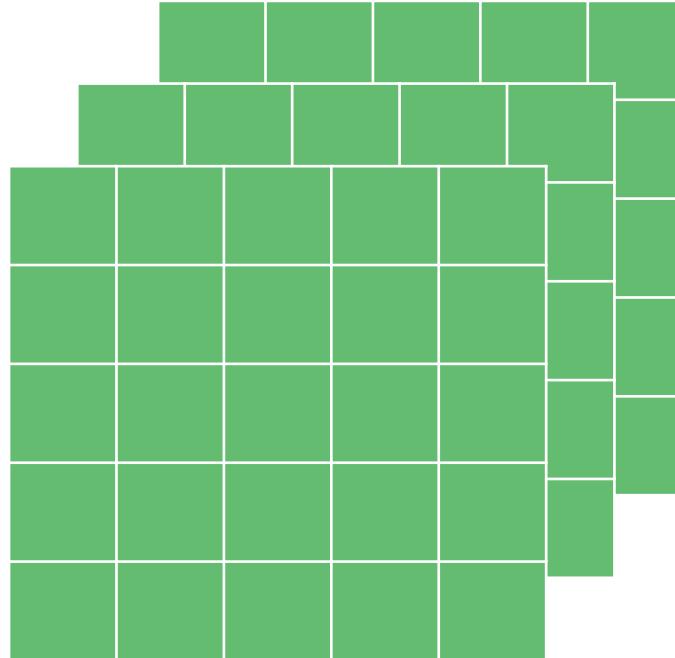
Swin Transformer中的FLOPs<sup>[1]</sup>

## 回顾一下

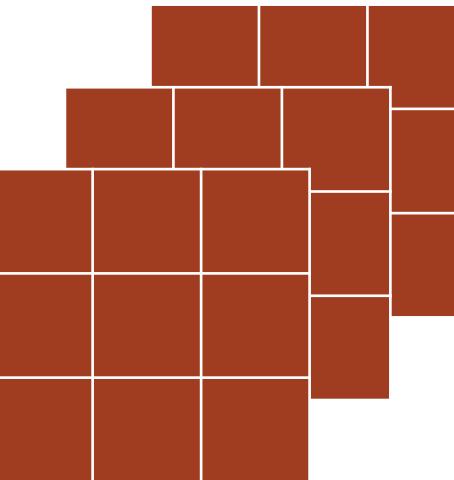
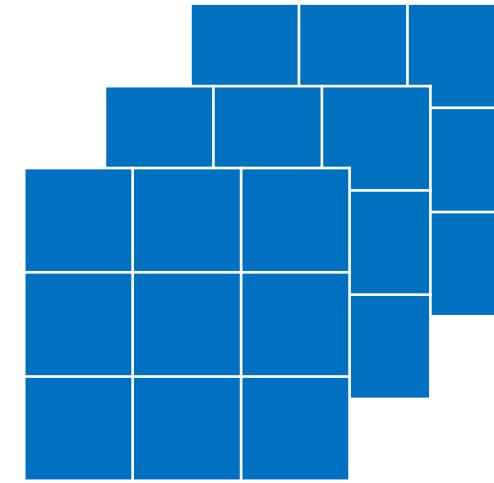
- 计算量
- 计算峰值
- 参数量
- **访存量**
- 带宽

单位是Byte，表示模型中某一个算子，或者某一层layer进行计算时需要与memory产生read/write的量。是分析模型中某些计算的计算效率的标准之一

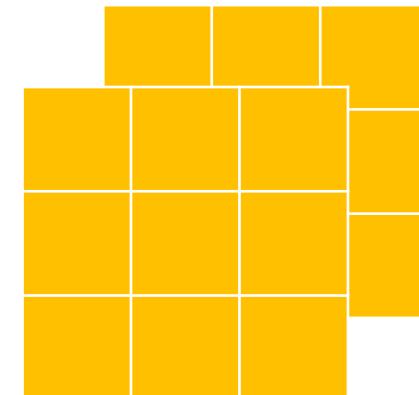
**陷阱：**参数量和访存量的单位都是byte，但不一样。conv的参数量就是weight的大小，跟input/output无关。transformer的参数会根据输入tensor大小改变而改变



input tensor: 5x5x3



kernel : 3x3x3, 2



output tensor: 3x3x2

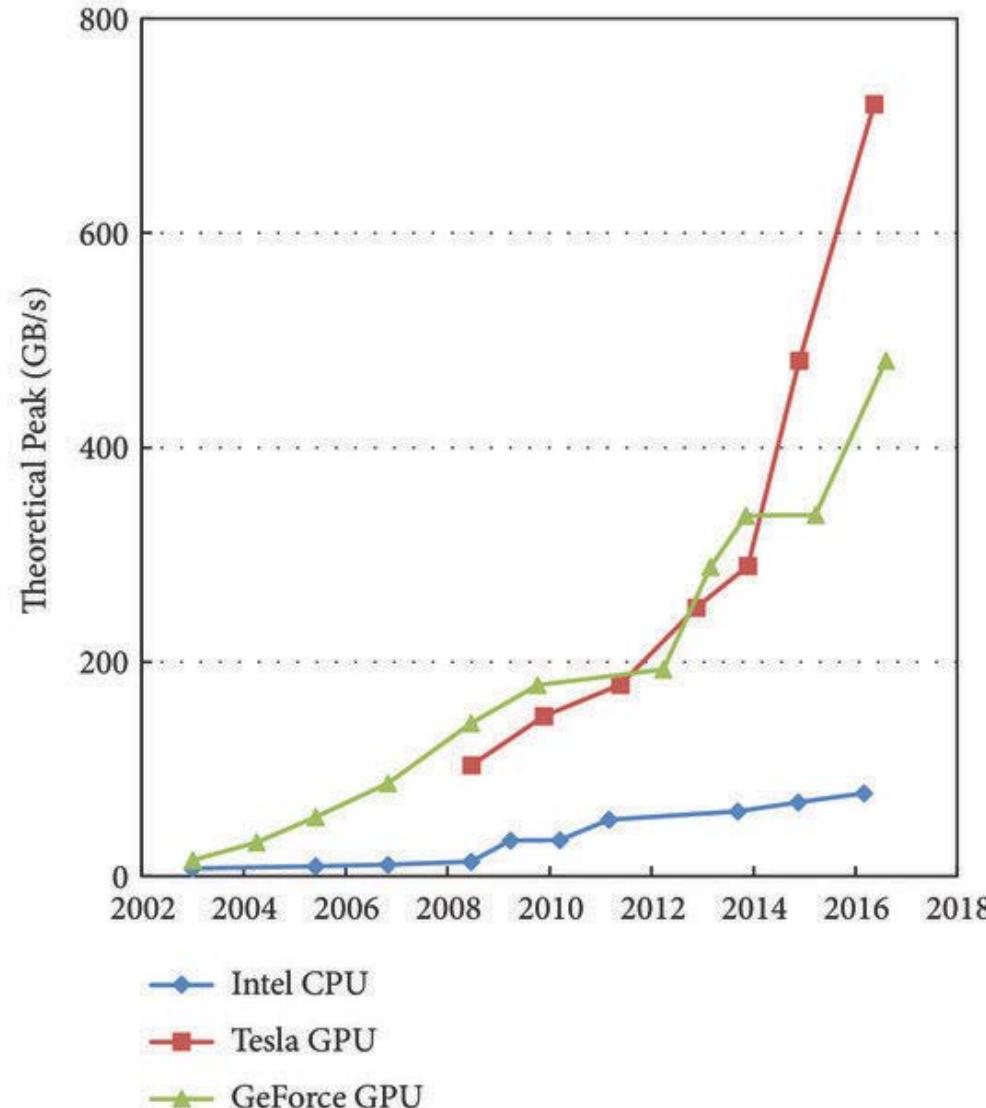
所需要的访存量：  
 $(3 \times 3 \times 3 \times 2 + 3 \times 3 \times 2) * 4 = 588 \text{ Byte} = 0.12 \text{ KB}$

## 回顾一下

- 计算量
- 计算峰值
- 参数数量
- 访存量
- 带宽

单位是Byte/s, 全称是memory bandwidth, 表示的是单位时间内可以传输的数据量的多少。是衡量计算机硬件memor性能的一个标准

非常重要，但最容易被忽略的点  
大家经常听到的  
DDR/DDR2/DDR3/DDR4/GDDR表示的是这个



CPU和GPU在带宽上的比较(2002~2018)  
普遍上GPU的带宽要比CPU高<sup>[1]</sup>

<sup>[1]</sup>[A Multi-GPU Parallel Algorithm in Hypersonic Flow Computations](#)

# 回顾一下

- 计算量
- 计算峰值
- 参数量
- 访存量
- 带宽



memory clock

(表示的是单位时间内可以read/write的频率(Hz), 一般以GHz为基本单位)

带宽跟以下因素有关:

- memory clock (GHz)
- memory bus width (Byte)
- memory channel



memory bus width

(如同字面意思, 表示的是可以同时读写的数据多少。单位是Byte)



memory channel

(表示的是通道数量, 越多越好)

## 回顾一下

- 计算量
- 计算峰值
- 参数数量
- 访存量
- 带宽

### 举例

- Intel Xeon Gold 6000 (server)
  - memory: DDR4-2666
  - memory clock: 2666 MHz
  - memory bus width: 8 Bytes
  - memory channel: 6
  - => memory bandwidth =  $2666 \text{ MHz} * 8 \text{ Bytes} * 6 = 128\text{GB/s}$
- Intel Core i7-8700 (desktop/laptop)
  - memory: DDR4-2666
  - memory clock: 2666 MHz
  - memory bus width: 8 Bytes
  - memory channel: 2
  - => memory bandwidth =  $2666 \text{ MHz} * 8 \text{ Bytes} * 2 = 42\text{GB/s}$

## 回顾一下

- 计算量
- 计算峰值
- 参数数量
- 访存量
- 带宽

### 举例

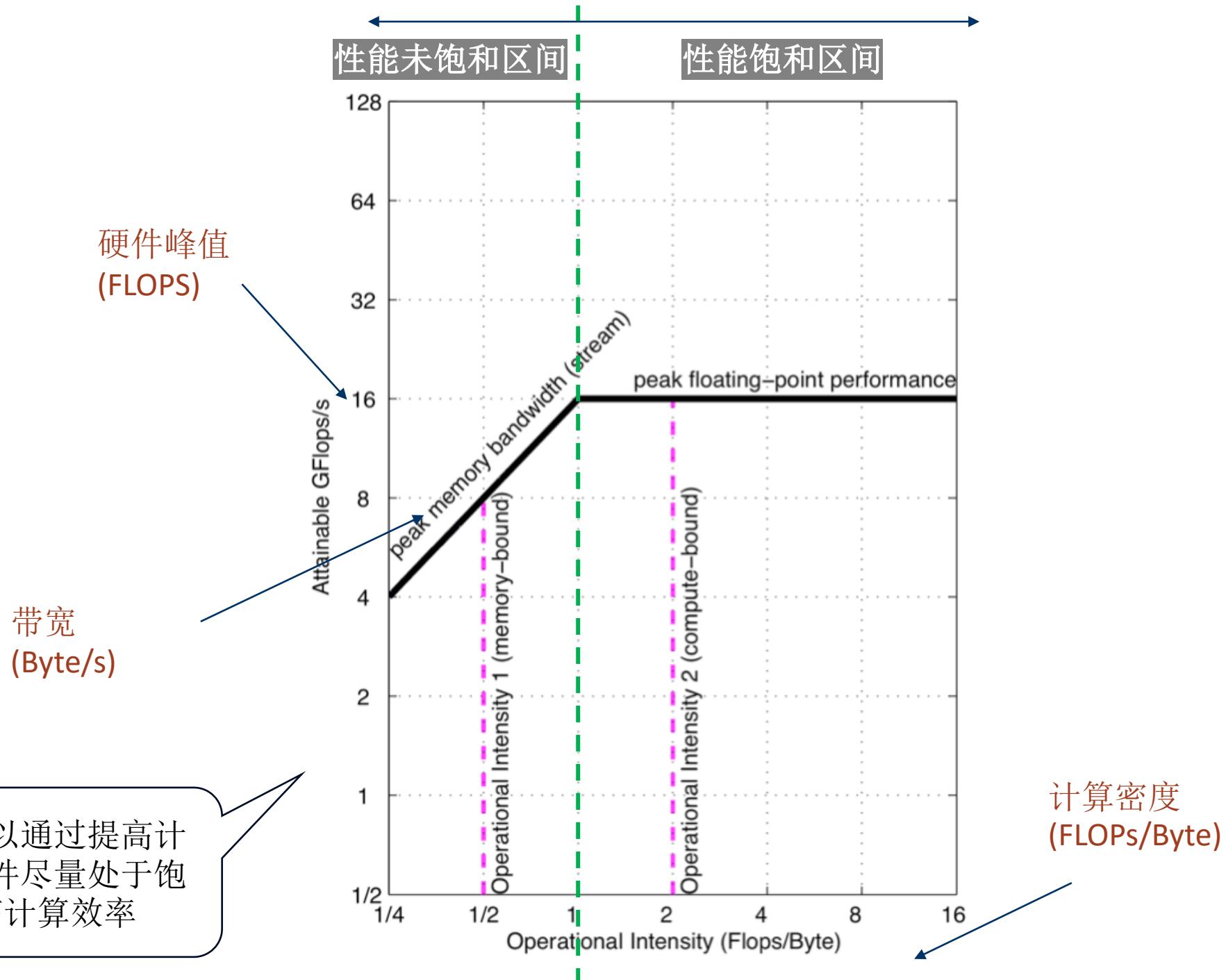
- NVIDIA Quadro RTX 6000
  - memory: GDDR6
  - memory clock: 1750 MHz
  - memory clock effective:  $1750 \text{ MHz} * 8 = 14\text{Gbps}$
  - memory interface width: 48 Bytes (384 bits)
- $\Rightarrow \text{memory bindwidth} = 14 \text{ Gbps} * 48 \text{ Bytes} * 1 = 672\text{GB/s}$
- NVIDIA Tesla V100
  - memory: HBM2
  - memory clock: 877 MHz
  - memory interface width: 512 Bytes (4096 bits)
- $\Rightarrow \text{memory bindwidth} = 877 \text{ MHz} * 512 \text{ Bytes} * 2 = 898\text{GB/s}$

# Roofline model

## Operational intensity (计算密度)

单位是FLOPs/Byte，表示的是传送单位数据可以进行的浮点运算数。

$$\text{计算密度} = \frac{\text{计算量}}{\text{访存量}}$$



# Roofline model (性能分析)

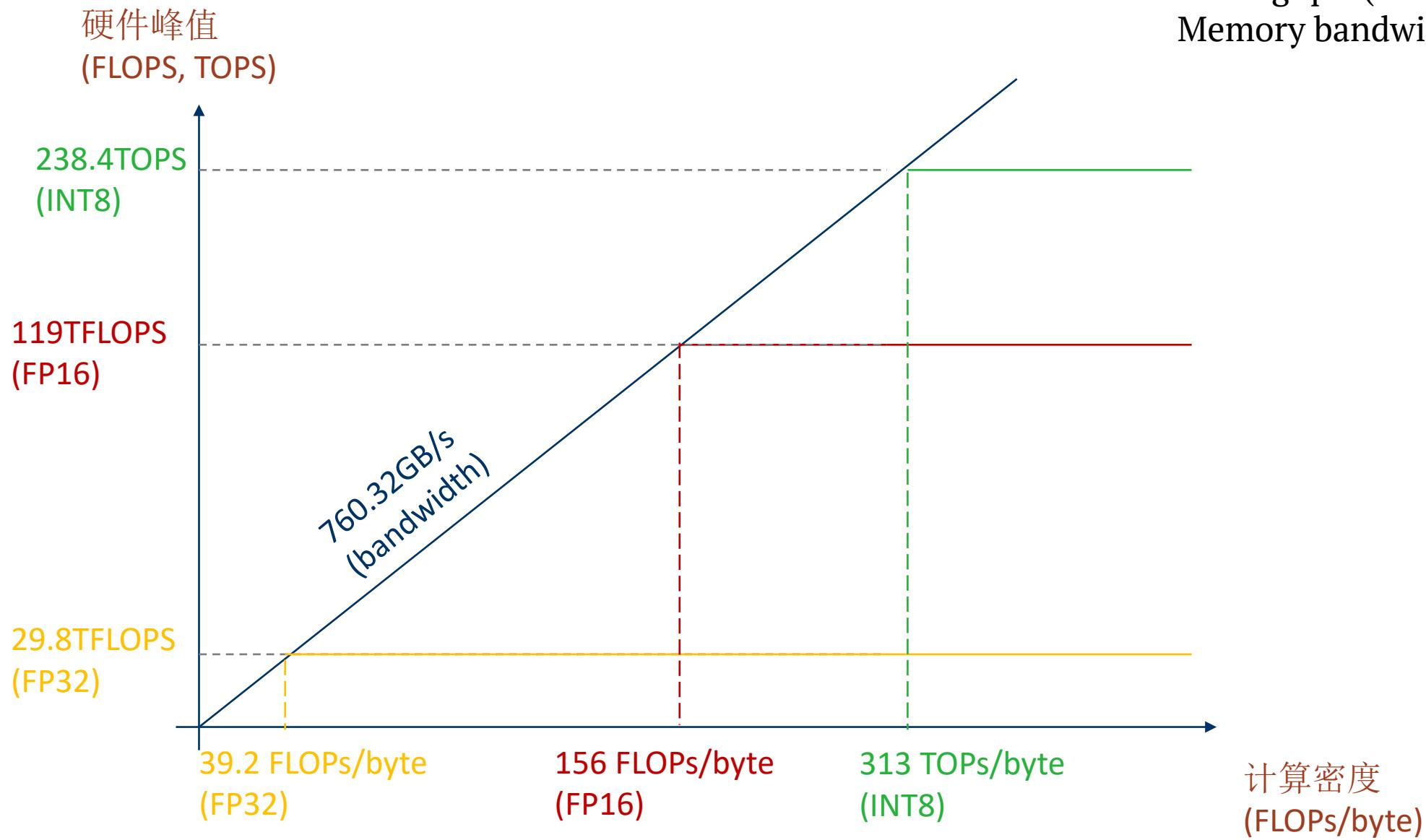


## 3080 Ampere 架构

- Core种类与数量
  - 8704 CUDA cores
  - 272 Tensor cores
  - 68 SMs
- 计算峰值
  - (FP32) 29.8 TFLOPS
  - (FP16) 119 TFLOPS
  - (INT8) 238 TOPS
- 带宽
  - 760.32 GB/s
- 频率
  - 1.7GHz



# Roofline model (性能分析)



Throughput(FP32): 29.8 TFLOPS  
Throughput(FP16): 119 TFLOPS  
Throughput(INT8): 238 TOPS  
Memory bandwidth: 760.32GB/s

## Roofline model (性能分析)

	性能未饱和	性能已饱和
FP32	< 39.2 TFLOPs/byte	$\geq 39.2$ TFLOPs/byte
FP16	< 156 TFLOPs/byte	$\geq 156$ TFLOPs/byte
INT8	< 313 TOPs/byte	$\geq 313$ TOPs/byte

RTX 3080 Ampere架构的计算密度与性能饱和的关系表

# Operational intensity (计算密度) – kernel size的影响(FP32)

单位是FLOPs/Byte，表示的是传送单位数据可以进行的浮点运算数。

$$\text{计算密度} = \frac{\text{计算量}}{\text{访存量}}$$

$$\begin{aligned}\text{计算量: } & M^2 * K^2 * C_{in} * C_{out} \\ \text{访存量: } & (K^2 * C_{in} * C_{out} + M^2 * C_{out}) * 4\end{aligned}$$

M: 卷积核输出特征图的H和W。不同的时候可以用M<sub>h</sub>, M<sub>w</sub>表示  
K: 卷积核大小

elementwise conv

M	K	Cin	Cout	GFLOPs	Memory (Kernel)	Memory (Output)	Memory (All)(MB)	Intensity (FLOPs/Byte)
56	1	256	256	0.205520896	65536	802816	3.473408	59.16981132
56	3	256	256	1.849688064	589824	802816	5.57056	332.0470588
56	5	256	256	5.1380224	1638400	802816	9.764864	526.1744966
56	7	256	256	10.0705239	3211264	802816	16.05632	627.2
56	9	256	256	16.64719258	5308416	802816	24.444928	681.0080429
56	11	256	256	24.86802842	7929856	802816	34.930688	711.9249531

elementwise conv(1x1 conv)的虽然较少计算量，但是计算密度也很低。随着kernel size增大，计算密度增长率逐渐下降

# Operational intensity (计算密度) – output size的影响(FP32)

单位是FLOPs/Byte，表示的是传送单位数据可以进行的浮点运算数。

$$\text{计算密度} = \frac{\text{计算量}}{\text{访存量}}$$

$$\begin{aligned}\text{计算量: } & M^2 * K^2 * C_{in} * C_{out} \\ \text{访存量: } & (K^2 * C_{in} * C_{out} + M^2 * C_{out}) * 4\end{aligned}$$

M: 卷积核输出特征图的H和W。不同的时候可以用M<sub>h</sub>, M<sub>w</sub>表示  
K: 卷积核大小

M	K	Cin	Cout	GFLOPs	Memory (Kernel)	Memory (Output)	Memory (All)(MB)	Intensity (FLOPs/Byte)
7	3	256	256	0.028901376	589824	12544	2.409472	11.99490013
14	3	256	256	0.115605504	589824	50176	2.56	45.1584
28	3	256	256	0.462422016	589824	200704	3.162112	146.238342
56	3	256	256	1.849688064	589824	802816	5.57056	332.0470588
112	3	256	256	7.398752256	589824	3211264	15.204352	486.6206897
224	3	256	256	29.59500902	589824	12845056	53.73952	550.7121951

output size对计算密度的影响，随着output size变大，计算密度的增长率逐渐下降

# Operational intensity (计算密度) – channel size的影响(FP32)

单位是FLOPs/Byte，表示的是传送单位数据可以进行的浮点运算数。

$$\text{计算密度} = \frac{\text{计算量}}{\text{访存量}}$$

$$\begin{aligned}\text{计算量: } & M^2 * K^2 * C_{in} * C_{out} \\ \text{访存量: } & (K^2 * C_{in} * C_{out} + M^2 * C_{out}) * 4\end{aligned}$$

M: 卷积核输出特征图的H和W。不同的时候可以用Mh, Mw表示  
K: 卷积核大小

M	K	Cin	Cout	GFLOPs	Memory (Kernel)	Memory (Output)	Memory (All)(MB)	Intensity (FLOPs/Byte)
56	3	16	16	0.007225344	2304	50176	0.20992	34.4195122
56	3	32	32	0.028901376	9216	100352	0.438272	65.94392523
56	3	64	64	0.115605504	36864	200704	0.950272	121.6551724
56	3	128	128	0.462422016	147456	401408	2.195456	210.6268657
56	3	256	256	1.849688064	589824	802816	5.57056	332.0470588
56	3	512	512	7.398752256	2359296	1605632	15.859712	466.5123967

channel size对计算密度的影响，越大的channel size计算密度越高。

# Operational intensity (计算密度) – group convolution的影响(FP32)

单位是FLOPs/Byte，表示的是传送单位数据可以进行的浮点运算数。

$$\text{计算密度} = \frac{\text{计算量}}{\text{访存量}}$$

$$\begin{aligned}\text{计算量: } & M^2 * K^2 * C_{in} * C_{out} \\ \text{访存量: } & (K^2 * C_{in} * C_{out} + M^2 * C_{out}) * 4\end{aligned}$$

M: 卷积核输出特征图的H和W。不同的时候可以用M<sub>h</sub>, M<sub>w</sub>表示  
K: 卷积核大小

depthwise conv

M	K	Cin	Cout	Group	GFLOPs	Memory (Kernel)	Memory (Output)	Memory (All)(MB)	Intensity (FLOPs/Byte)
56	3	32	32	32	0.000903168	288	100352	0.40256	2.243561208
56	3	32	32	16	0.001806336	576	100352	0.403712	4.474318326
56	3	32	34	8	0.003612672	1152	100352	0.406016	8.897856242
56	3	32	38	4	0.007225344	2304	100352	0.410624	17.59600998
56	3	32	32	2	0.014450688	4608	100352	0.41984	34.4195122
56	3	32	32	1	0.028901376	9216	100352	0.438272	65.94392523

group convolution对计算密度的影响。depthwise虽然降低了计算量，但计算密度也下降的很多

# Operational intensity (计算密度) – tensor reshape的影响(FP32)

单位是FLOPs/Byte，表示的是传送单位数据可以进行的浮点运算数。

$$\text{计算密度} = \frac{\text{计算量}}{\text{访存量}}$$

$$\begin{aligned}\text{计算量: } & M^2 * K^2 * C_{in} * C_{out} \\ \text{访存量: } & (K^2 * C_{in} * C_{out} + M^2 * C_{out}) * 4\end{aligned}$$

M: 卷积核输出特征图的H和W。不同的时候可以用M<sub>h</sub>, M<sub>w</sub>表示  
K: 卷积核大小

M	K	Cin	Cout	GFLOPs	Memory (Kernel)	Memory (Output)	Memory (All)(MB)	Intensity (FLOPs/Byte)
112	3	32	32	0.115605504	9216	401408	1.642496	70.3840399
112	3	32	32	0.115605504	9216	401408	1.642496	70.3840399
112	3	32	32	0.115605504	9216	401408	1.642496	70.3840399
112	3	32	32	0.115605504	9216	401408	1.642496	70.3840399
112	3	32	32	0.115605504	9216	401408	1.642496	70.3840399
112	3	32	32	0.115605504	9216	401408	1.642496	70.3840399
Summary		0.693633024					9.854976	70.3840399

不参与计算的tensor reshape对计算密度的影响  
(模型中没有tensor reshape)

# Operational intensity (计算密度) – tensor reshape的影响(FP32)

单位是FLOPs/Byte，表示的是传送单位数据可以进行的浮点运算数。

$$\text{计算密度} = \frac{\text{计算量}}{\text{访存量}}$$

$$\begin{aligned}\text{计算量: } & M^2 * K^2 * C_{in} * C_{out} \\ \text{访存量: } & (K^2 * C_{in} * C_{out} + M^2 * C_{out}) * 4\end{aligned}$$

M: 卷积核输出特征图的H和W。不同的时候可以用M<sub>h</sub>, M<sub>w</sub>表示  
K: 卷积核大小

M	K	Cin	Cout	GFLOPs	Memory (Kernel)	Memory (Output)	Memory (All)(MB)	Intensity (FLOPs/Byte)
112	3	32	32	0.115605504	9216	401408	1.642496	70.3840399
112	0	32	32	0	0	401408	1.605632	0
112	3	32	32	0.115605504	9216	401408	1.642496	70.3840399
112	0	32	32	0	0	401408	1.605632	0
112	3	32	32	0.115605504	9216	401408	1.642496	70.3840399
112	0	32	32	0	0	401408	1.605632	0
Summary		0.346816512					9.854976	35.59142497

不参与计算的tensor reshape对计算密度的影响  
(模型中有3个tensor reshape)

# Operational intensity (计算密度) – tensor reshape的影响(FP32)

单位是FLOPs/Byte，表示的是传送单位数据可以进行的浮点运算数。

$$\text{计算密度} = \frac{\text{计算量}}{\text{访存量}}$$

$$\begin{aligned}\text{计算量: } & M^2 * K^2 * C_{in} * C_{out} \\ \text{访存量: } & (K^2 * C_{in} * C_{out} + M^2 * C_{out}) * 4\end{aligned}$$

M: 卷积核输出特征图的H和W。不同的时候可以用M<sub>h</sub>, M<sub>w</sub>表示  
K: 卷积核大小

M	K	Cin	Cout	GFLOPs	Memory (Kernel)	Memory (Output)	Memory (All)(MB)	Intensity (FLOPs/Byte)
112	3	32	32	0.115605504	9216	401408	1.642496	70.3840399
112	0	32	32	0	0	401408	1.605632	0
112	0	32	32	0	0	401408	1.605632	0
112	0	32	32	0	0	401408	1.605632	0
112	0	32	32	0	0	401408	1.605632	0
112	0	32	32	0	0	401408	1.605632	0
Summary		0.115605504					9.854976	11.95425667

不参与计算的tensor reshape对计算密度的影响  
(模型中有5个tensor reshape)

# Operational intensity (计算密度) – FC的影响(FP32)

单位是FLOPs/Byte，表示的是传送单位数据可以进行的浮点运算数。

$$\text{计算密度} = \frac{\text{计算量}}{\text{访存量}}$$

$$\begin{aligned}\text{计算量: } & C_{in} * C_{out} \\ \text{访存量: } & (C_{in} * C_{out}) * 4\end{aligned}$$

M: 卷积核输出特征图的H和W。不同的时候可以用M<sub>h</sub>, M<sub>w</sub>表示  
K: 卷积核大小

M	K	Cin	Cout	GFLOPs	Memory (Kernel)	Memory (Output)	Memory (All)(MB)	Intensity (FLOPs/Byte)
-	-	25088	4096	0.102760448	102760448	4096	411.058176	0.249990035
-	-	4096	4096	0.016777216	16777216	4096	67.125248	0.24993898
-	-	4096	1000	0.004096	4096000	1000	16.388	0.24993898

FC对计算密度的影响  
(FC的计算密度非常低的原因在于它的大量的访存)

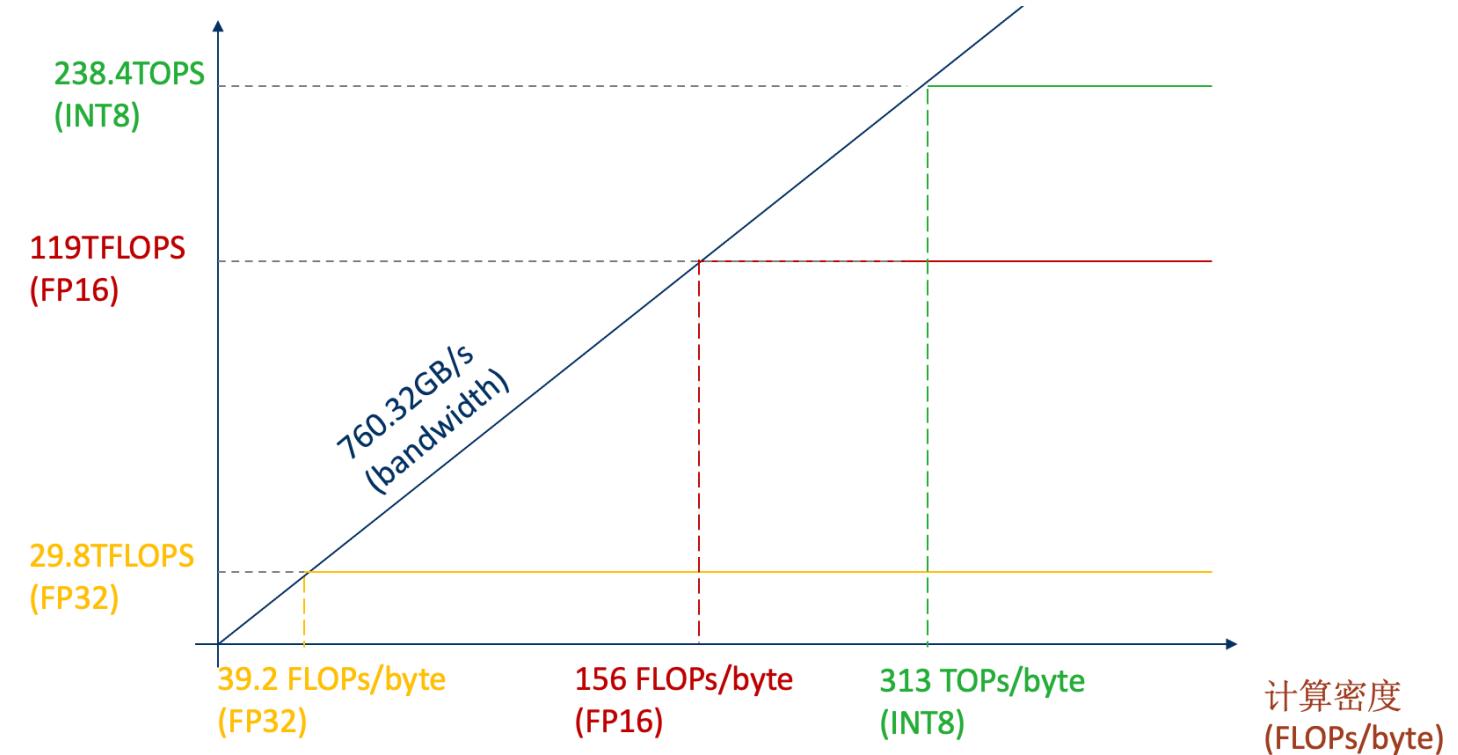
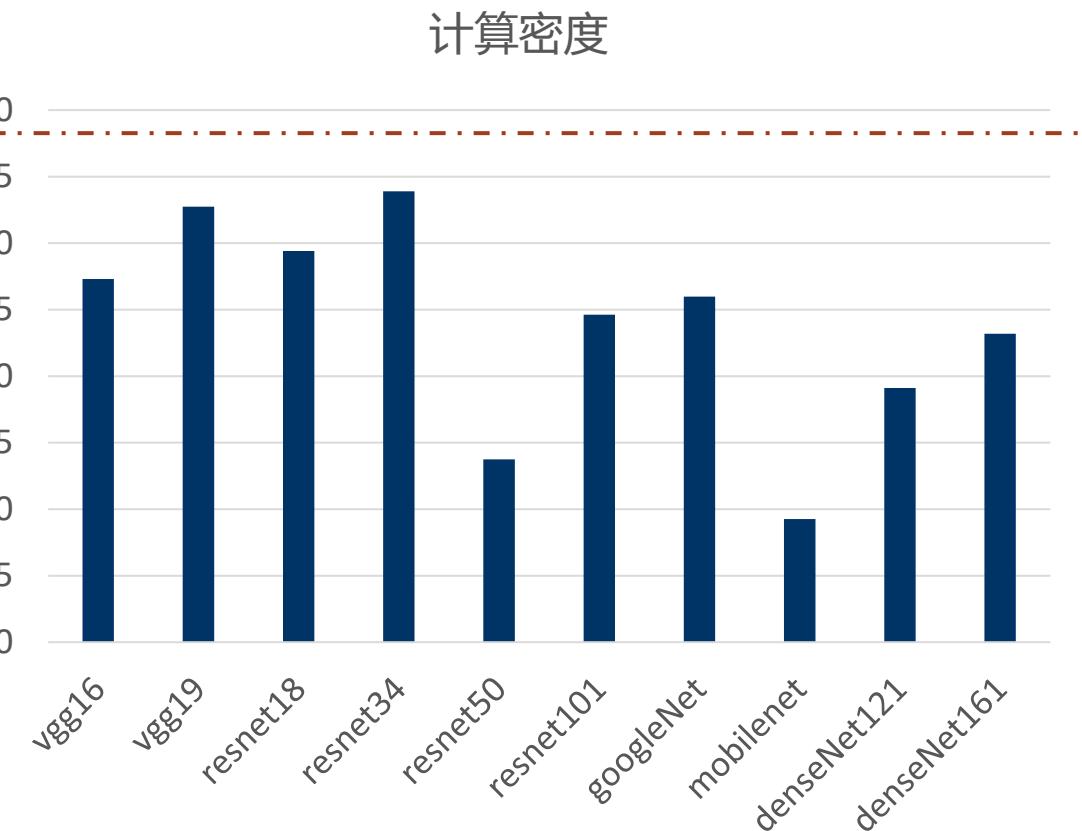
## 模型分析

目前我们单独分析了几个layer对计算密度的影响。但DNN是一个多个layer的组合，所以我们也需要对整个模型进行分析

Model	Memory (kernel)	Memory (output)	Memory (all)	GFLOPs	Intensity (FLOPs/Byte)
vgg16	528	58	586	16	27.30375427
vgg19	548	63	611	20	32.73322422
resnet18	45	23	68	2	29.41176471
resnet34	83	35	118	4	33.89830508
resnet50	98	193	291	4	13.74570447
resnet101	170	155	325	8	24.61538462
googleNet	51	26	77	2	25.97402597
mobilenet	16	38	54	0.5	9.259259259
denseNet121	31	126	157	3	19.10828025
denseNet161	110	235	345	8	23.1884058

## 模型分析

目前我们单独分析了几个layer对计算密度的影响。但DNN是一个多个layer的组合，所以我们也需要对整个模型进行分析



RTX 3080 Ampere架构中FP32的计算在39.2FLOPs/byte才达到计算饱和。所以这些模型其理论上都没有计算饱和

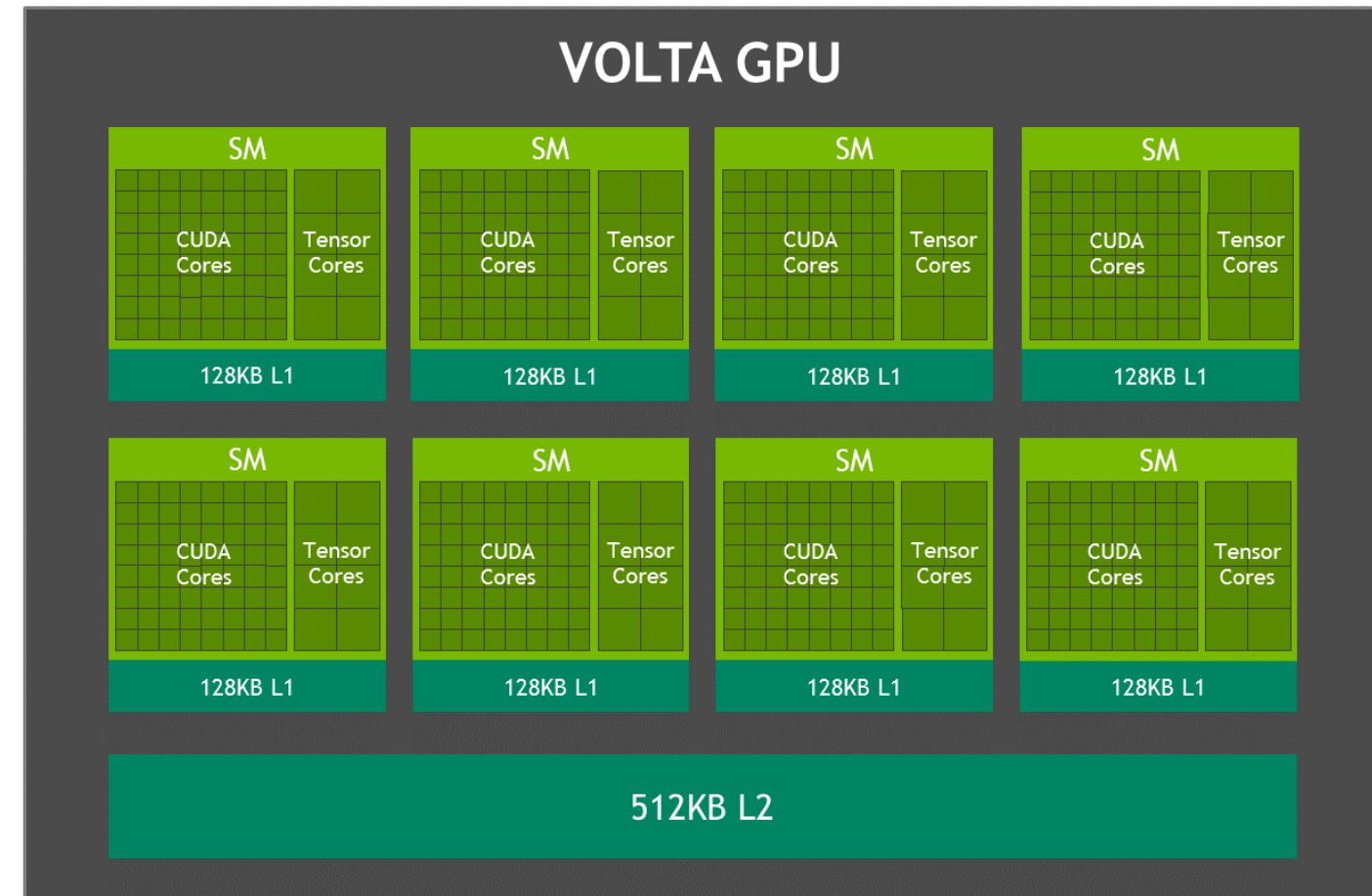
# Roofline model (性能分析)



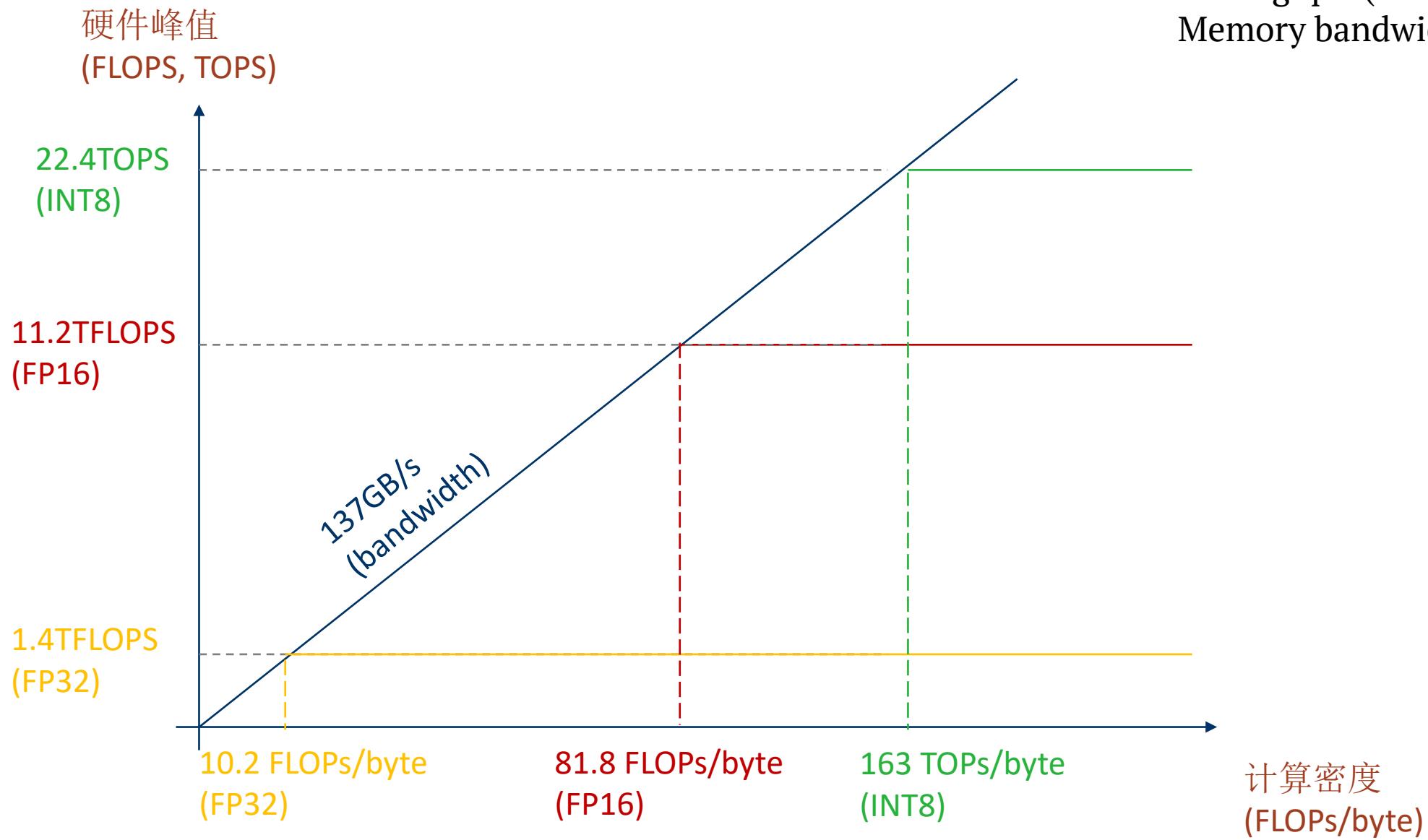
Jetson Xavier AGX Volta架构

- Core种类与数量
  - 512 CUDA cores
  - 64 Tensor cores
  - 8 SMs
- 计算峰值
  - (FP32) 1.4 TFLOPS
  - (FP16) 11 TFLOPS
  - (INT8) 22 TOPS
- 带宽
  - 137 GB/s
- 频率
  - 900MHz

(\*)这里有个陷阱。官网文档说峰值是32TOPS指的是TensorCore和DLA同时在INT8的情况下使用时的峰值。然而实际上Tensor Core与DLA不能同时使用。所以单独使用Tensor Core在INT8下的峰值是22TOPS。



# Roofline model (性能分析)



Throughput(FP32): 1.4 TFLOPS  
Throughput(FP16): 11.2 TFLOPS  
Throughput(INT8): 22.4 TOPS  
Memory bandwidth: 137GB/s

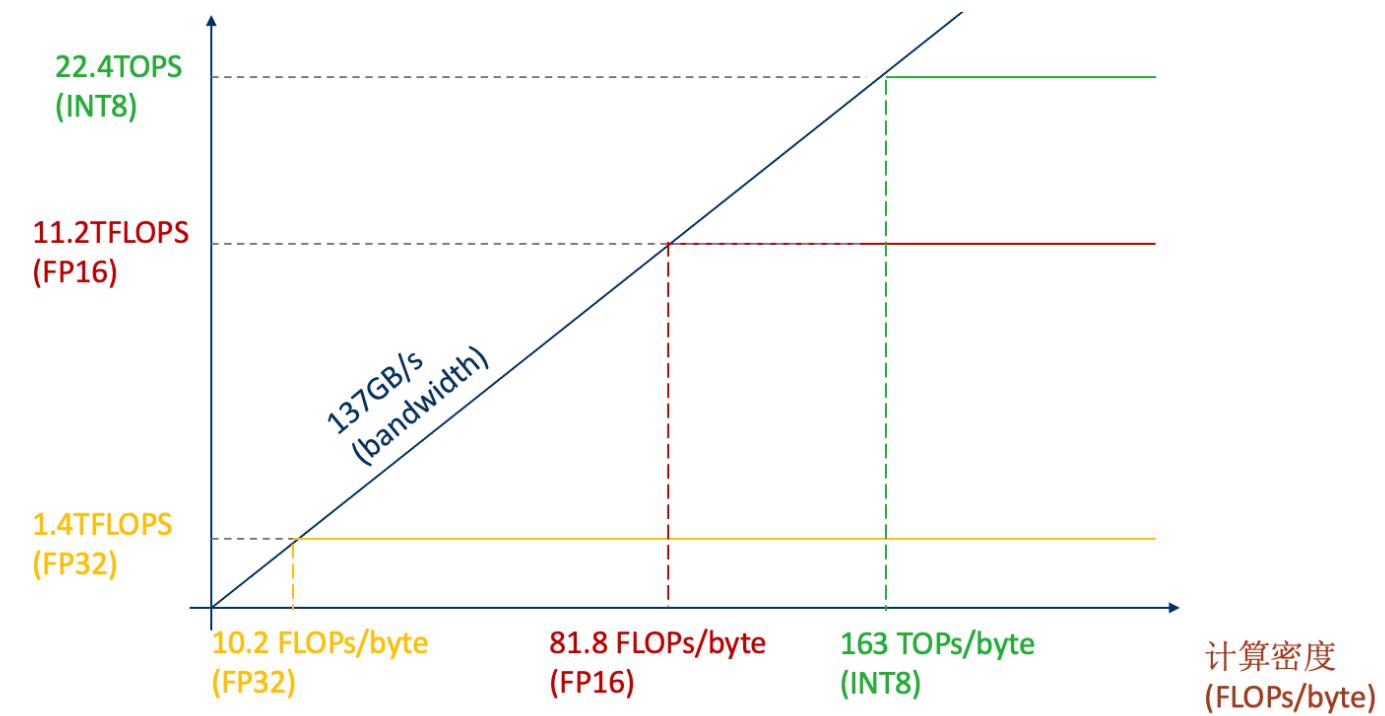
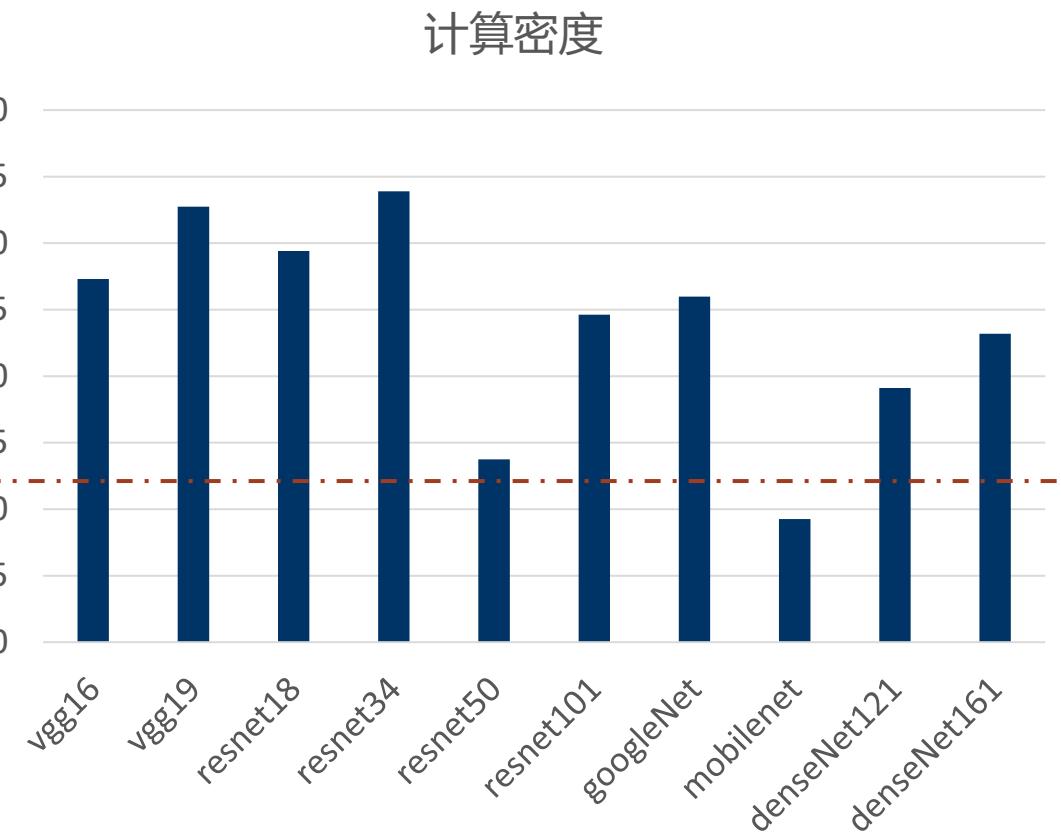
## Roofline model (性能分析)

	性能未饱和	性能已饱和
FP32	< 10.2 FLOPs/byte	$\geq 10.2$ FLOPs/byte
FP16	< 81.8 FLOPs/byte	$\geq 81.8$ FLOPs/byte
INT8	< 163 OPs/byte	$\geq 163$ OPs/byte

Jetson Xavier AGX Volta架构的计算密度与性能饱和的关系表

## 模型分析

目前我们单独分析了几个layer对计算密度的影响。但DNN是一个多个layer的组合，所以我们也需要对整个模型进行分析



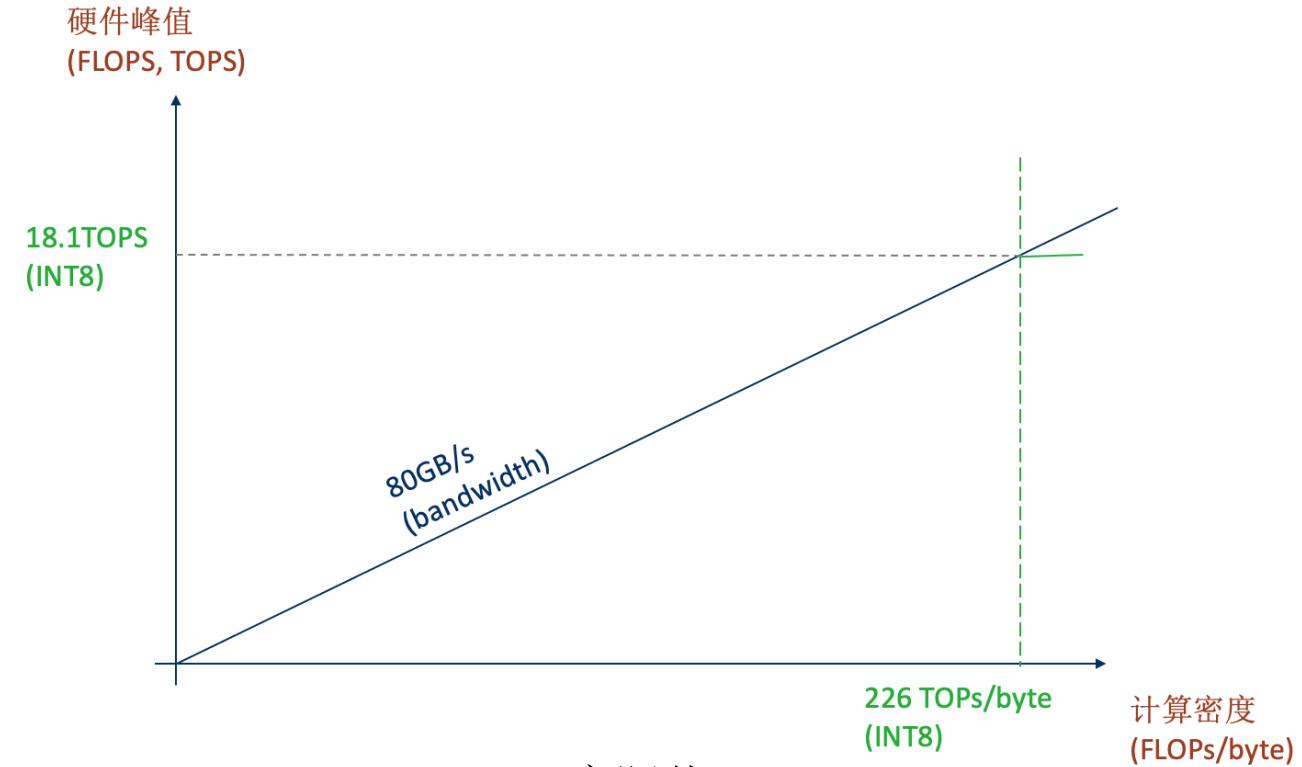
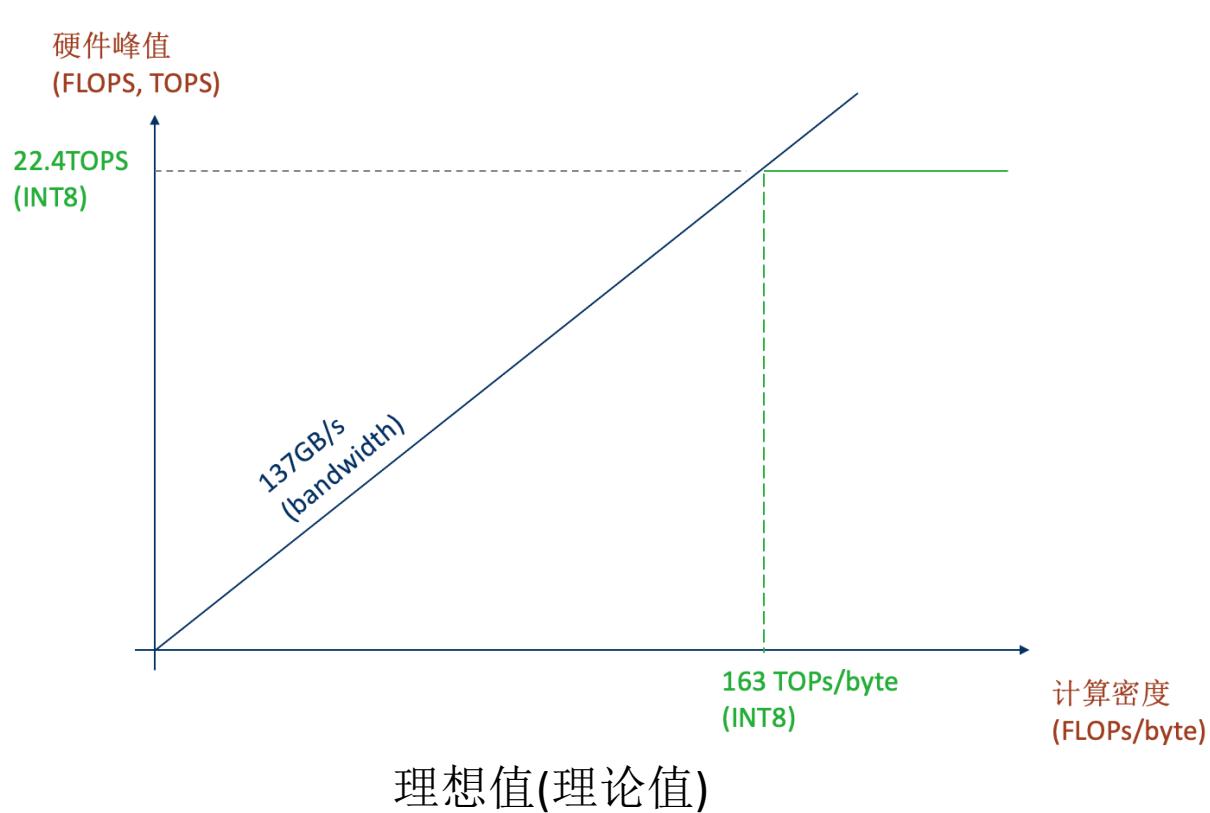
Jetson AGX Xavier架构中FP32的计算在10.2FLOPs/byte就计算饱和。所以这些模型其实都理论上已经计算饱和

# Roofline model (性能分析)

(重点)到目前讲的是理论值。然而实际上我们会发现:

- 峰值可能会小于22.4TOPS
- bandwidth可能会小于137GB/s

需要根据一系列benchmark找到部署架构的真实值。(比如自己写几个计算密集的核函数)





02

## 模型部署的几大误区

Goal: 理解部署中模型优化的过程中容易踩的坑

## 注意点1

### FLOPs不能衡量模型性能

- 因为FLOPs只是模型计算大小的单位
- 还需要考虑
  - 访存量
  - 跟计算无关的DNN部分
    - (reshape, shortcut, nchw2nhwc等等)
  - DNN以外的部分
    - (前处理、后处理这些)

## 注意点2

### 不能够完全依靠TensorRT

- TensorRT可以对模型做适当的优化，但是有上限
- 比如
  - 计算密度低的1x1 conv, depthwise conv不会重构
  - GPU无法优化的地方会到CPU执行
    - 可以手动修改代码实现部分，让部分cpu执行转到gpu执行
  - 有些冗长的计算，TensorRT可能不能优化
    - 直接修改代码实现部分
  - 存在TensorRT尚未支持的算子
    - 可以自己写plugin
  - TensorRT不一定会分配Tensor Core
    - 因为TensorRT kernel auto tuning会选择最合适的kernel

## 注意点3

### CUDA Core和Tensor Core的使用

- 有的时候TensorRT并不会分配Tensor Core
  - kernel auto tuning自动选择最优解
  - 所以有时会出现类似于INT8的速度比FP16反而慢了
  - 使用Tensor Core需要让tensor size为8或者16的倍数

## 注意点4

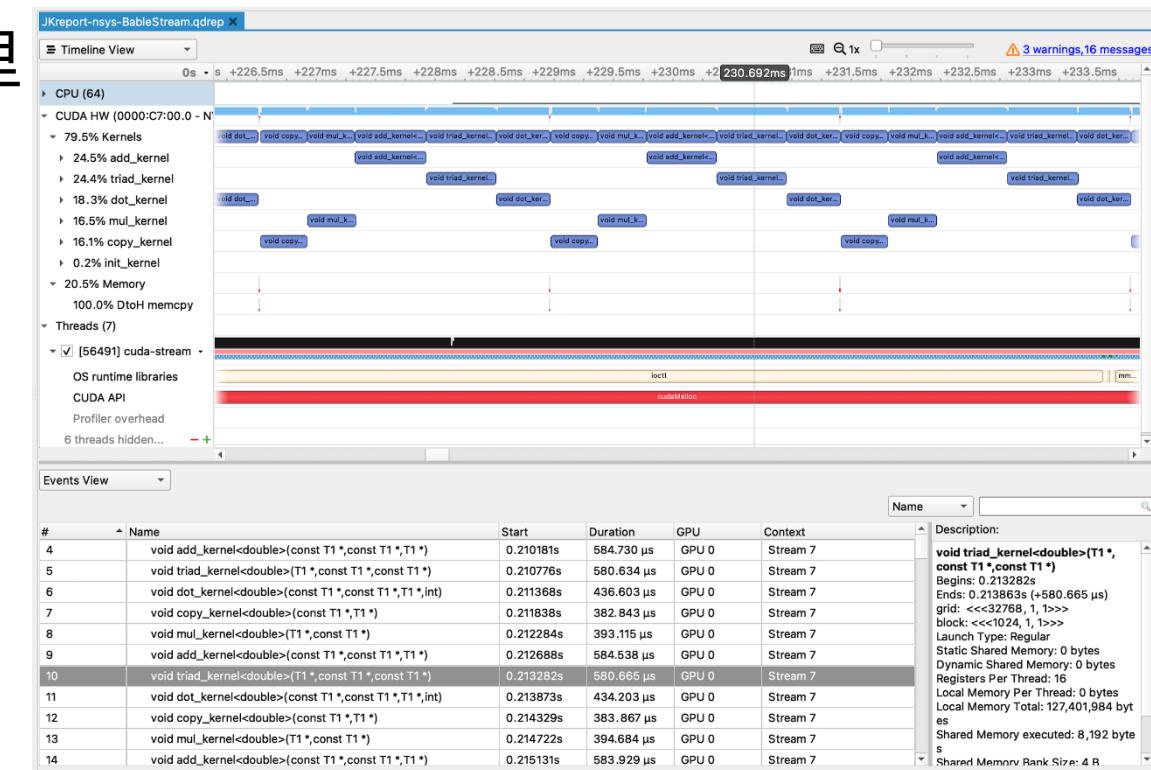
### 不能忽视 前处理/后处理 的overhead

- 对于一些轻量的模型，相比于DNN推理部分，前处理/后处理可能会更耗时间
- 因为有些前处理/后处理的复杂逻辑不适合GPU并行
- 然而有很多种解决办法
  - 可以把前处理/后处理中可并行的地方拿出来让GPU并行话
    - 比如RGB2BGR, Normalization, resize, crop, NCHW2NHWC
  - 可以在cpu上使用一些针对图像处理的优化库
    - 比如Halide
      - 使用Halide进行blur, resize, crop, DBSCAN, sobel这些会比CPU快

## 注意点5

### 对使用TensorRT得到的推理引擎做benchmark和profiling

- 使用TensorRT得到推理引擎并实现infer只是优化的第一步
- 需要使用NVIDIA提供的benchmark tools进行profiling
  - 分析模型瓶颈在哪里
  - 分析模型可进一步优化的地方在哪里
  - 分析模型中多余的memory access在哪里
- 可以使用：
  - nsys, nvprof, dlprof, Nsight这些工具



## 注意点6

其他...



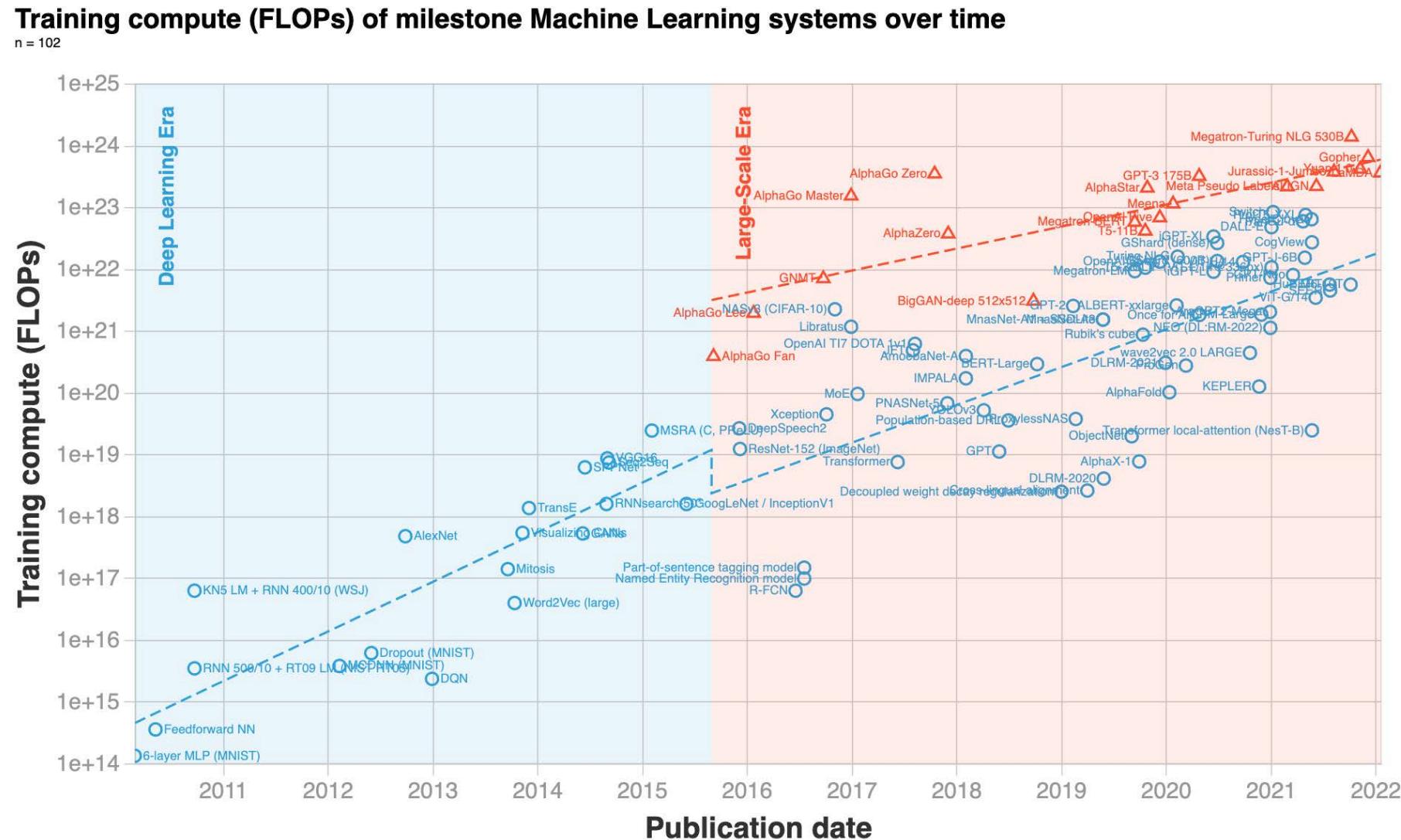
03

## Quantization (mapping and shift)

Goal: 理解量化诞生的背景，量化的计算方法，以及对称量化和非对称量化的区别

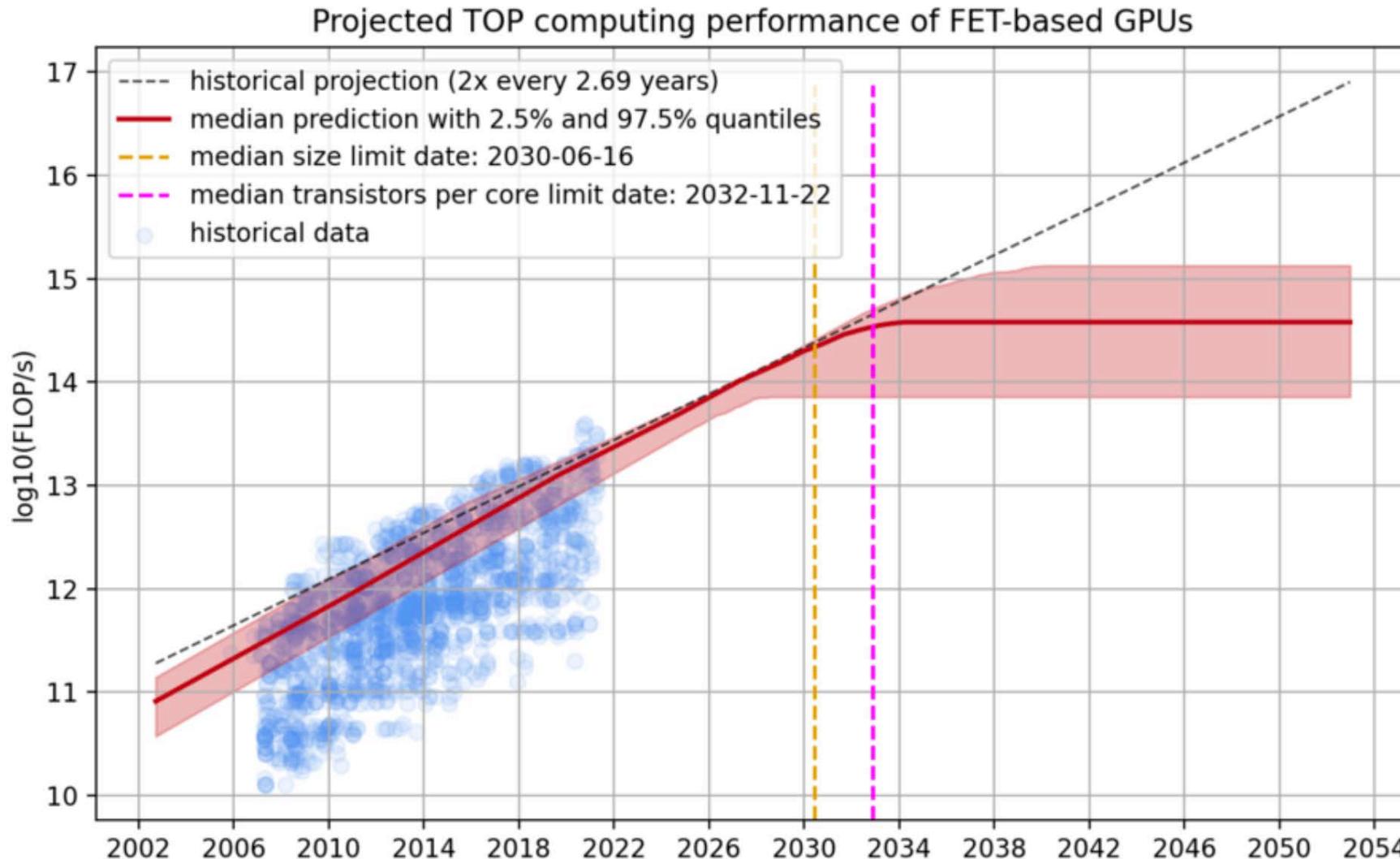
# 近10年模型的变化与硬件的发展

DNN模型的大小，几乎在以每年10倍的FLOPs在增长



## 近10年模型的变化与硬件的发展

相反，硬件的性能却以仅每年0.74倍FLOP/s的速度增长



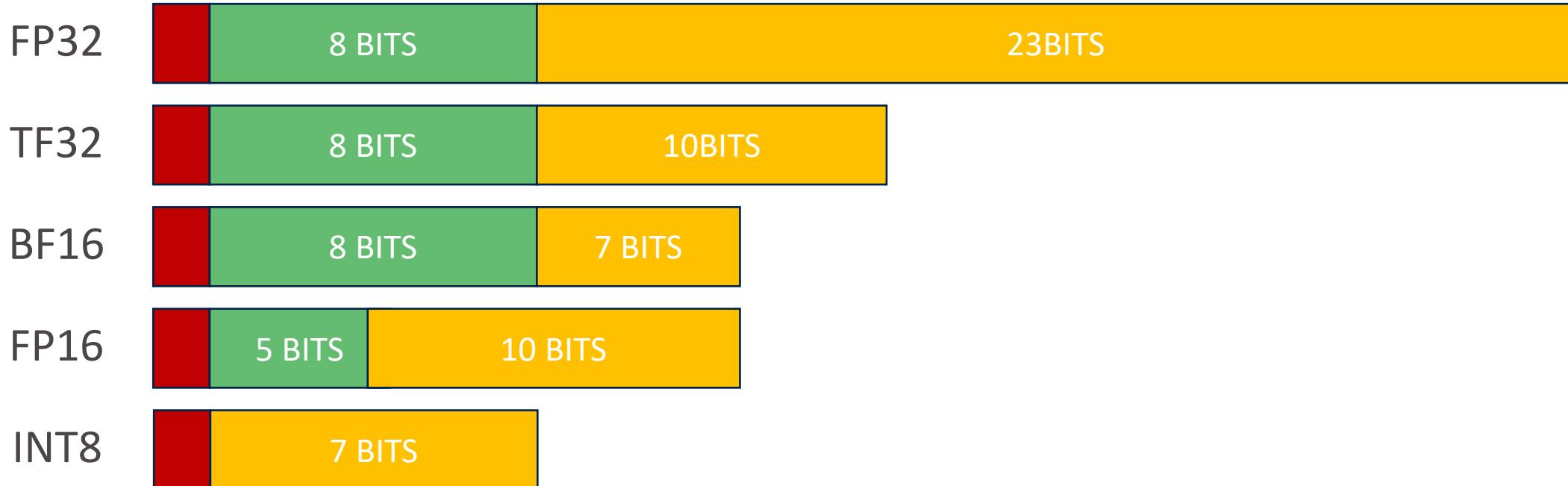
## More background...

相比于模型的发展，硬件的发展速度很慢。即便硬件有了，还需要有相对应的的编译器。有了基本的编译器后，还需要有编译器的优化(TensorRT 3.x~8.x的演变)，还需要有一套其他的SDK。。。

Vision Transformer虽然在2020年左右兴起了，但是真正在硬件上达到如同现在的CNN的优化效果，可能需要8~10年时间。

所以，大家一般会考虑如果用现有的硬件基础上减少模型计算量、增大模型计算密度等等。所以针对这些需求，就有了“量化(quantization)”，“剪枝(Prunning)”等这些优化方法。

## 模型量化回顾



存储FP32需要4个字节，但存储INT8只需要一个字节。因为训练的时候，我们想训练到**非常细节**的部分，所以我们使用FP32。但是如果我们在部署的时候只用8个bit就把FP32的数据给表现出来，那么计算量以及能源消耗不就更好？**模型量化**就是做这个的

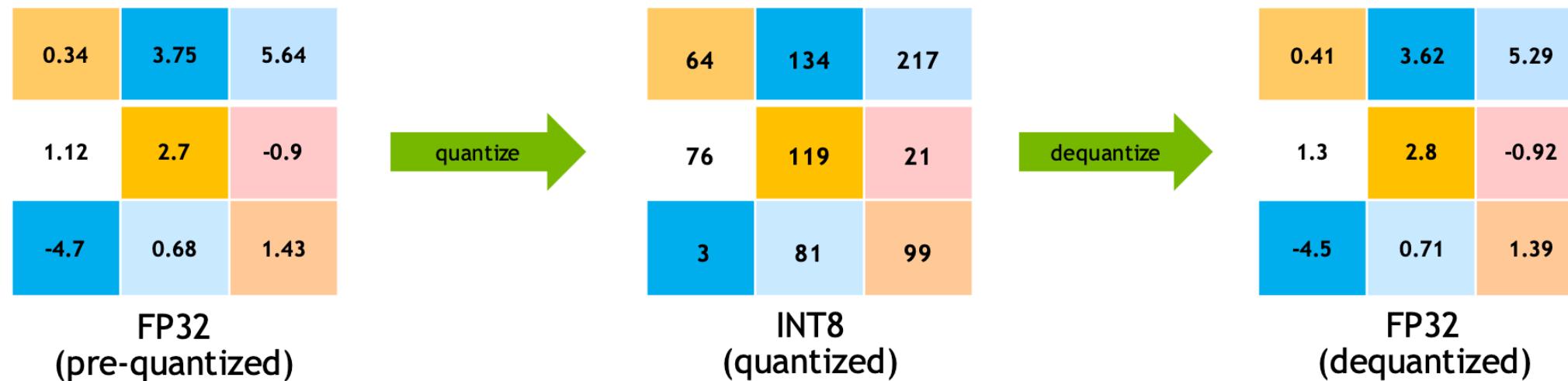
# 什么是量化

模型量化是通过减少模型中计算精度从而减少模型计算所需要的访存量，进而进一步提高计算密度的一种方法。计算精度可以分为FP32, FP16, FP8, INT8, INT32, TF32这些

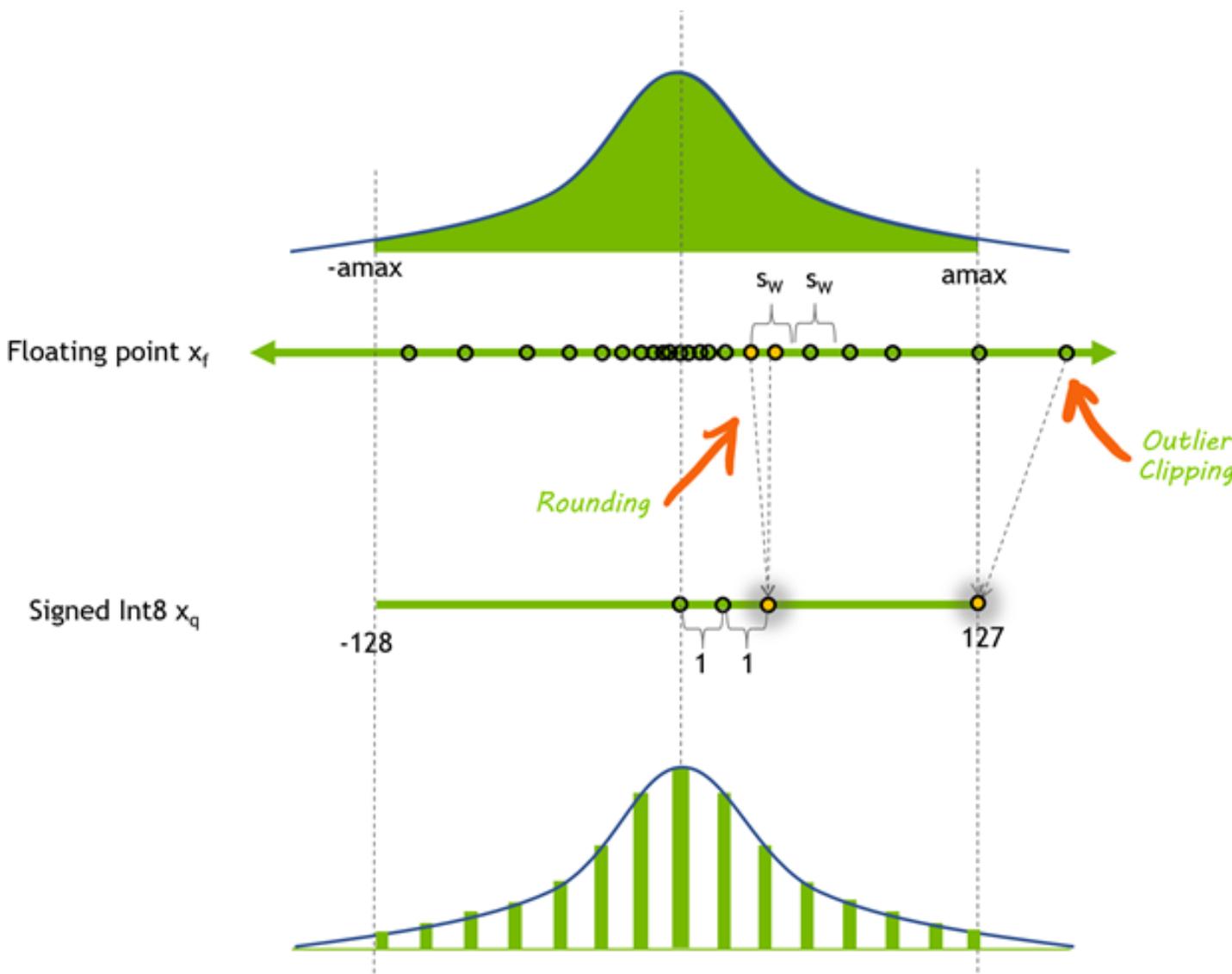
量化针对的是

- activation value
- weight

所以一般来说我们会对conv或者linear这些计算密集型算子进行量化



# 量化会出现什么问题



	Dynamic range
FP32	$-3.4 * 10^{38} \sim 3.4 * 10^{38}$
FP16	$-65504 \sim 65504$
INT8	$-128 \sim 127$

仅仅用256种数据去表现FP32的所有可能出现的数据，有可能会造成**表现力下降**。如果能够比较完美的用这256个数据去最大限度的表现FP32的原始数据分布，是量化的一个很大挑战。

换句话说，就是如何合理的设计这个**dynamic range**是量化的重点

## 量化的基本原理: 映射和偏移

e.g.

$$R: \{x | x \in [-100, 0], x \in Z\}$$

$$Q: \{y | y \in [0, 20], y \in Z\}$$

倘若想把R中的数据用Q来表示, 如何做? (思考一分钟)

## 量化的基本原理: 映射和偏移

e.g.

$$R: \{x | x \in [-100, 0] \text{ } x \in Z\}$$

$$Q: \{y | y \in [0, 20], y \in Z\}$$

倘若想把R中的数据用Q来表示，如何做？

**方法一：**

根据R和Q中x和y可以取的最大值和最小值，计算得到一个缩放比(ratio)；

$$ratio = \frac{(R_{max} - R_{min})}{(Q_{max} - Q_{min})}$$

以及缩放后的A要在B的范围内显示，所需要的偏移量(distance)

$$distance = Q_{min} - \frac{R_{min}}{ratio}$$

最终，通过ratio和distance的到x和y的关系

$$y = \frac{x}{ratio} + distance$$

## 量化的基本原理: 映射和偏移

e.g.

$$R: \{x | x \in [-100, 0] \text{ } x \in Z\}$$

$$Q: \{y | y \in [0, 20], y \in Z\}$$

倘若想把R中的数据用Q来表示，如何做？

方法一：

$$ratio = \frac{(R_{max} - R_{min})}{(Q_{max} - Q_{min})} = \frac{100}{20} = 5$$

$$distance = 0 - \frac{-100}{5} = 20$$

通过ratio和distance我们可以这么理解：

Q中每一个元素可以代表R中每5个元素，并且偏移量是20。

# 量化的基本原理: 映射和偏移

e.g.

$$R: \{x | x \in [-100, 0] \text{ } x \in Z\}$$

$$Q: \{y | y \in [0, 20], y \in Z\}$$

倘若想把R中的数据用Q来表示，如何做？

方法一：(问题所在)

如果说可以通过下面的公式将R中的数据映射到Q中的话，

$$y = \frac{x}{ratio} + distance$$

那么我们按照下面的公式反着计算的话，是不是就可以通过Q中的数据得到R呢？

$$x = (y - distance) * ratio$$

$$y = 0, x = -100$$

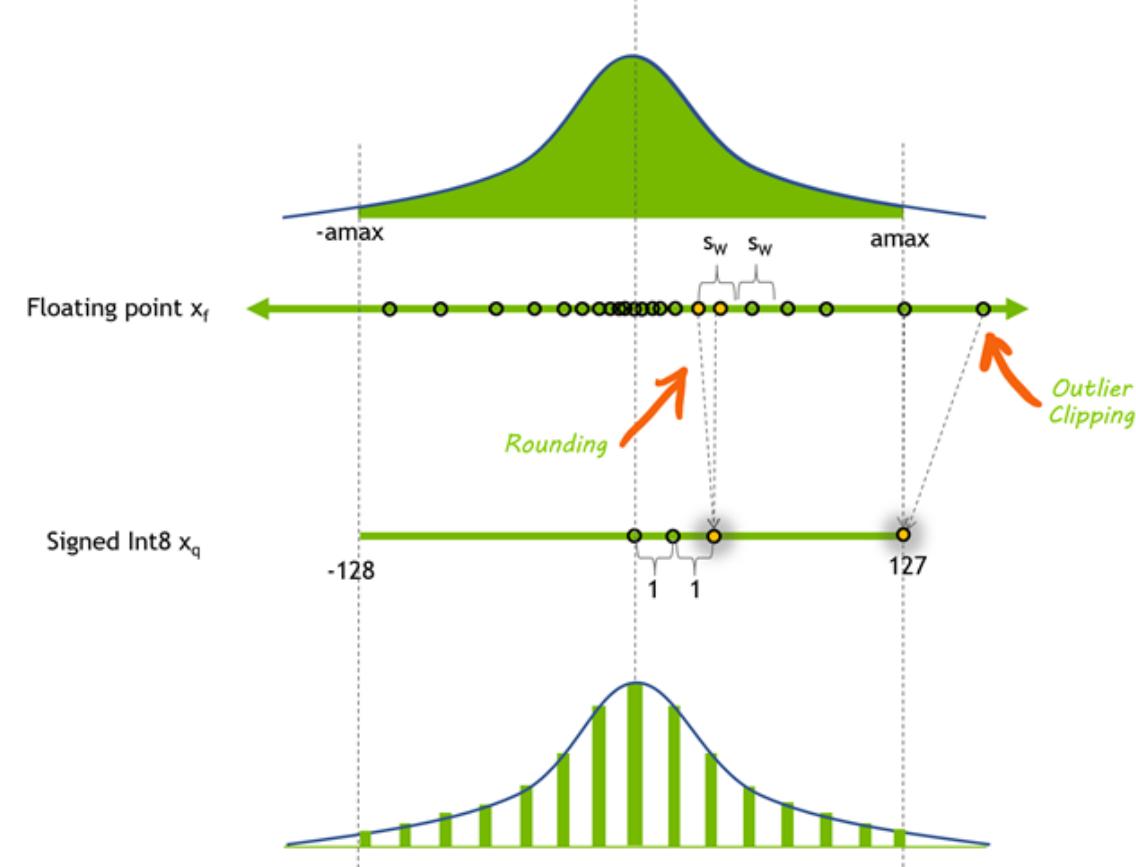
$$y = 1, x = -95$$

$$y = 2, x = -90$$

$$y = 3, x = -85$$

$$y = 4, x = -80$$

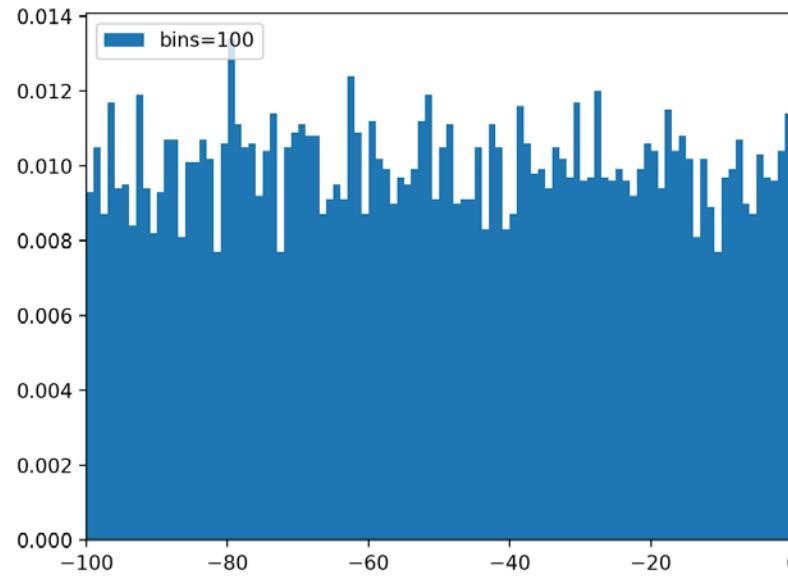
...



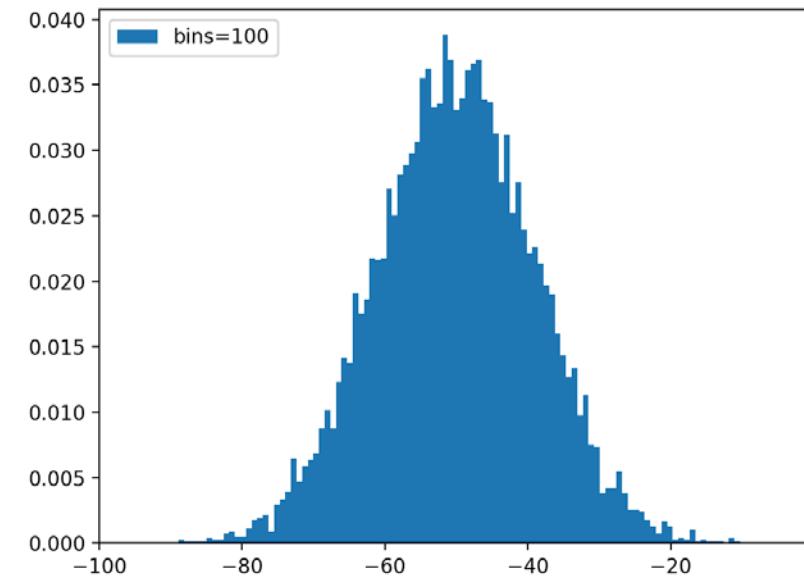
相比于原本的101个R中的数据，如今我们只能够得到R中21个数据。  
比如说-96, -93, -81是无法得到的  
(思考五分钟)这种做法是否okay? 如果不可以，那么问题出现在哪里? 以及如何解决?

# 量化的基本原理: 映射和偏移

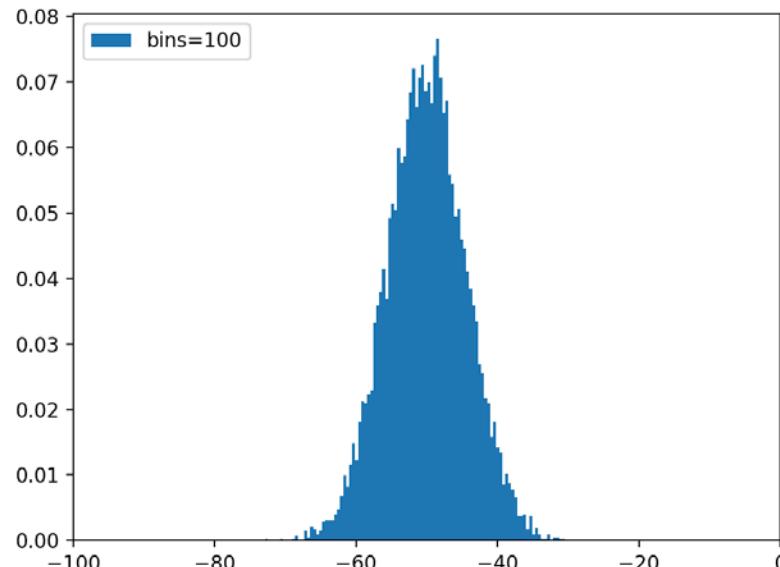
$$R: \{x | x \in [-100, 0] \wedge x \in Z\}$$



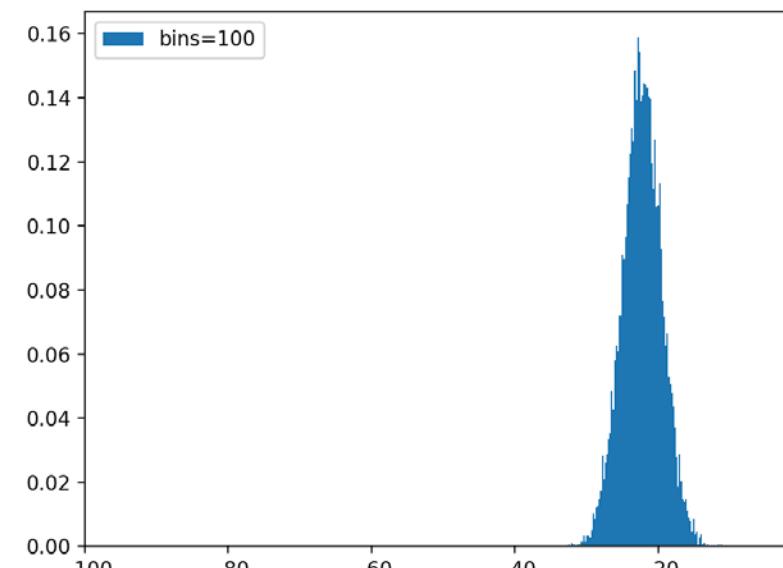
*example1:*  
R中的数据分布不均匀,  
随机出现在-100到0中



*example2:*  
R中的数据呈现高斯分布,  
主要集中在-80~-20中



*example3:*  
R中的数据呈现高斯分布,  
主要集中在-65~-45中



*example4:*  
R中的数据呈现高斯分布,  
主要集中在-30~-15中

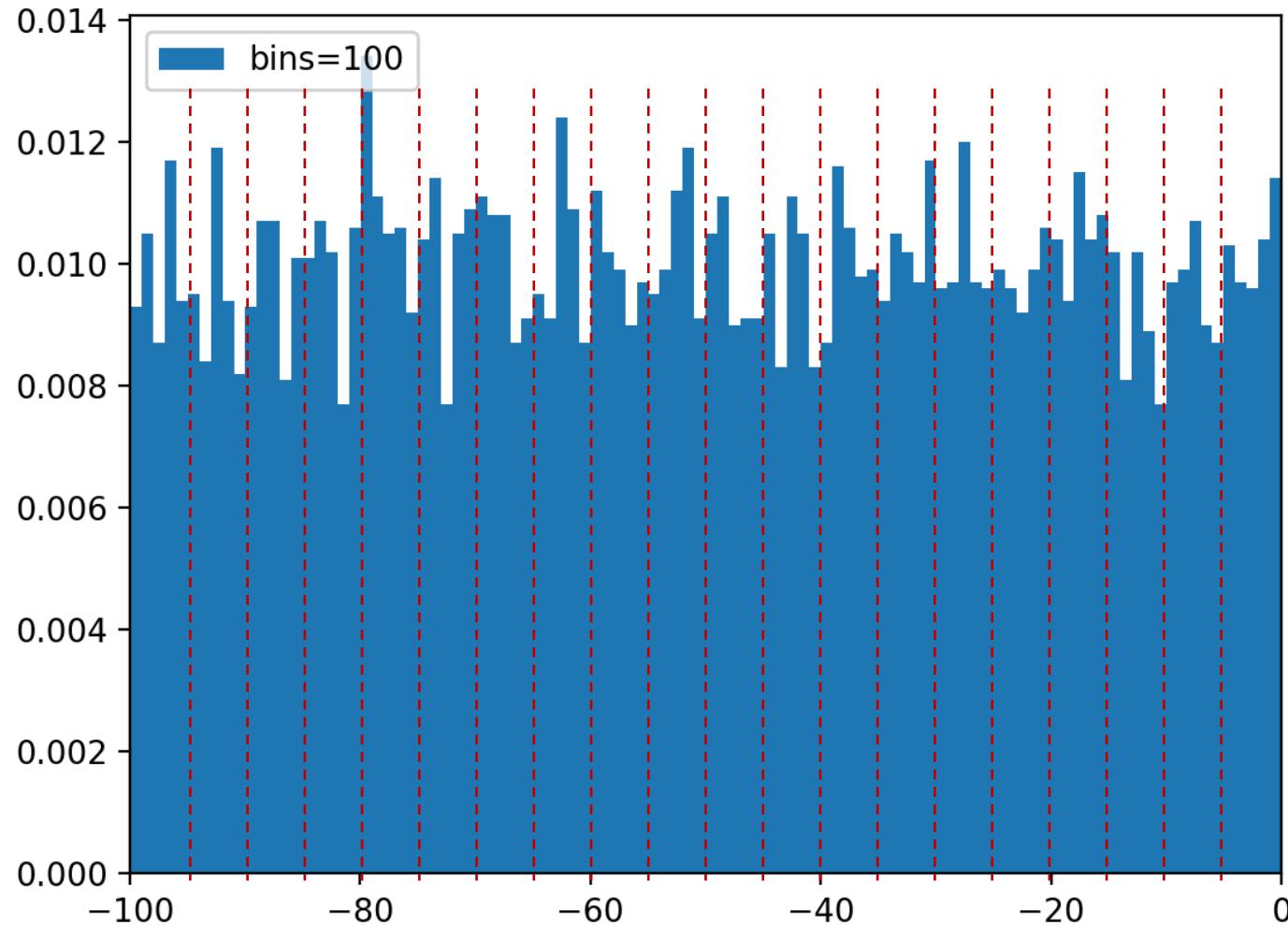
## 量化的基本原理: 映射和偏移

很明显，虽然上述的4个example中数据都呈现-100~0中，但是由于数据的分布形式不同，如果我们统一都用一种ratio和distance的话，会有很大的误差

## 量化的基本原理: 映射和偏移

$$R: \{x | x \in [-100, 0], x \in Z\}$$

$$Q: \{y | y \in [0, 20], y \in Z\}$$



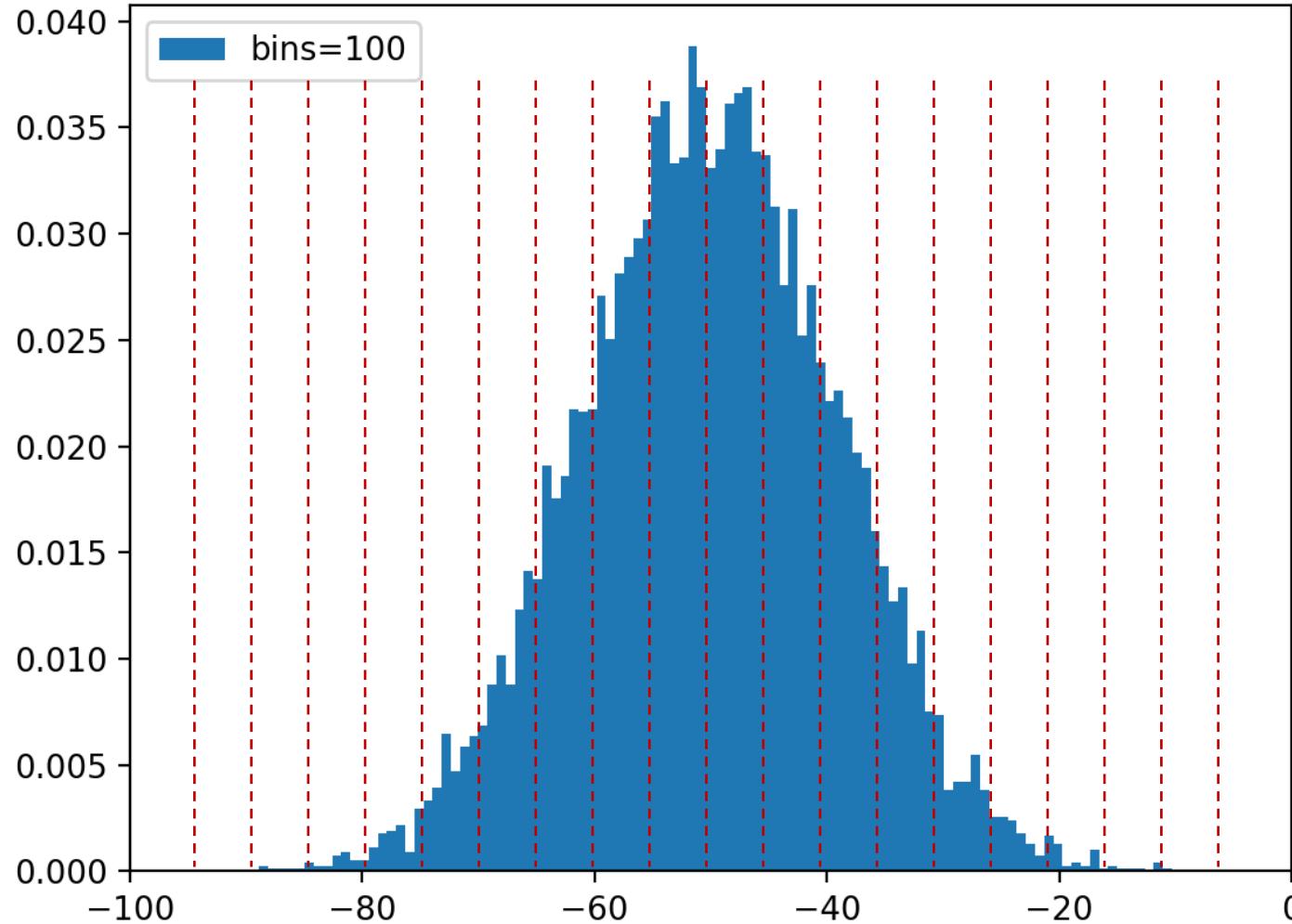
*example1:*  
R中的数据分布不均匀，随机出现在-100到0中。

这个时候因为数据分布不均匀，所以将-100~0均分20个部分映射到Q中是可以接受的，误差也会比较小

# 量化的基本原理: 映射和偏移

$$R: \{x | x \in [-100, 0], x \in Z\}$$

$$Q: \{y | y \in [0, 20], y \in Z\}$$



*example2:*

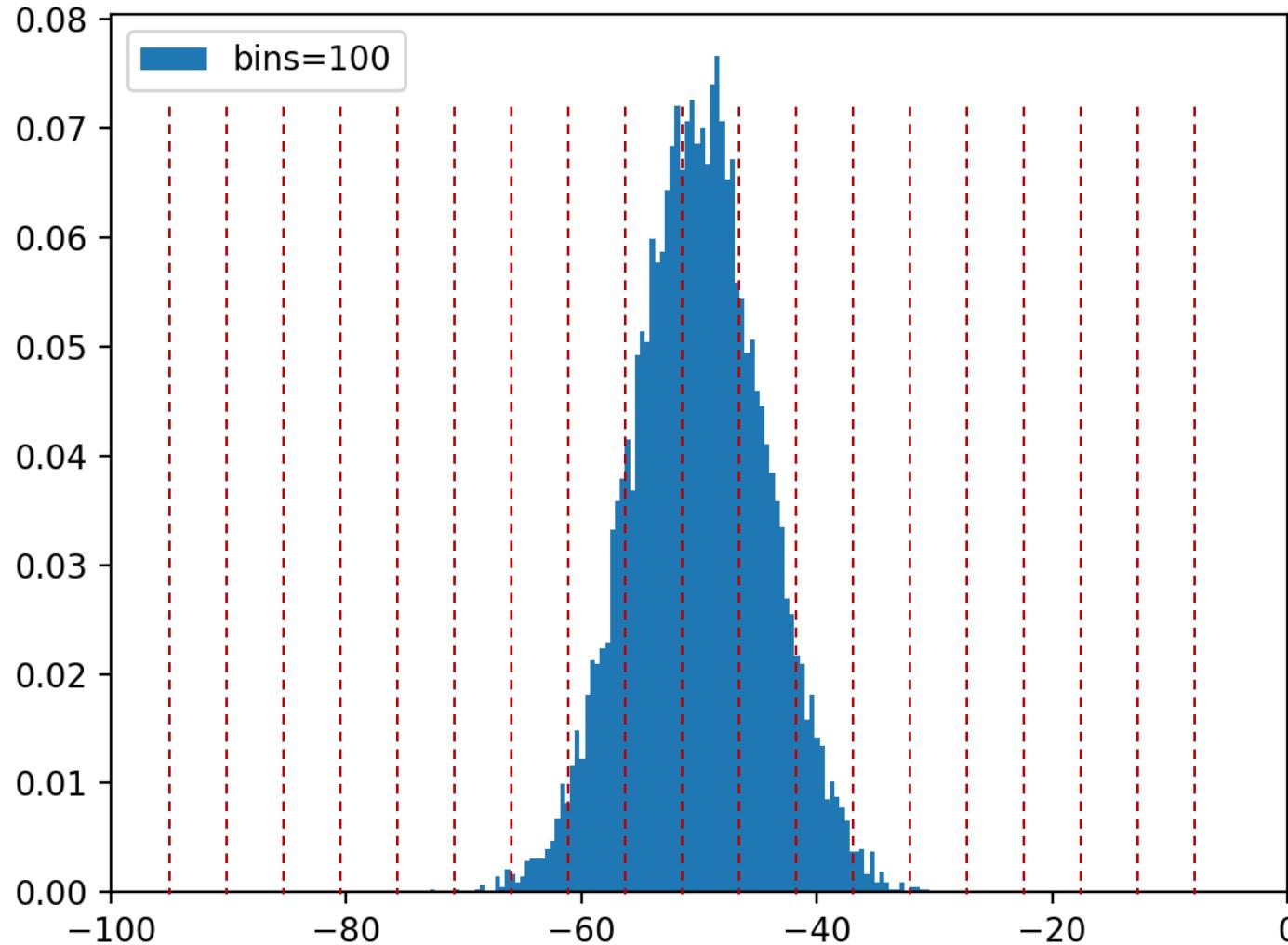
R中的数据呈现高斯分布，主要集中在-80~-20中

这个时候R中的数据分布是属于正态分布，所以数据主要集中在靠拢中间的部分，靠近边缘的数据出现的概率比较低。如果依然等分为20个部分的话，靠近中间的数据会出现较大的误差

# 量化的基本原理: 映射和偏移

$$R: \{x | x \in [-100, 0], x \in Z\}$$

$$Q: \{y | y \in [0, 20], y \in Z\}$$



*example3:*

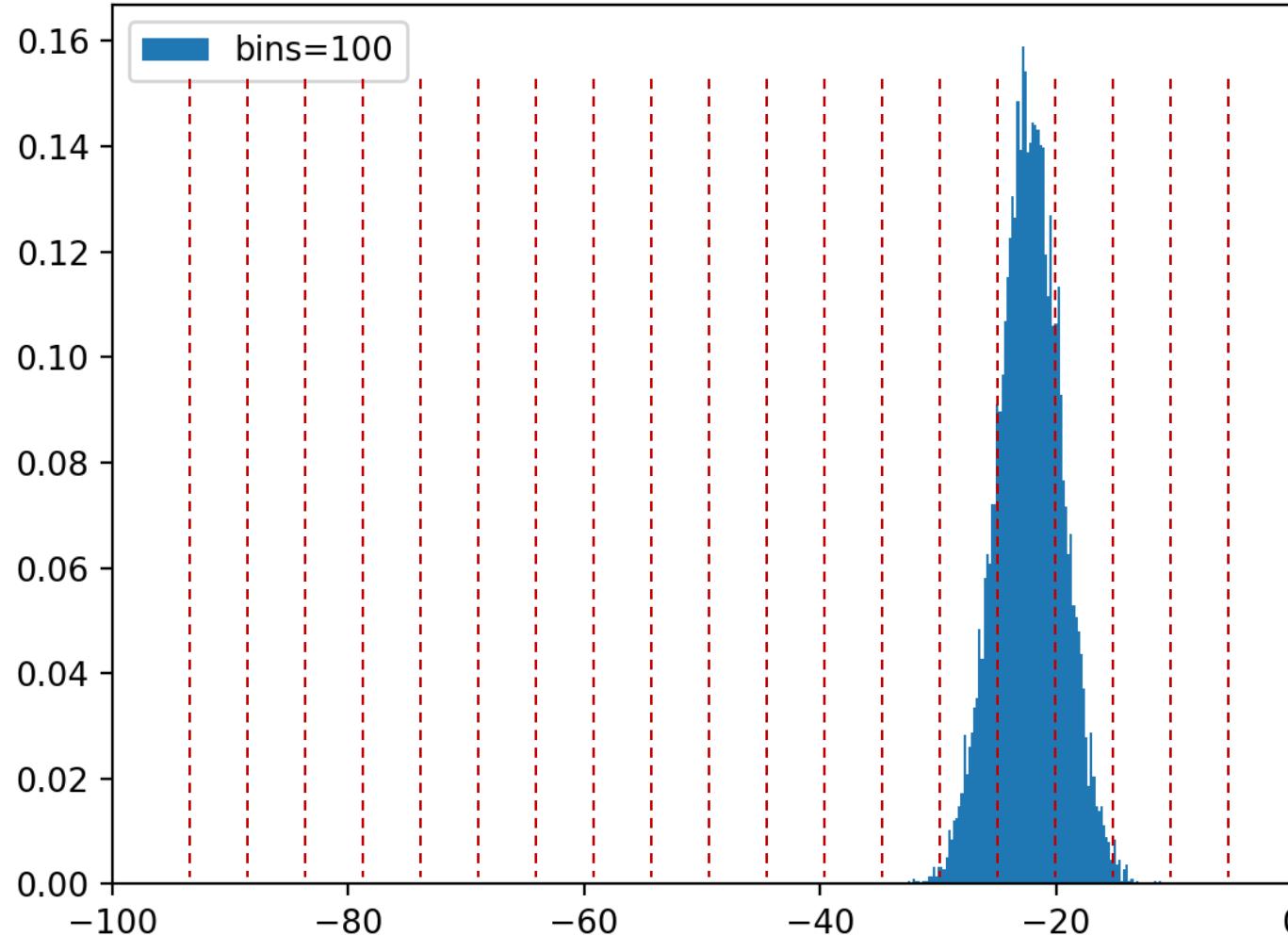
$R$ 中的数据呈现高斯分布，主要集中在-65~-45中

这个时候 $R$ 中的数据分布是属于正态分布，所以数据主要集中在靠拢中间的部分。但是由于方差比较大，所以接近两侧的数据几乎没有。如果按照左图分配的话， $Q$ 中0~5, 15~20所代表的数字没有意义。

## 量化的基本原理: 映射和偏移

$$R: \{x | x \in [-100, 0], x \in Z\}$$

$$Q: \{y | y \in [0, 20], y \in Z\}$$



*example4:*

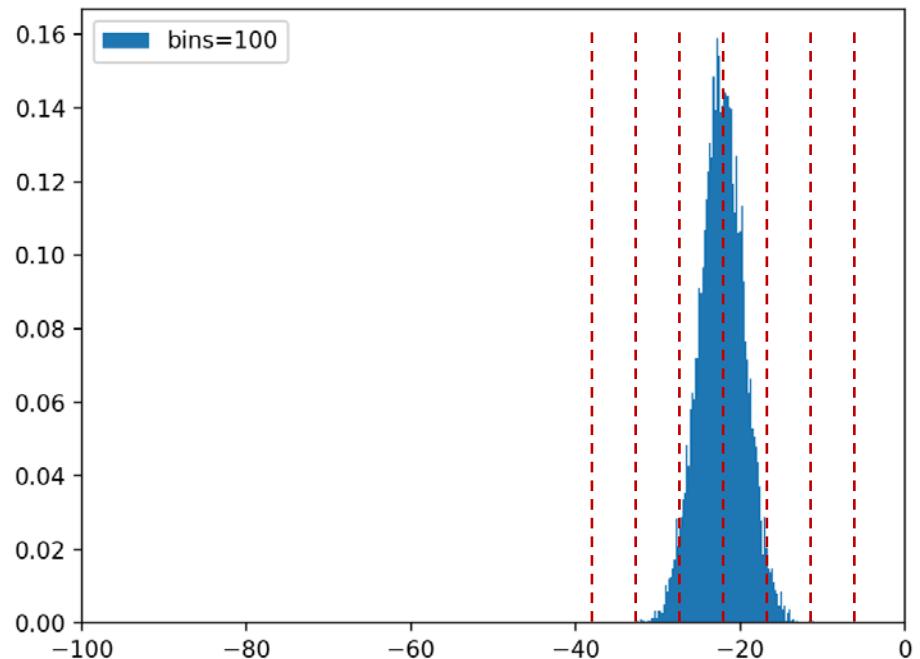
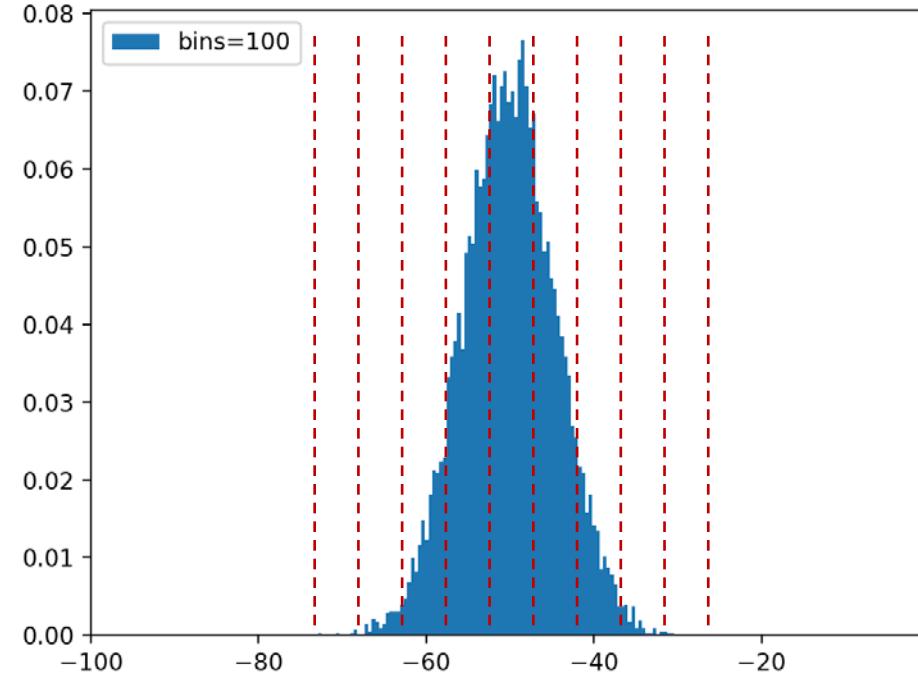
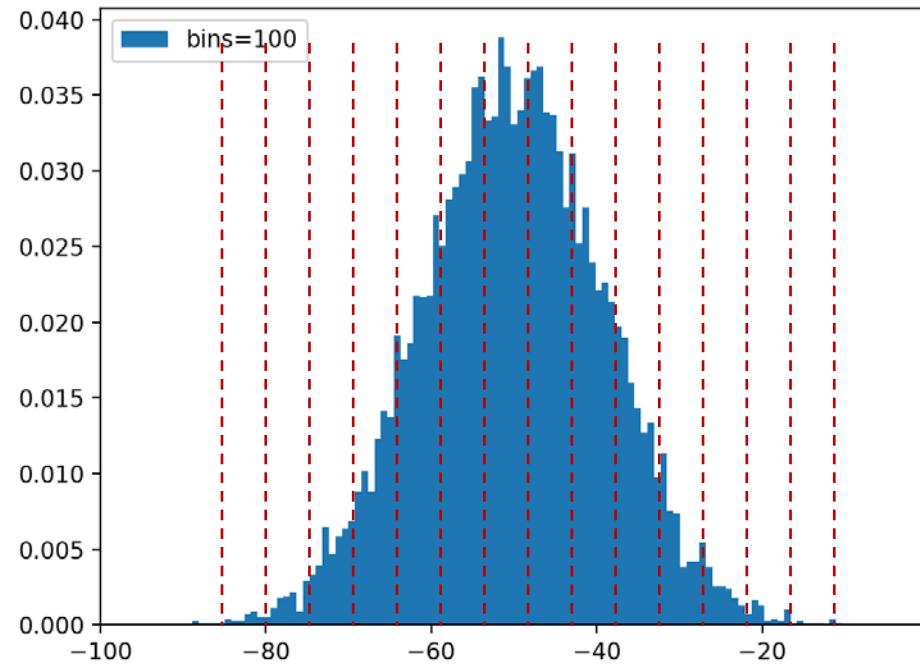
$R$ 中的数据呈现高斯分布，主要集中在-30~-15中

这个时候 $R$ 中的数据分布是属于正态分布，所以数据主要集中在靠拢中间的部分。但是由于方差比较大，以及均值有所偏移，所以 $R$ 中-100~-30这段区域几乎没有。将这段区域映射到 $Q$ 是没有意义的

## 量化的基本原理: 映射和偏移

所以，为了能够让R到Q的映射合理，以及将Q中的数据还原为R时误差能够控制到最小，我们需要根据**R中的数据分布**合理的设计这个ratio和distance。

# 量化的基本原理: 映射和偏移



## 量化的基本原理: 基本术语

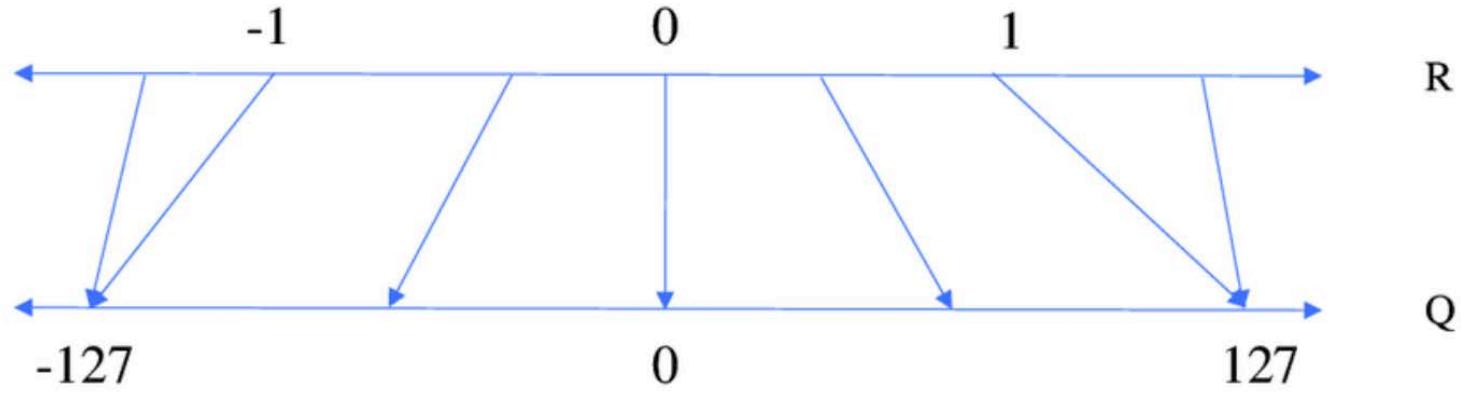
目前为止我们所做的事情其实就是所谓的**量化**，我们稍微把一些概念和术语给整理一下

- R是一组FP32的数据，能够表现的数据种类有很多，大约是 $2^{32}$ 种(4亿)，
  - 范围是:  $-1.2 * 10^{-38} \sim 3.4 * 10^{38}$
- Q是一组INT8的数据，只能够表现 $2^8$ 种数据(256)。
  - 范围是:  $-128 \sim 128$  or  $0 \sim 255$
- R到Q的映射的**缩放因子scale**的计算公式为：
  - $$scale = \frac{(R_{max} - R_{min})}{(Q_{max} - Q_{min})}$$
- R缩放之后映射到Q时，所需要的**偏移量z**为：
  - $$z = Q_{min} - \frac{R_{min}}{ratio}$$
- 这样R中每一个元素转移到Q的过程称为量化(Quantization)，公式是
  - $$Q_i = \frac{R_i}{scale} + z$$
- 将Q空间中一个元素转换回R的空间的过程为反量化(Dequantization)，公式是
  - $$R_i = (Q_i - z) * scale$$

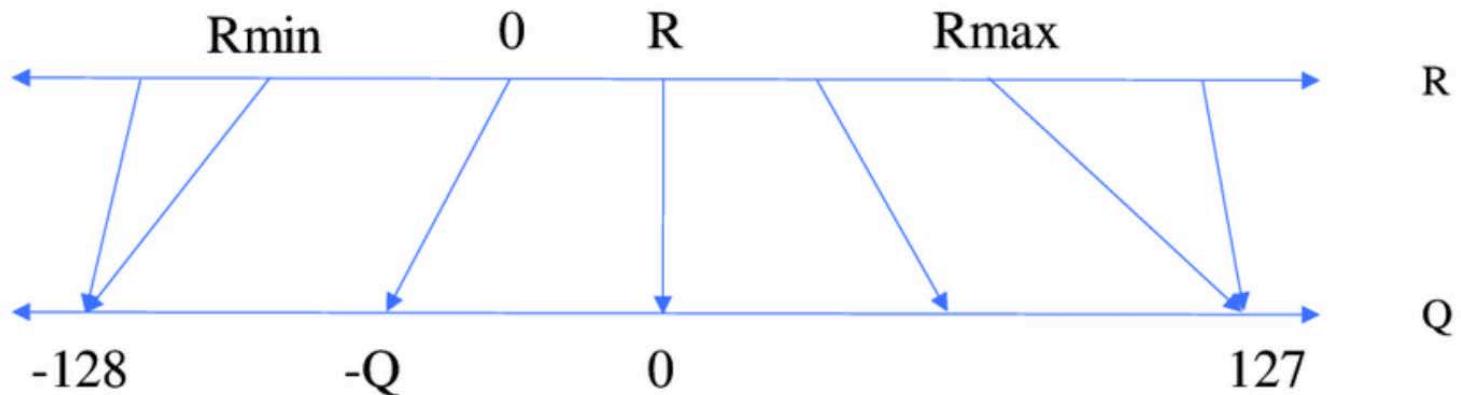
## 量化的基本原理: 基本术语(对称映射, 非对称映射)

(\*)补充: 非对称量化的另外一个叫法是Affine Quantization。对称量化的另外一个叫法是Scale Quantization。这两个称呼是比较早期的。大家在读论文时遇到可以注意一下

Symmetric quantization of weights



Asymmetric quantization of activation functions



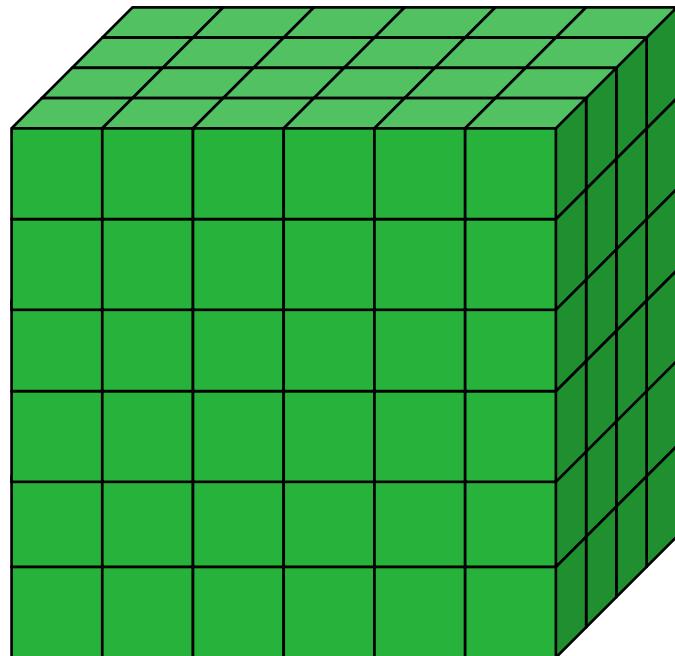
根据R和Q的dynamic range的选择以及mapping的方式，我们可以分为，对称映射(symmetric quantization)以及，非对称映射(asymmetric quantization)<sup>(\*)</sup>。

对称量化中量化前后的0是对齐的，所以不会有偏移量(z, shift)的存在，这个可以让量化过程的计算简单。  
NVIDIA默认的mapping就是对称量化，因为快！

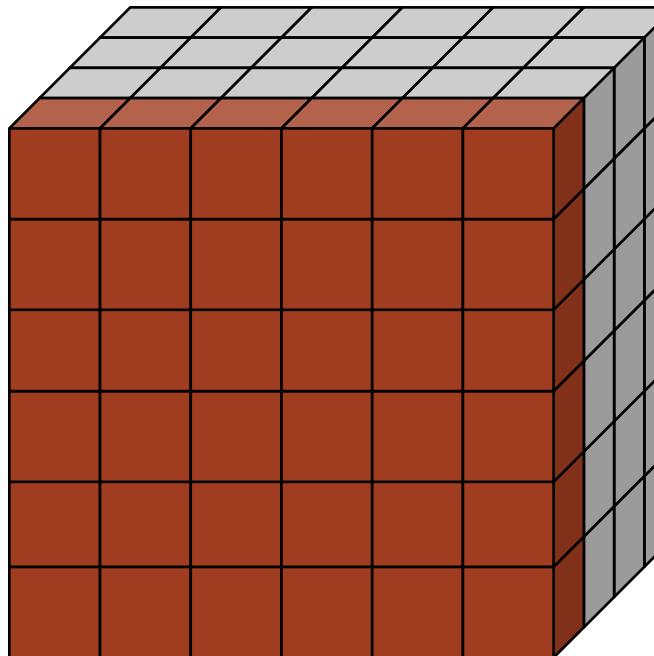
## 量化的基本原理: 基本术语(量化粒度)

量化中非常重要的概念: Quantization Granularity(量化粒度)

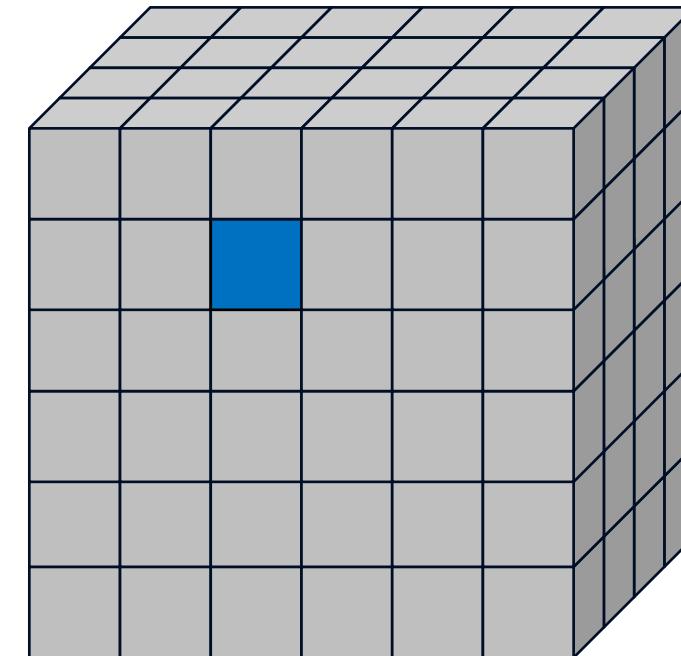
指的是对于一个Tensor, 以多大的粒度去共享scale和z, 或者dynamic range  
具体选哪一个粒度好会很大程度影响性能和精度! (下一个结讲)



per-tensor quantization  
(一个tensor中所有的  
element共享同一个  
dynamic range)



per-channel quantization  
(一个tensor中每一个layer都  
有一个自己的dynamic  
range)



per-element quantization  
(一个tensor中每一个element都  
有一个自己的dynamic range。  
也可以叫做element-wise  
quantization)

## 量化的基本原理: 基本术语(校准)

量化中另外一个非常重要的概念: Calibration(校准)

对于一个训练好的模型, 权重是固定的, 所以可以通过一次计算就可以得到每一层的量化参数。但是activation value(激活值)是根据输入的改变而改变的。所以需要通过类似于统计的方式去寻找对于不同类型的输入的不同的dynamic range。这个过程叫做校准

跟量化粒度一样, 不同的校准算法的选择会很大程度影响精度! (下下一个结讲)

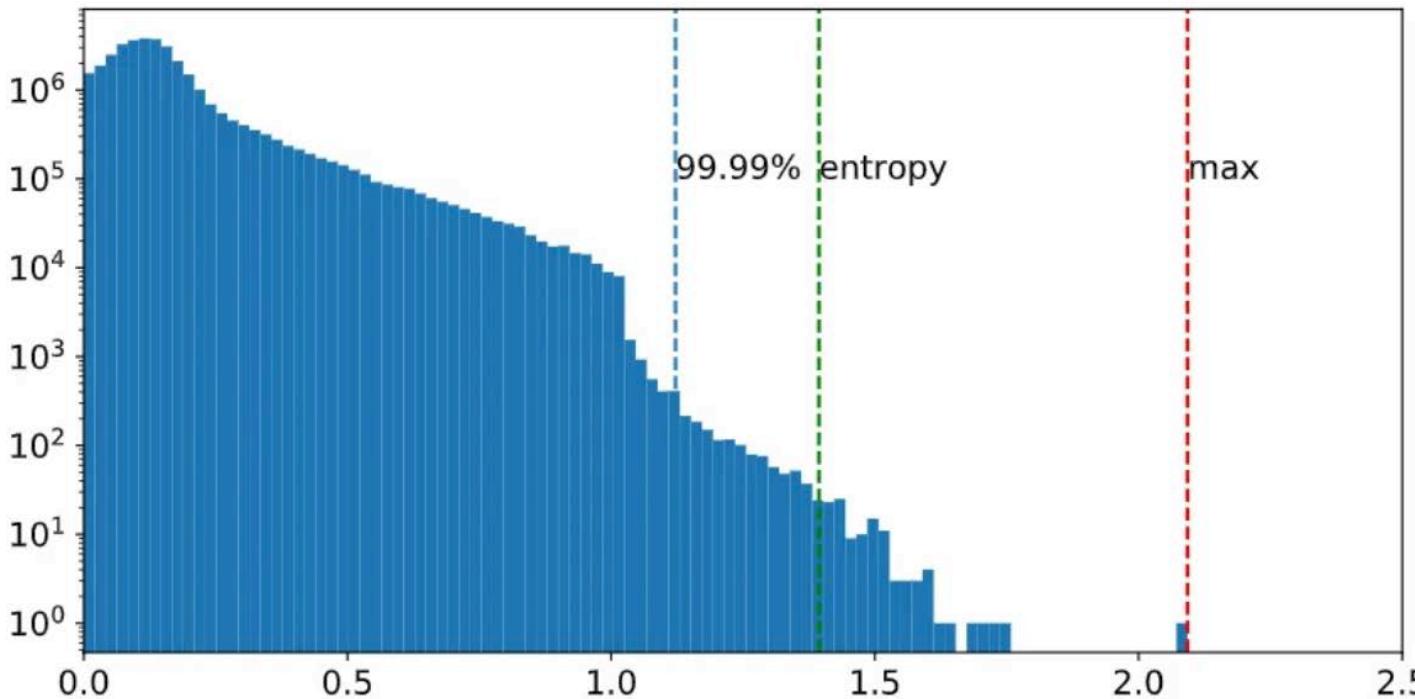


Fig.对于某一个tensor的histogram, 不同的校准算法所截取的范围的不同<sup>[1]</sup>

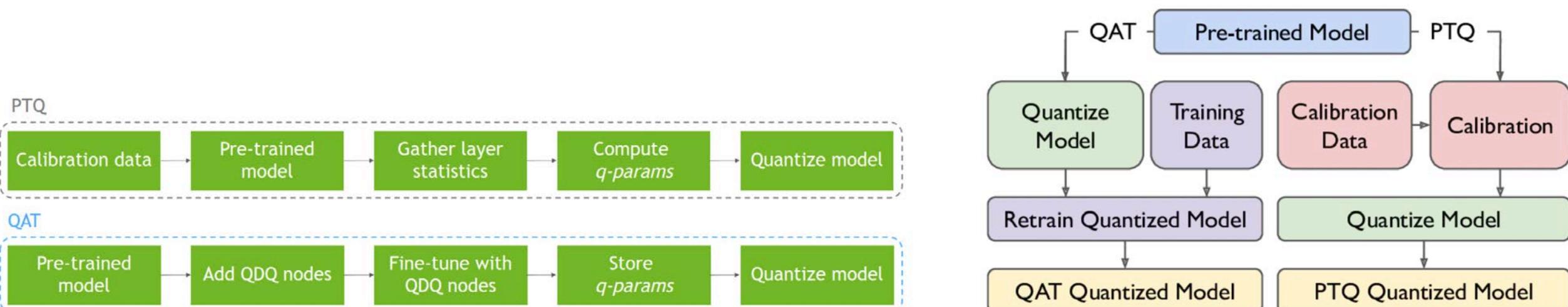
[1]Integer-Only Inference for Deep Learning in Native C

# 量化的基本原理: 基本术语(PTQ, QAT)

根据量化的时机，一般我们会把量化分为

- PTQ(Post-Training Quantization), 训练后量化
- QAT(Quantization-Aware Training), 训练时量化

PTQ一般是指对于训练好的模型，通过calibration算法等来获取dynamic range来进行量化。但量化普遍上会产生精度下降。所以QAT为了弥补精度下降，在学习过程中通过Fine-tuning权重来适应这种误差，实现精度下降的最小化。所以一般来讲，QAT的精度会高于PTQ。但并不绝对。详细在下下下一小节讲。



PTQ与QAT的workflow上的不同[1, 2]

[1] [Accelerating Quantized Networks with the NVIDIA QAT Toolkit for TensorFlow and NVIDIA TensorRT](#)

[2] [Neural Network Quantization for Efficient Inference: A Survey](#)

## 其他：有关量化学习的激活函数

量化学习是一个Fine-tuning的过程。那么选取什么样子的激活函数会更好呢？我们可以结合量化的特性去思考。我们希望整个学习过程让权重或者激活值控制在某个区域范围内，所以我们需要实现某种Clipping。推荐两个激活函数：

- PACT(Paramertized Clipping Activation Function)
- ReLU6

知道ReLU6的人应该比较多，不清楚的可以回顾一下MobileNet的文章。对于PACT的介绍，推荐阅读一下IBM的论文<sup>[1]</sup>，这里不展开讲了

$$y = PACT(x) = 0.5(|x| - |x - \alpha| + \alpha) = \begin{cases} 0, & x \in (-\infty, 0) \\ x, & x \in [0, \alpha) \\ \alpha, & x \in [\alpha, +\infty) \end{cases}$$



03

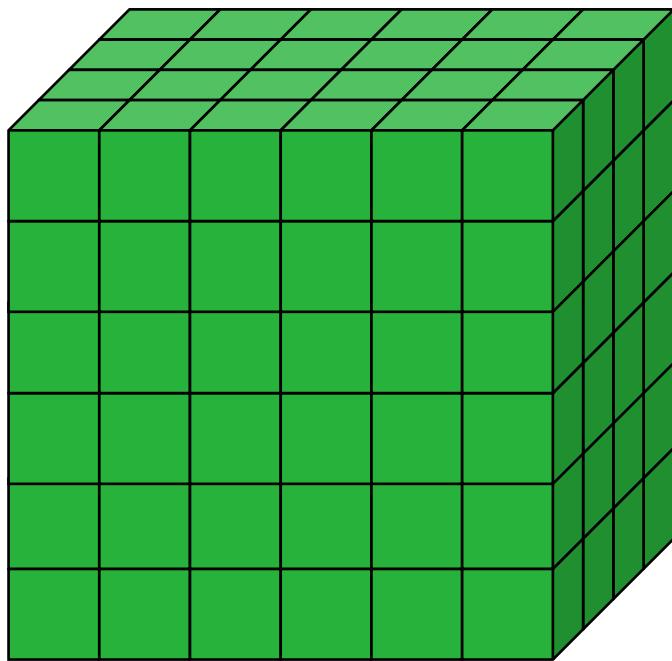
# Quantization (quantization granularity)

Goal: 理解量化粒度是什么，如何正确的选择量化粒度，以及量化粒度与精度/计算量/计算效率的关系

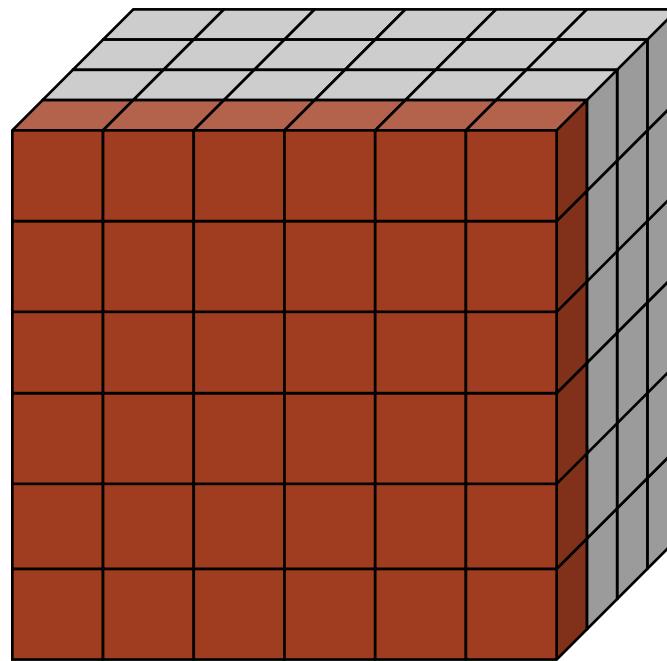
## 量化的基本原理: 基本术语(量化粒度)

量化中非常重要的概念: Quantization Granularity(量化粒度)

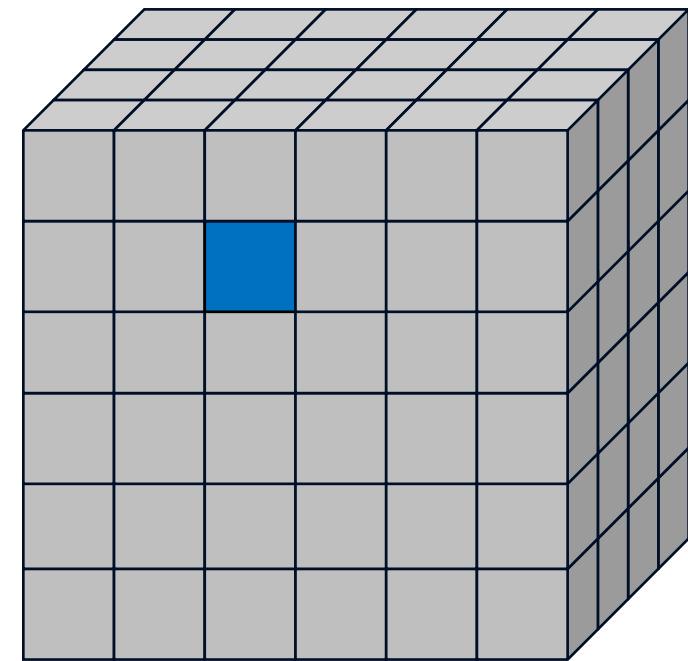
指的是对于一个Tensor, 以多大的粒度去共享scale和z, 或者dynamic range



per-tensor quantization  
(一个tensor中所有的  
element共享同一个  
dynamic range)

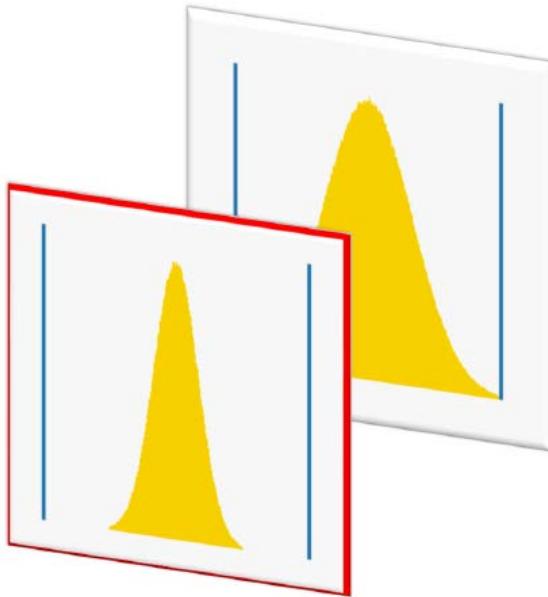


per-channel quantization  
(一个tensor中每一个layer  
都有一个自己的dynamic  
range)



per-element quantization  
(一个tensor中每一个element都  
有一个自己的dynamic range。  
也可以叫做element-wise  
quantization)

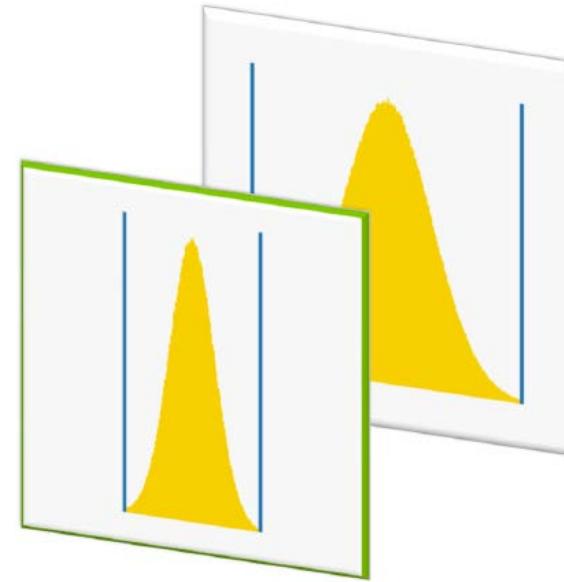
# Per-tensor & Per-channel量化



## Per-tensor量化

- (优点)低延迟: 一个tensor共享同一个量化参数
- (缺点)高错误率: 一个scale很难覆盖所有FP32的dynamic range

(思考一下)什么情况该用Per-tensor, 什么情况该用Per-channel?



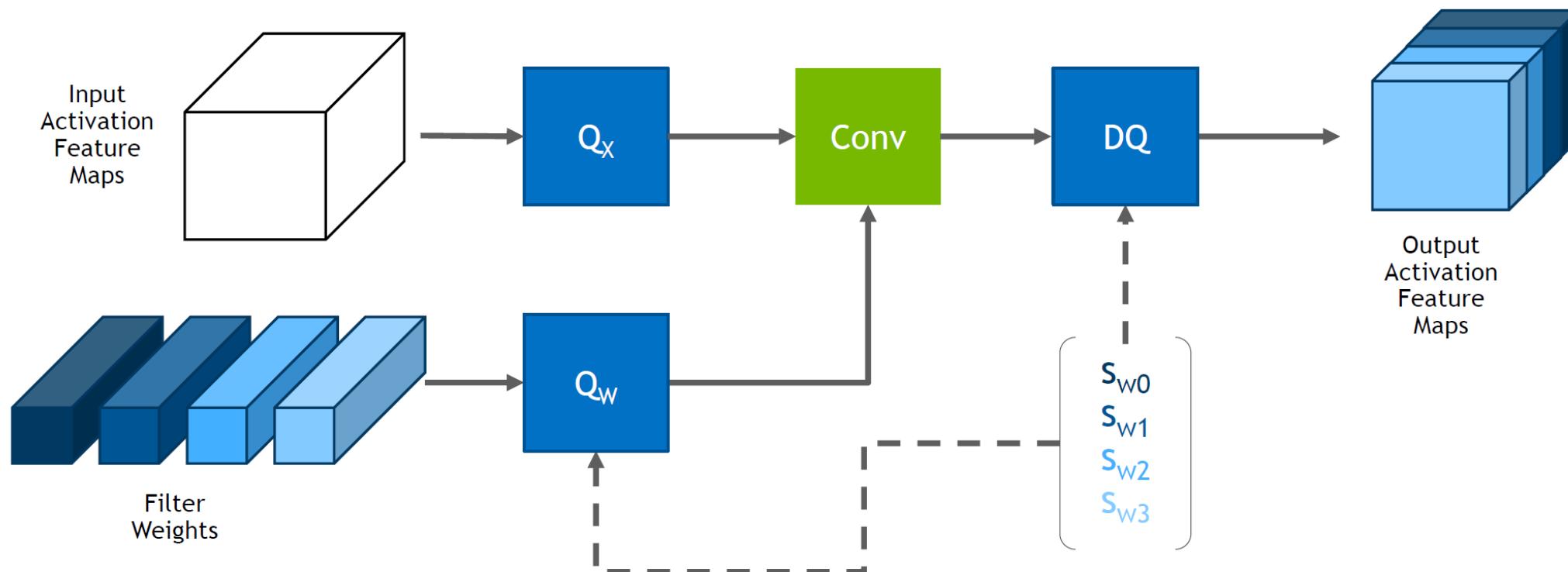
## Per-channel (layer)量化

- (优点)低错误率: 每一个channel都有自己的scale来体现这个channel中的数据的dynamic range
- (缺点)高延迟: 需要使用vector来存储每一个channel的scale

# 量化粒度选择的推荐方法

(重点)从很多实验结果与测试中，对于weight和activation values的量化方法，一般会选取

- 对于activation values，选取per-tensor量化
- 对于weights，选取per-channel量化

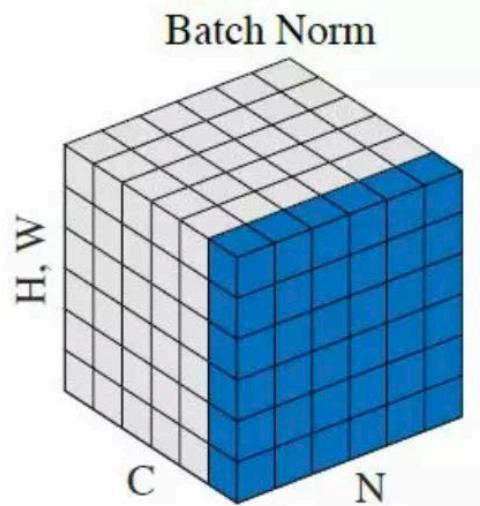


# 量化粒度选择的推荐方法

为什么weight需要per-channel呢？主要是因为

- BN计算与线性计算的融合 (BN folding)
- depthwise convolution

线性变化 $y = w * x$ 的BN folding可以把BN的参数融合在线性计算中。但是BN的可参数是per-channel的。如果weights用per-tensor的话，会掉精度。



$$\mu_B = \frac{1}{B} \sum_{i=1}^B x_i$$

$$\sigma_B^2 = \frac{1}{B} \sum_{i=1}^B (x_i - \mu_B)^2 + \epsilon$$

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$y_i = \gamma * \hat{x}_i + \beta$$

$$y = \gamma * \frac{w * x + b - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} + \beta$$

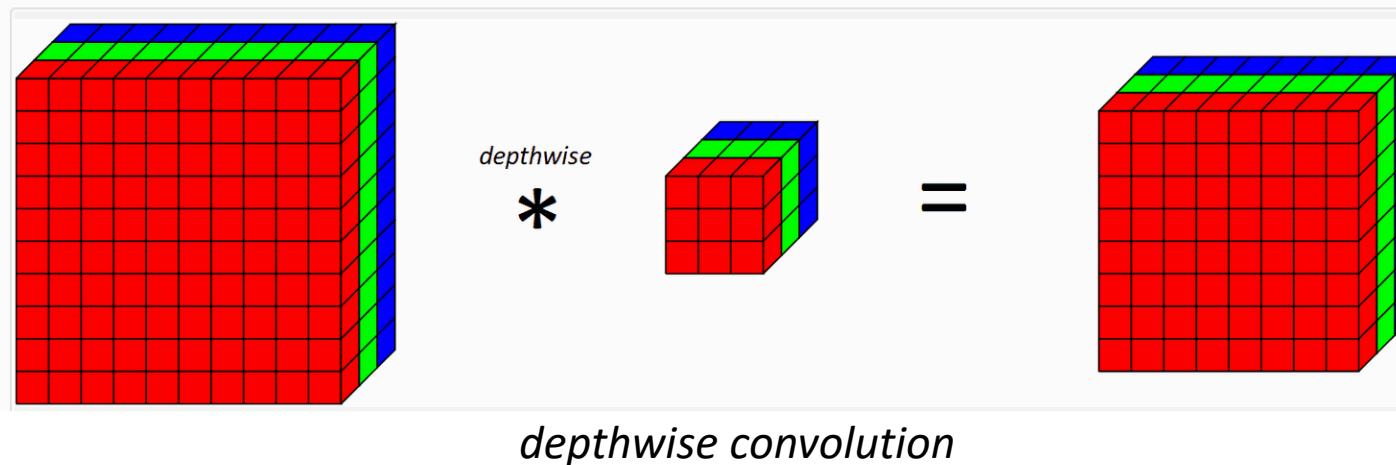
$$y = \frac{\gamma * w}{\sqrt{\sigma_B^2 + \epsilon}} * x + \frac{\gamma}{\sqrt{\sigma_B^2 + \epsilon}} (b - \mu_B) + \beta$$

$$y = \left( \frac{\gamma * w}{\sqrt{\sigma_B^2 + \epsilon}} \right) * x + \left( \frac{\gamma}{\sqrt{\sigma_B^2 + \epsilon}} (b - \mu_B) + \beta \right)$$

# 量化粒度选择的推荐方法

为什么weight需要per-channel呢？主要是因为

- BN计算与线性计算的融合 (BN folding)
- depthwise convolution



depthwise convolution中kernel的channel size是1，每一个kernel针对输入的对应的channel做卷积。所以每一个channel中的参数可能差别会比较大。如果用per-tensor的话容易掉精度比较严重

- e.g.
  - MobileNet: (FP32) 71.88,
  - MobileNet: (int8 Per-channel weight quantization) 71.56
  - MobileNet: (int8 Per-tensor weight quantization) 66.88
- EfficientNet: (FP32) 76.85,
- EfficientNet: (int8 Per-channel weight quantization) 76.72
- EfficientNet: (int8 Per-tensor weight quantization) 12.93

## 量化粒度选择的推荐方法

重点！！

目前的TensorRT已经默认对于Activation values选用Per-tensor, Weights选用Per-channel，这是他们做了多次实验所得出的结果。很多其他平台的SDK可能不会提供一些默认的量化策略，这是我们需要谨慎选择，尽快找到掉点的原因



03

## Quantization (calibration)

Goal: 理解calibration的作用。不同calibration algorithm的算法不同点，以及什么时候使用哪种算法。calibration与batch size的关系

## 量化的基本原理: 基本术语(校准)

量化中另外一个非常重要的概念: Calibration(校准)

对于一个训练好的模型, 权重是固定的, 所以可以通过一次计算就可以得到每一层的量化参数。但是activation value(激活值)是根据输入的改变而改变的。所以需要通过类似于统计的方式去寻找对于不同类型的输入的不同的dynamic range。这个过程叫做校准

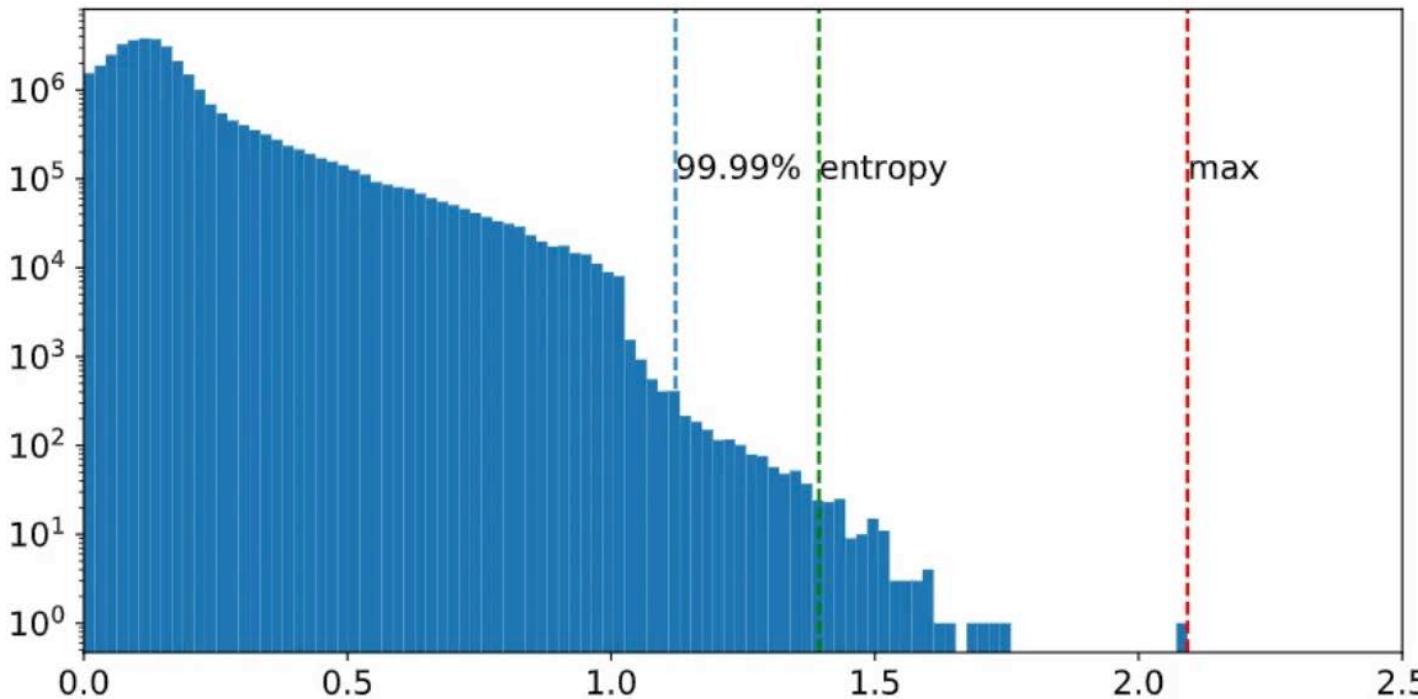


Fig.对于某一个tensor的histogram, 不同的校准算法所截取的范围的不同<sup>[1]</sup>

[1]Integer-Only Inference for Deep Learning in Native C

# Calibration dataset



针对不同的输入，各层layer的input activation value都会有不同的分布和取值。大数据集的差别比较大。我们需要通过训练数据集中的一部分数据来尝试表征整个数据集的分布。

这个小数据集就是**calibration dataset**。一般往往很小，但需要尽量有整体的表征

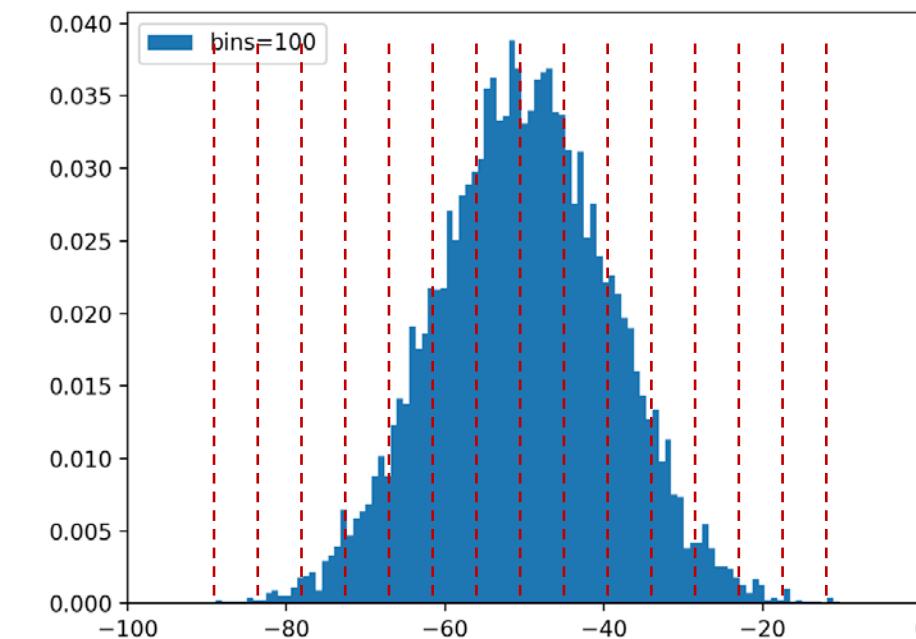
在使用TensorRT时，对于  
**calibration dataset**的选择，以  
及统计时所用的batch size这  
里有个坑，后面讲

# Calibration algorithm

calibration的过程一般是在模型训练以后进行的，所以一般与PTQ<sup>(\*)</sup>搭配使用。整体的流程就是：

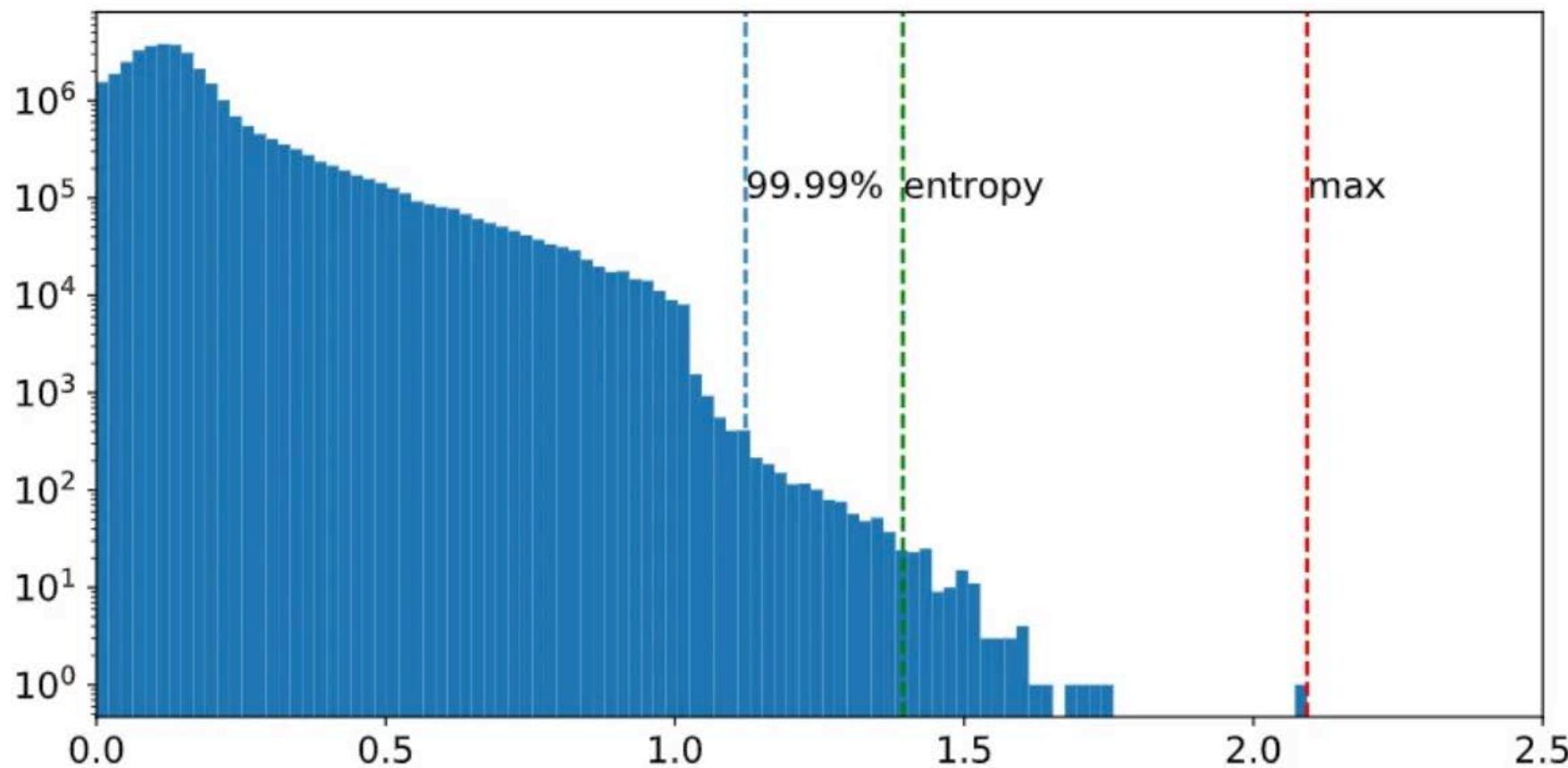
- 在calibration dataset中做一次FP32的推理
- 以**histogram**的形式去统计每一层的floating point的分布
  - (注意，因为activation value是per-tensor quantization)
- 寻找能够表征当前层的floating point分布的scale
  - 这里会有几种不同的算法，比较常见的有
    - Minmax calibration
    - Entropy calibration
    - Percentile calibration

(以上这些过程TensorRT都已经帮我们封装好了，可以拿来直接用)



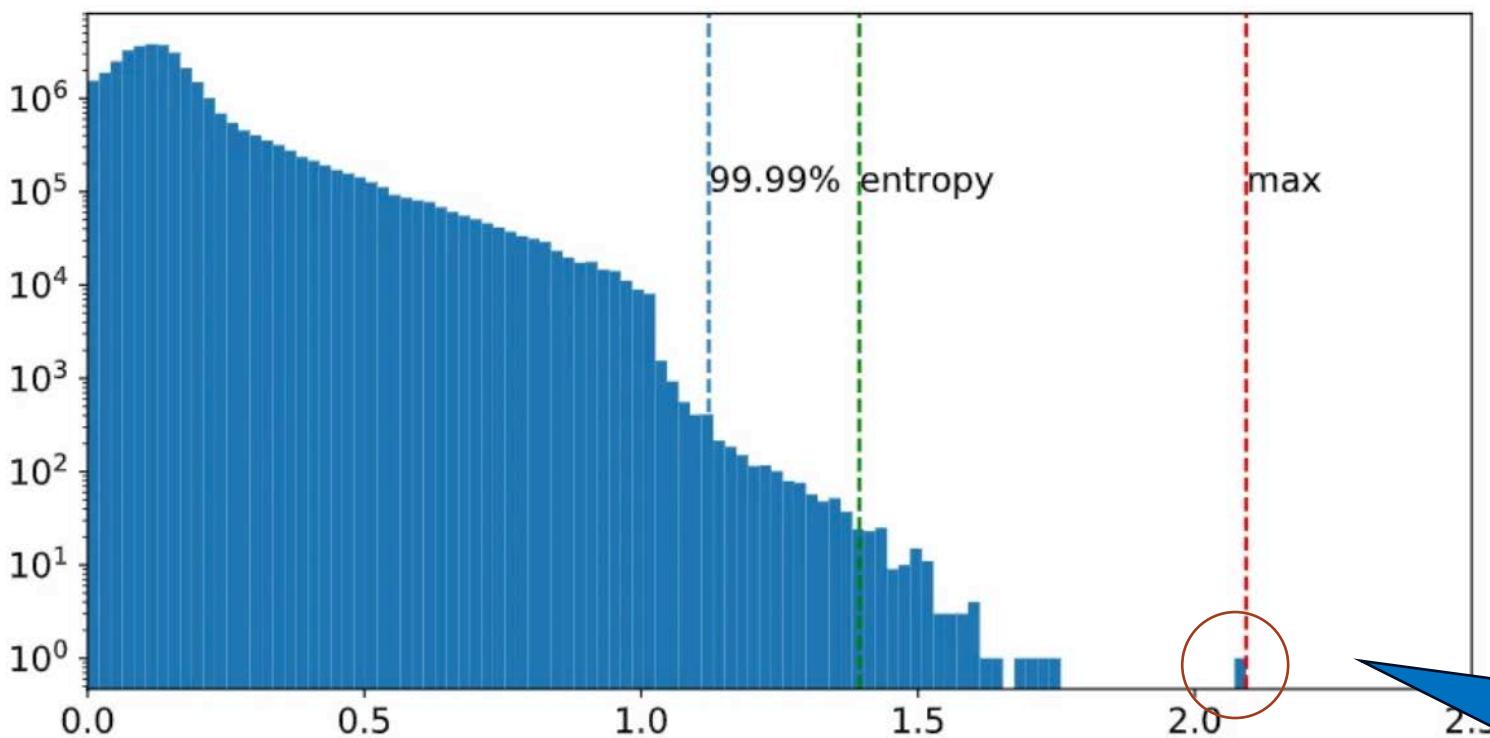
## Calibration algorithm

- 这三种calibration的区别是什么?
  - Minmax calibration
  - Entropy calibration
  - Percentile calibration



# Calibration algorithm

- 这三种calibration的区别是什么?
  - Minmax calibration
  - Entropy calibration
  - Percentile calibration



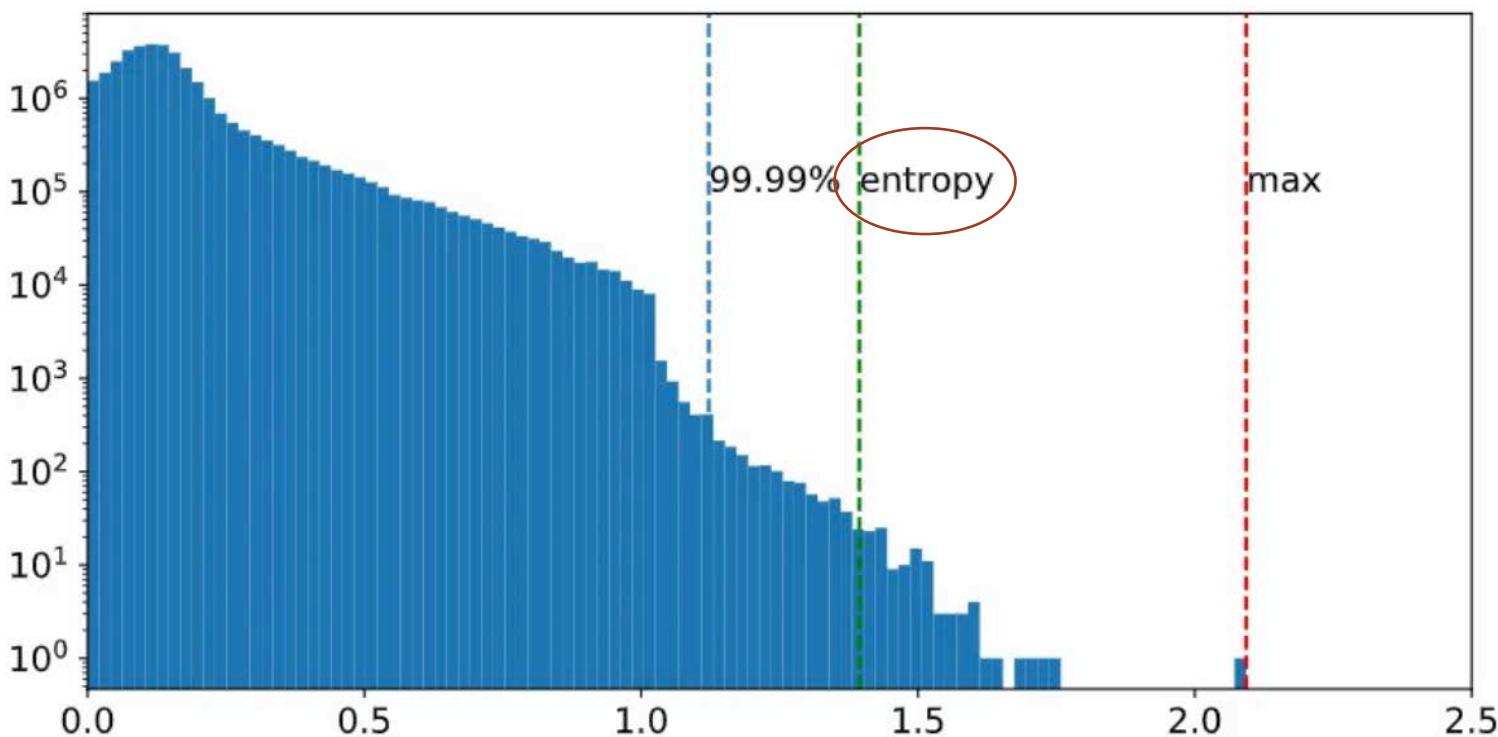
FP32->INT8的scale需要能够把FP32中的最大最小值都给覆盖住。

如果floating point的分布比较离散，各个区间下的分布都比较均匀，minmax是个不错的选择

然而，如果只是极个别数据分布在这种地方的话，会让dynamic range变得比较稀疏，不适合用minmax

# Calibration algorithm

- 这三种calibration的区别是什么?
  - Minmax calibration
  - Entropy calibration
  - Percentile calibration

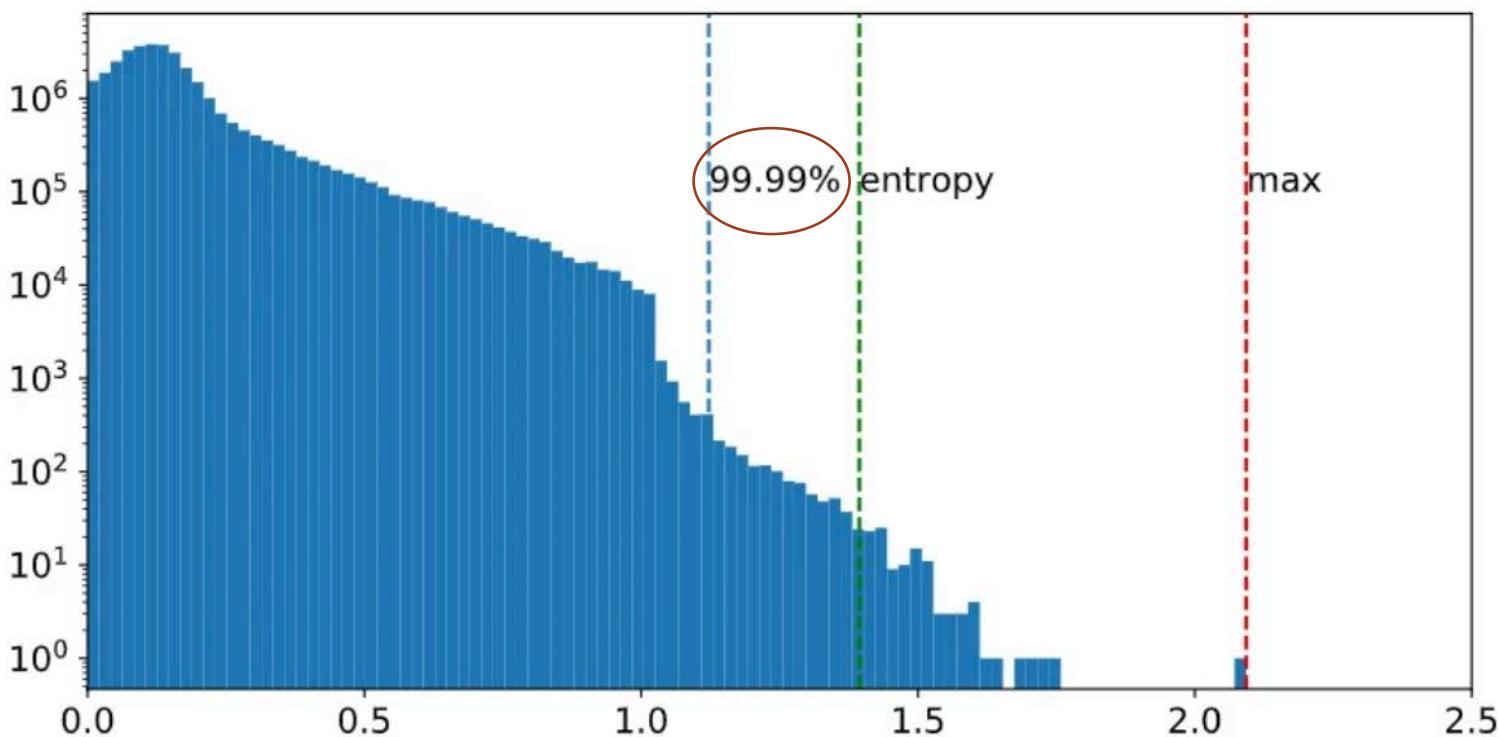


通过计算KL散度，寻找一种 threshold，能够最小化量化前的 FP32 的浮点数分布于 INT8 的量化后整形分布

目前TensorRT使用默认的是Entropy calibration。一般来讲使用entropy calibration精度可以比较好

# Calibration algorithm

- 这三种calibration的区别是什么?
  - Minmax calibration
  - Entropy calibration
  - Percentile calibration



如同字面意思，表示的是FP32中占据99.99%的浮点数参与量化。这样可以避免极个别特殊点(误差)参与量化，导出量化出现问题

Percentile有99.9%, 99.99%, 99.999%等等

# Calibration algorithm

## 7.3. Post-Training Quantization Using Calibration

In post-training quantization, TensorRT computes a scale value for each tensor in the network. This process, called *calibration*, uses statistics from an activation tensor.

The amount of input data required is application-dependent, but experiments indicate that about 500 images are sufficient.

Given the statistics for an activation tensor, deciding on the best scale value is not an exact science - it requires balancing two factors: *overflow error* (where the quantized value becomes larger) and *truncation error* (where values are clamped to the limits of the representable range.) The calibration step performs layer fusion for GPU to optimize away unneeded Tensors before performing calibration. This can be problematic if the calibration step is run on CPU.

Calibration batch size can also affect the *truncation error* for `IInt8EntropyCalibrator2` and `IInt8EntropyCalibrator`. If the calibration step is run on CPU, a large batch size may result in a poor scale value. For each calibration step, TensorRT updates the histogram distribution for each activation tensor. If it encouters a new maximum value, it increases the histogram bin width by two to accommodate the new maximum value. This approach works well unless histogram reallocation occurs in the last calibration step. This also makes calibration susceptible to the order of calibration batches, that is, a different order of calibration batches can result in different scale values. To avoid this issue, calibrate with as large a single batch as possible, and ensure that calibration batches are well randomized and have similar sizes.

### `IInt8EntropyCalibrator2`

Entropy calibration chooses the tensor's scale factor to optimize the quantized tensor's information-theoretic content. It is the recommended choice for DLA. Calibration happens before Layer fusion by default. Calibration batch size may impact the final result. It is recommended to use a large batch size.

### `IInt8MinMaxCalibrator`

This calibrator uses the entire range of the activation distribution to determine the scale factor. It seems to work better than the EntropyCalibrator for BERT (an optimized version of [Google's official implementation](#)).

### `IInt8EntropyCalibrator`

This is the original entropy calibrator. It is less complicated to use than the LegacyCalibrator and typically produces better results.

### `IInt8LegacyCalibrator`

This calibrator is for compatibility with TensorRT 2.0 EA. This calibrator requires user parameterization and is provided for legacy compatibility. It is recommended to customize this calibrator to implement percentile max, for example, 99.99% percentile max is observed to have best results.

When building an INT8 engine, the builder performs the following steps:

1. Build a 32-bit engine, run it on the calibration set, and record a histogram for each tensor of the distribution of activation values.
2. Build from the histograms a calibration table providing a scale value for each tensor.
3. Build the INT8 engine from the calibration table and the network definition.

## 如何选择calibration algorithm

(思考)如同quantization granularity对不同的情况会有不同的策略。calibration的算法选择也会根据输入是activation value还是weights而改变。那么我们该如何选择呢？

# 如何选择calibration algorithm

calibration算法的选择，我们按照这个规律去选

- weight的calibration，选用minmax
- activation的calibration，选用entropy或者percentile

Model	fp32	Accuracy	Relative	Model	fp32	Accuracy	Relative
MobileNet v1	71.88	71.59	-0.40%	Faster R-CNN	36.95	36.86	-0.24%
MobileNet v2	71.88	71.61	-0.38%	Mask R-CNN	37.89	37.84	-0.13%
ResNet50 v1.5	76.16	76.14	-0.03%	Retinanet	39.30	39.20	-0.25%
ResNet152 v1.5	78.32	78.28	-0.05%	FCN	63.70	63.70	0.00%
Inception v3	77.34	77.44	0.13%	DeepLabV3	67.40	67.40	0.00%
Inception v4	79.71	79.64	-0.09%	GNMT	24.27	24.41	0.58%
ResNeXt50	77.61	77.62	0.01%	Transformer	28.27	28.58	1.10%
ResNeXt101	79.30	79.29	-0.01%	Jasper	96.09	96.10	0.01%
EfficientNet b0	76.85	76.72	-0.17%	Bert Large	91.01	90.94	-0.08%
EfficientNet b3	81.61	81.55	-0.07%				

Table 4: Accuracy with int8 quantization of weights only: per-channel granularity, max calibration

Models	fp32	Max	Entropy	99.9%	99.99%	99.999%	99.9999%
MobileNet v1	71.88	69.51	70.19	<b>70.39</b>	70.29	69.97	69.57
MobileNet v2	71.88	69.41	70.28	70.68	<b>71.14</b>	70.72	70.23
ResNet50 v1.5	76.16	75.82	<b>76.05</b>	75.68	75.98	75.97	76.00
ResNet152 v1.5	78.32	77.93	<b>78.21</b>	77.62	78.17	78.17	78.19
Inception v3	77.34	72.53	<b>77.54</b>	76.21	77.52	77.43	77.37
Inception v4	79.71	0.12	79.60	78.16	<b>79.63</b>	79.12	71.19
ResNeXt50	77.61	77.31	<b>77.46</b>	77.04	77.39	77.45	77.39
ResNeXt101	79.30	78.74	79.09	78.77	79.15	<b>79.17</b>	79.05
EfficientNet b0	76.85	22.3	<b>72.06</b>	70.87	68.33	51.88	42.49
EfficientNet b3	81.61	54.27	76.96	77.80	<b>80.28</b>	80.06	77.13
Faster R-CNN	36.95	36.38	<b>36.82</b>	35.22	36.69	36.76	36.78
Mask R-CNN	37.89	37.51	37.75	36.17	37.55	37.72	<b>37.80</b>
Retinanet	39.30	38.90	38.97	35.34	38.55	<b>39.19</b>	<b>39.19</b>
FCN	63.70	63.40	64.00	62.20	64.00	<b>63.90</b>	63.60
DeepLabV3	67.40	67.20	67.40	66.40	67.40	<b>67.50</b>	67.40
GNMT	24.27	24.31	<b>24.53</b>	24.34	24.36	24.38	24.33
Transformer	28.27	21.23	21.88	24.49	<b>27.71</b>	20.22	20.44
Jasper	96.09	95.99	<b>96.11</b>	95.77	96.09	96.09	96.03
BERT Large	91.01	85.92	37.40	26.18	89.59	<b>90.20</b>	90.10

Table 5: Post training quantization accuracy. Weights use per-channel or per-column max calibration. Activations use the calibration listed. Best quantized accuracy per network is in bold.

(左)固定activation values的精度，量化weights。 (右)固定weights的精度，量化activation values  
我们可以看出不同模型的量化选择上会有不同的策略。 calibration的选择与模型架构相关

# calibration dataset与batch size的关系

在使用calibration dataset中构建histogram是需要注意的一个点：calibration时的batch size会影响精度。更准确来说会影响histogram的分布，这个跟TensorRT在构建浮点数的histogram的算法有关

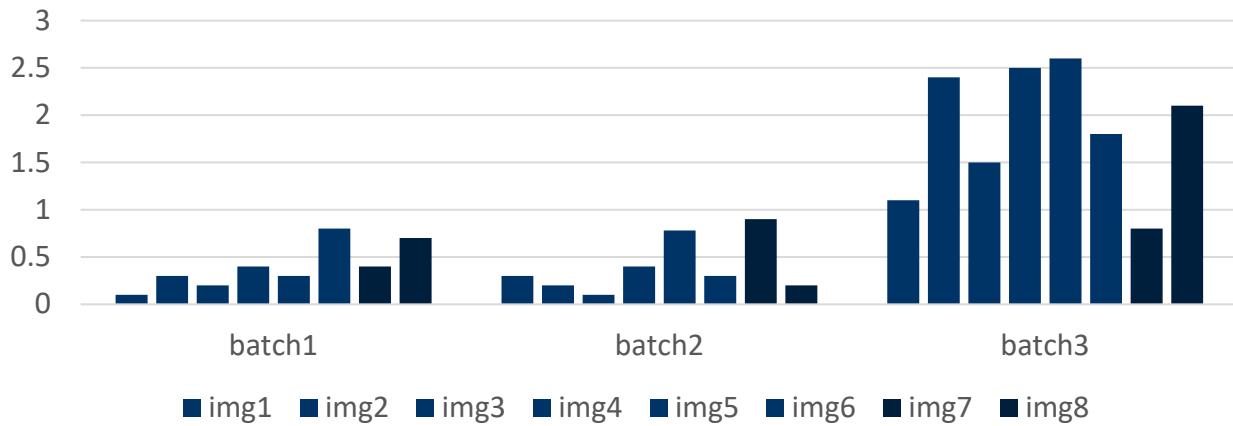
Calibration batch size can also affect the truncation error for `IInt8EntropyCalibrator2` and `IInt8EntropyCalibrator`. For example, calibrating using multiple small batches of calibration data may result in reduced histogram resolution and poor scale value. For each calibration step, TensorRT updates the histogram distribution for each activation tensor. If it encounters a value in the activation tensor, larger than the current histogram max, the histogram range is increased by a power of two to accommodate the new maximum value. This approach works well unless histogram reallocation occurs in the last calibration step, resulting in a final histogram with half the bins empty. Such a histogram can produce poor calibration scales. This also makes calibration susceptible to the order of calibration batches, that is, a different order of calibration batches can result in the histogram size being increased at different points, producing slightly different calibration scales. To avoid this issue, calibrate with as large a single batch as possible, and ensure that calibration batches are well randomized and have similar distribution.

上面的说法表明：在创建histogram直方图的时候，如果出现了大于当前histogram可以表示的最大值的时候，TensorRT会直接平方当前histogram的最大值，来扩大存储空间

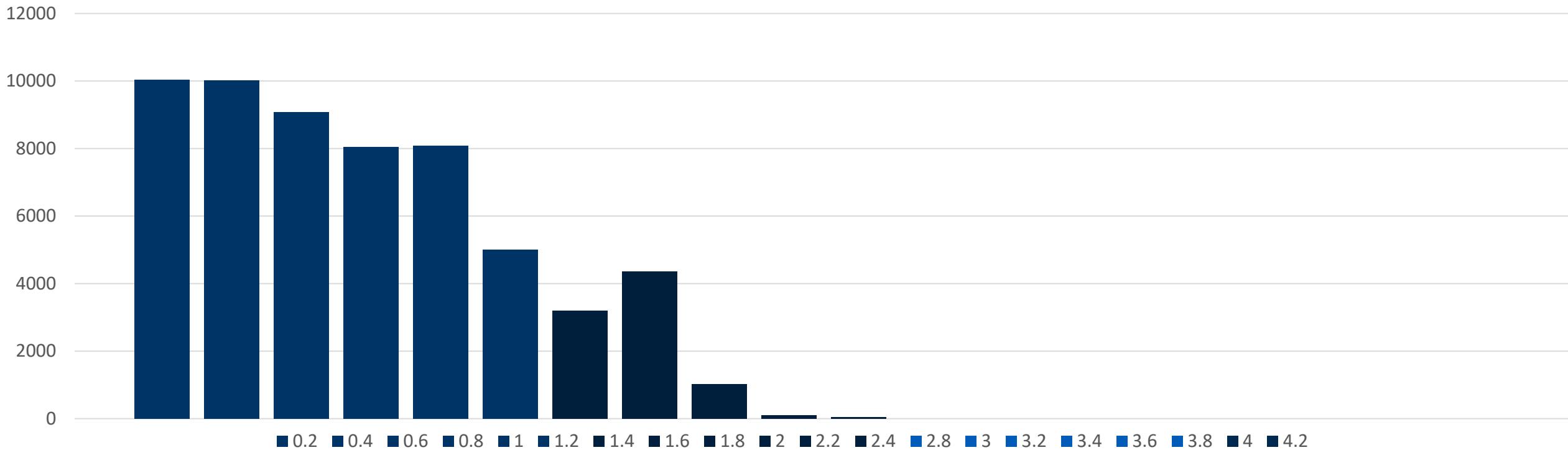
- (思考)如果batchsize=1，最后一个batch的浮点数很大，那么最终的histogram会呈现什么形状？
- (思考)如果batchsize=16，但每一个batch size的数据分布很均匀，histogram会呈现什么形状？
- (思考)如果模型的鲁棒性很强，batchsize=1和batchsize=16/32/64/128的区别会有吗

# calibration dataset与batch size的关系

batch size = 8

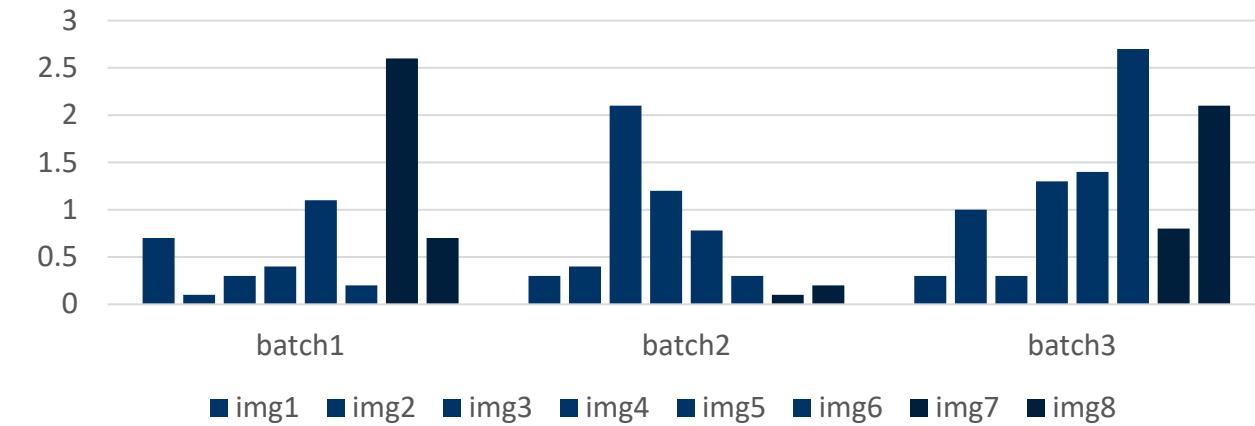


这是histogram的后半段很稀疏，甚至没有数据。在量化的时候会根据这个直方图来将FP32转为INT8，很显然这块领域是多余的



# calibration dataset与batch size的关系

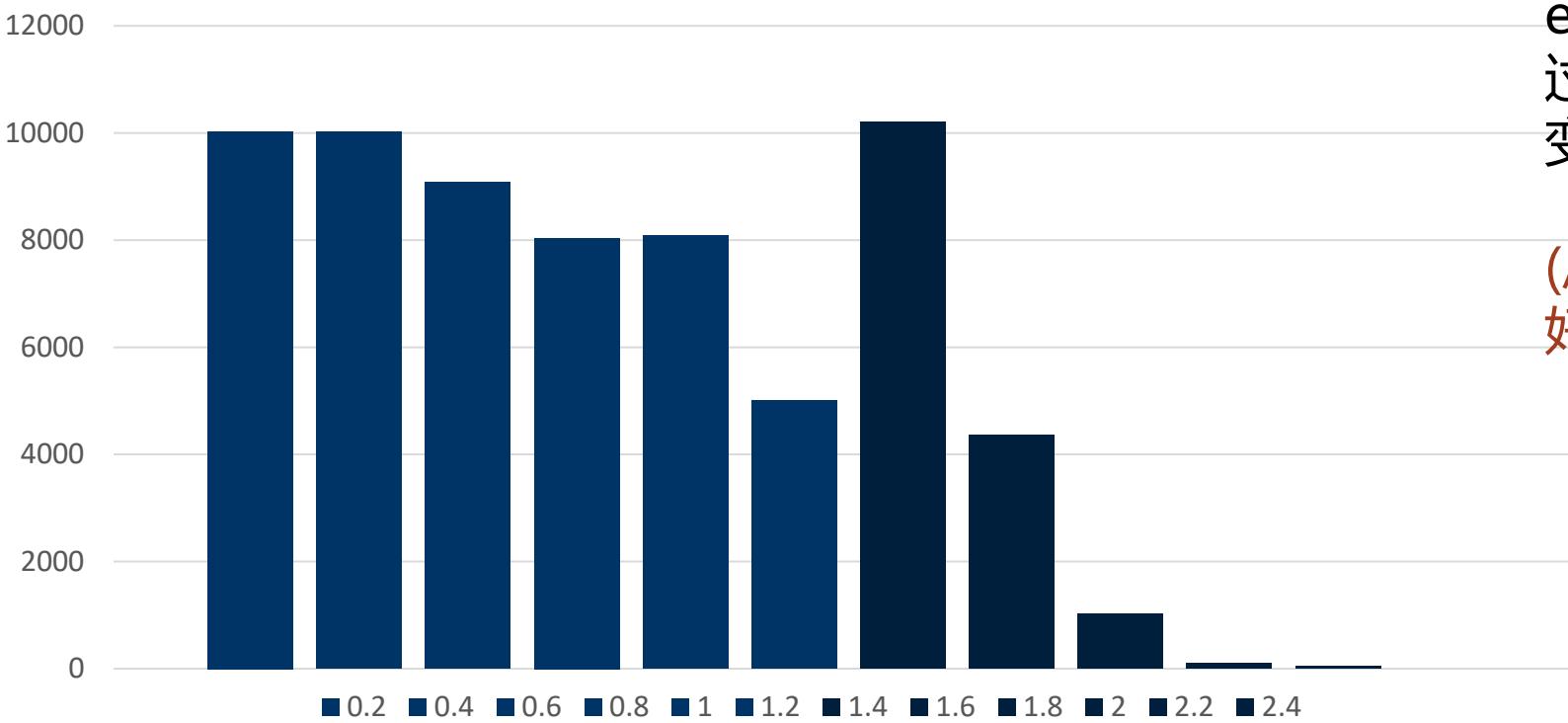
batch size = 8



我们希望每一个batch里面的数据比较均匀，让比较大的数据出现的时候，histogram的范围已经能够表现它了。

e.g. 当2.4出现的时候，如果之前已经出现过1.54，那么histogram的range不需要改变。否则range的最大值会变成5.76

(总的来讲，calibration的batch size越大越好，但不是绝对的)





03

## Quantization (PTQ-and- quantization-analysis)

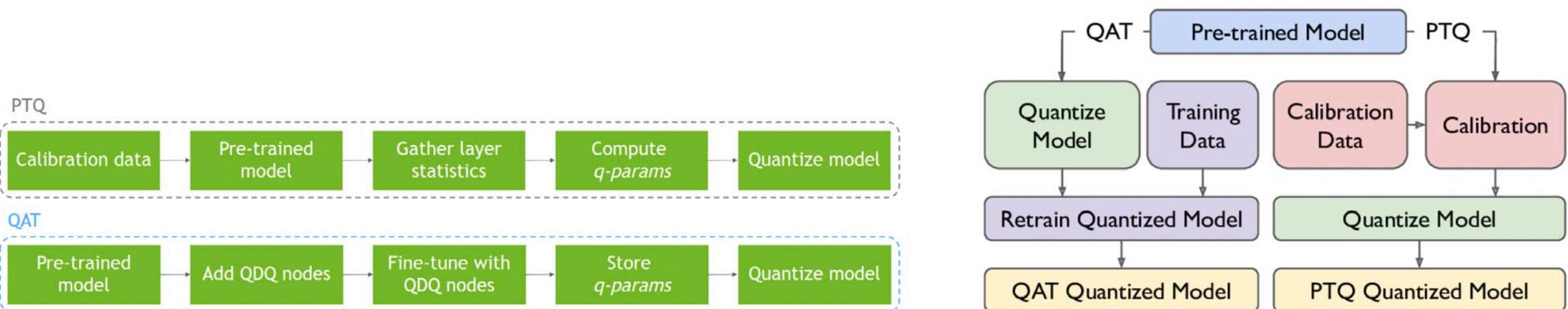
Goal: 理解PTQ和QAT的区别，以及PTQ的优缺点和layer-wise sensitive analysis

# 量化的基本原理: 基本术语(PTQ, QAT)

根据量化的时机，一般我们会把量化分为

- PTQ(Post-Training Quantization), 训练后量化
- QAT(Quantization-Aware Training), 训练时量化

PTQ一般是指对于训练好的模型，通过calibration算法等来获取dynamic range来进行量化。但量化普遍上会产生精度下降。所以QAT为了弥补精度下降，在学习过程中通过Fine-tuning权重来适应这种误差，实现精度下降的最小化。所以一般来讲，QAT的精度会高于PTQ。但并不绝对。



PTQ与QAT的workflow上的不同<sup>[1, 2]</sup>

[1] Accelerating Quantized Networks with the NVIDIA QAT Toolkit for TensorFlow and NVIDIA TensorRT

[2] Neural Network Quantization for Efficient Inference: A Survey

# PTQ是什么

PTQ(Post-training quantization)也被称作隐式量化(implicit quantization)。我们并不显式的对算子添加量化节点(Q/DQ)，calibration之后TensorRT根据情况进行量化

## 1. Post Training Quantization (PTQ)

Post Training Quantization computes *scale* after network has been trained. A representative dataset is used to capture the distribution of activations for each activation tensor, then this distribution data is used to compute the *scale* value for each tensor. Each weight's distribution is used to compute weight *scale*.

TensorRT provides a workflow for PTQ, called *calibration*.



trtexec在选择参数进行fp16或者int8指定的时候，使用的就是PTQ。(int8的时候需要指定calibration dataset)。很方便使用，但是我们需要先理解PTQ的利弊

# PTQ(优缺点分析)

## 优点

- 方便使用，不需要训练。可以在部署设备上直接跑

## 缺点

### 1. 精度下降

- 量化过程会导致精度下降。但PTQ没有类似于QAT这种fine-tuning的过程。所以权重**不会更新**来吸收这种误差

### 2. 量化不可控

- TensorRT会权衡量化后所产生的新添的计算或者访存，是否用INT8还是FP16。
- TensorRT中的kernel autotuning会选择核函数来做FP16/INT8的计算。来查看是否在CUDA core上跑还是在Tensor core上跑
- 有可能FP16是在Tensor core上，**但转为INT8之后就在CUDA core上了**

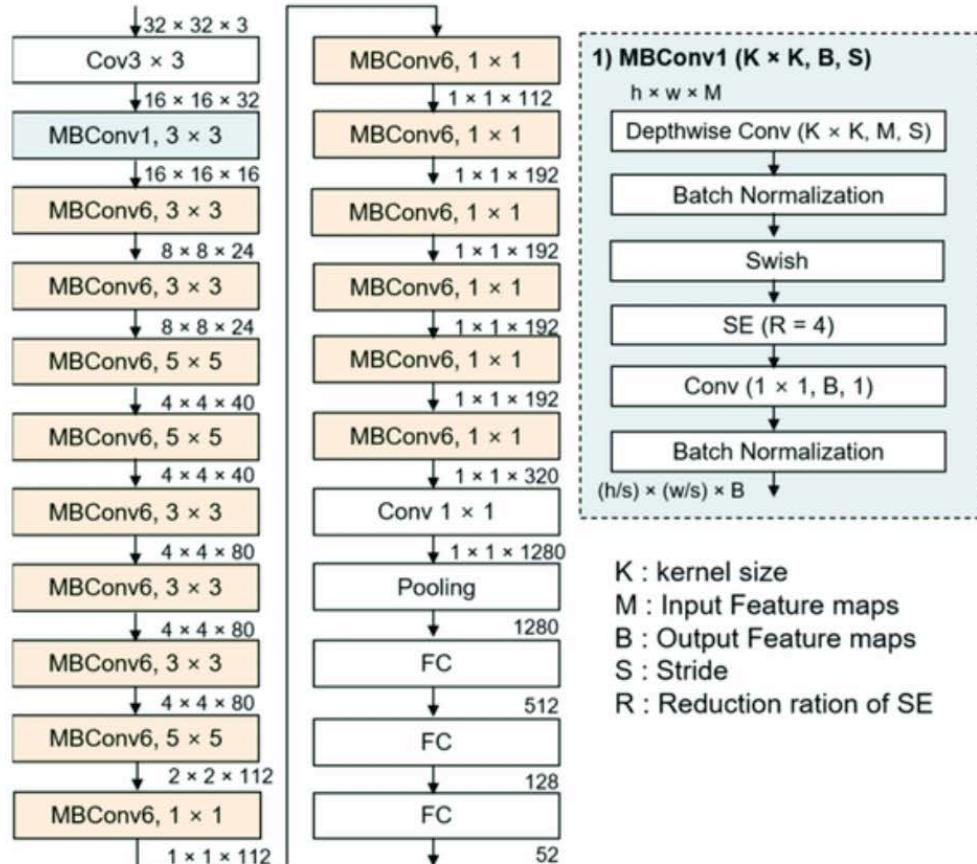
### 3. 层融合问题

- 量化后有可能出现之前可以融合的层，不能融合了
- 量化会添加reformatter这种更改tensor的格式的算子，如果本来融合的两个算子间添加了这个就不能被融合了
- 比如有些算子支持int8，但某些不支持。之前可以融合的，但因为精度不同不能融合了

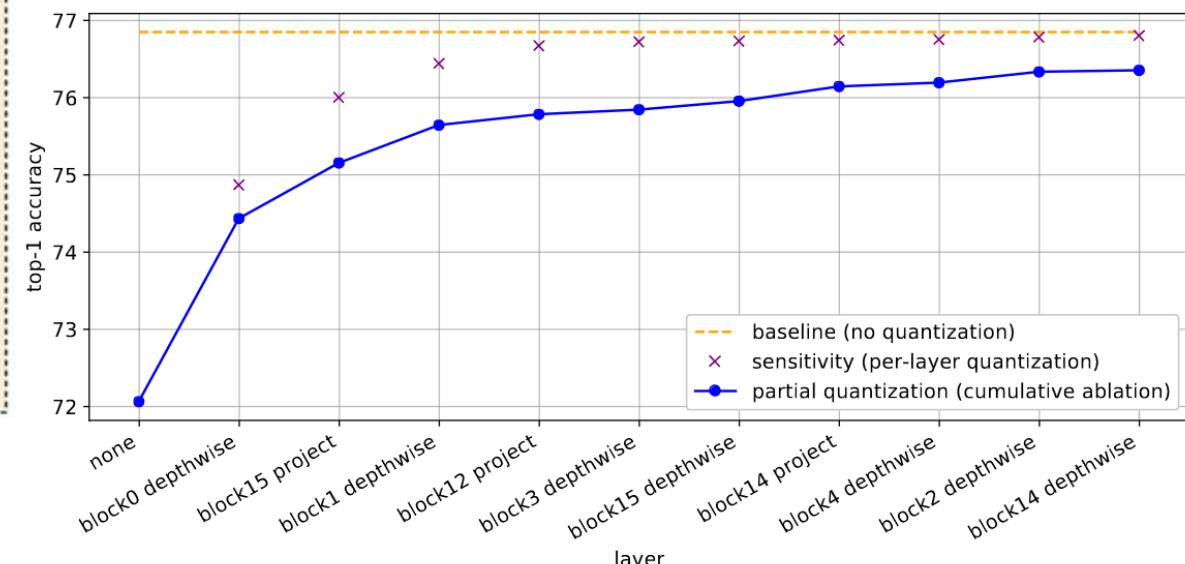
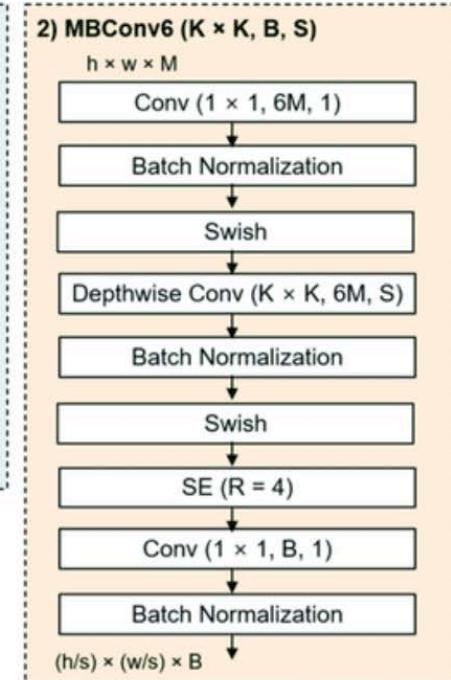
**如果INT8量化后速度反而会比FP16/FP32要慢，我们可以从以上的2和3去分析并排查原因**

# 量化中的sensitive analysis

从精度分析的角度去弥补PTQ的精度下降，我们可以进行layer-wise的量化分析。这种方法被称作layer-wise sensitive analysis



EfficientNetb0的模型框架<sup>[1]</sup>



对EfficientNetb0的各层进行量化分析，寻找影响精度的层<sup>[2]</sup>

[1]The structure of efficientNetb0

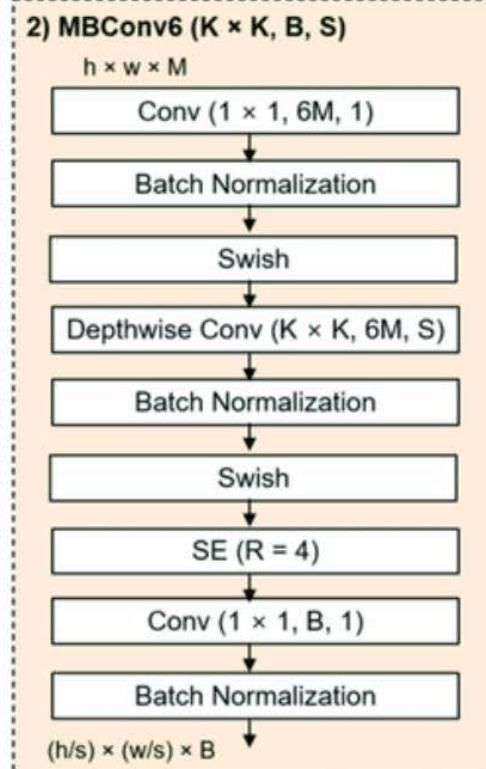
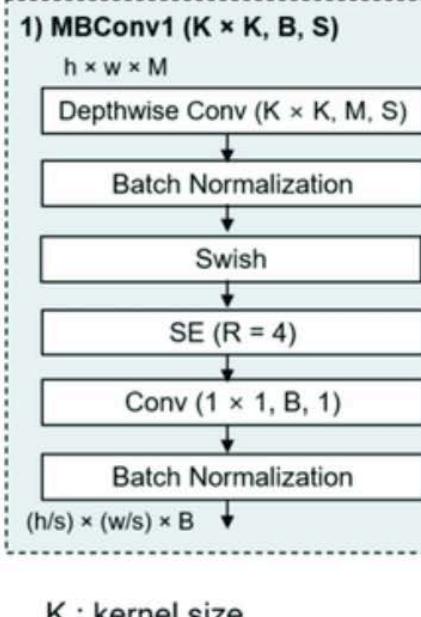
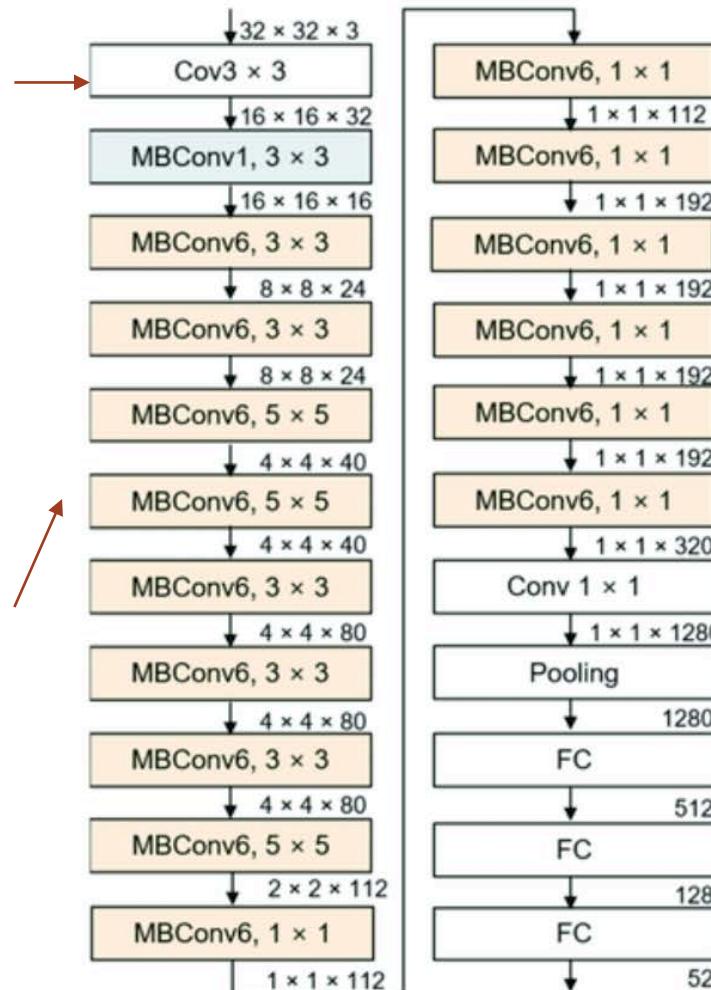
[2] Integer Quantization for Deep Learning Inference: Principles and Empirical Evaluation

# 量化中的sensitive analysis

普遍来讲，模型框架中会有一些层的量化对精度的影响比较大。我们管它们叫做敏感层(sensitive layer)。对于这些敏感层的量化我们需要非常小心。尽量用FP16。敏感层一般靠近模型的输入输出

靠近输入属于敏感层。  
channel还比较小，每一个位置所具有的特征量可能还比较分散。建议FP16

模型中间部分，计算比较密集，特征量也比较大。建议INT8



K : kernel size  
M : Input Feature maps  
B : Output Feature maps  
S : Stride  
R : Reduction ration of SE

靠近输出属于敏感层。后处理的部分对这里的tensor的数据要求比较高。建议FP16

最终模型的推理引擎是FP16+INT8精度的

# 学习使用Polygraphy

对于这种sensitive analysis，其实NVIDIA最近几年推出了一个叫做polygraphy的一个工具。可以用来分析并查找模型精度下降并且影响比较大的地方。非常好的工具，做TensorRT量化必须要掌握。能做的事情太多，比如

- onnxruntime与TensorRT engine的layer-wise的精度分析
- 输出每一层layer的权重histogram
- 截取影响整个网络中对精度影响最大的子网，并使用onnx-surgeon单独拿出来

```
$ polygraphy run example-fixed.onnx --trt --onnxrt --int8
[I] trt-runner | Activating and starting inference
[I] Loading bytes from example-fixed.onnx
[I] Building engine with configuration: max_workspace_size=16777216 (16.00 MB) | tf32=False,
    fp16=False, int8=True, strict_types=False | 1 profiles
[D] trt-runner | Completed 1 iteration(s) in 0.93 ms | Average inference time: 0.93[1] ms.
[I] onnxrt-runner. | Activating and starting inference
[D] onnxrt-runner | Completed 1 iteration(s) in 0.066 ms | Average inference time: 0.066 ms.
[I] Accuracy Comparison | trt-runner vs. onnxrt-runner
[I]     Comparing Output: 'output' (dtype=float32, shape=(1, 3))
[D]         PASSED | Difference is within tolerance (rel=1e-05, abs=1e-05)
[D]         PASSED | All outputs matched | Outputs: ['output']
[D] PASSED | Command: polygraphy run example-fixed.onnx --trt --onnxrt --int8
```

Polygraphy是TensorRT官方内部的一个tool，可以在TensorRT的git repository<sup>(\*)</sup>中找到。建议大家自己跟着它给的README.md学习一下。以后有时间单独讲。

## FP16/INT8对计算资源的利用

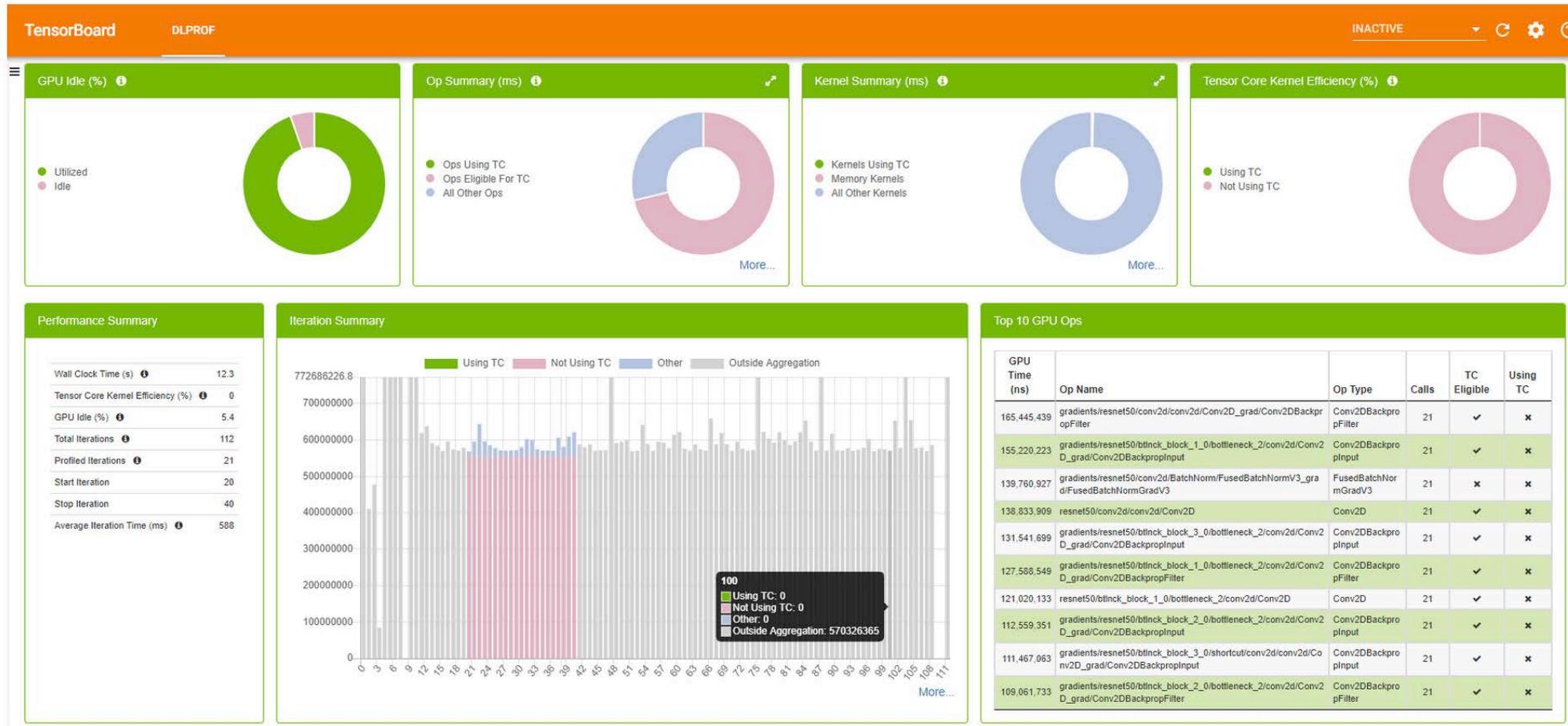
我们在做量化后，我们无法指定将量化后的conv或者gemm放在Tensor core还是在CUDA core上计算。这些是TensorRT在帮我们选择核函数的时候自动完成的。那么我们如何查看我们是否在用Tensor core呢？

我们一般有这么几个办法

- 使用dlprof
- 使用nsight system
- 使用trtexec

# DLProf

DLProf (Deep learning Profiler)工具可以把模型在GPU上的执行情况以TensorBoard的形式打印出来，分析TensorCore的使用情况。感兴趣的可以查看一下。但需要注意的是，DLProf不支持Jetson系列的Profile。对于Jetson，我们可以使用Nsight system或者trtexec



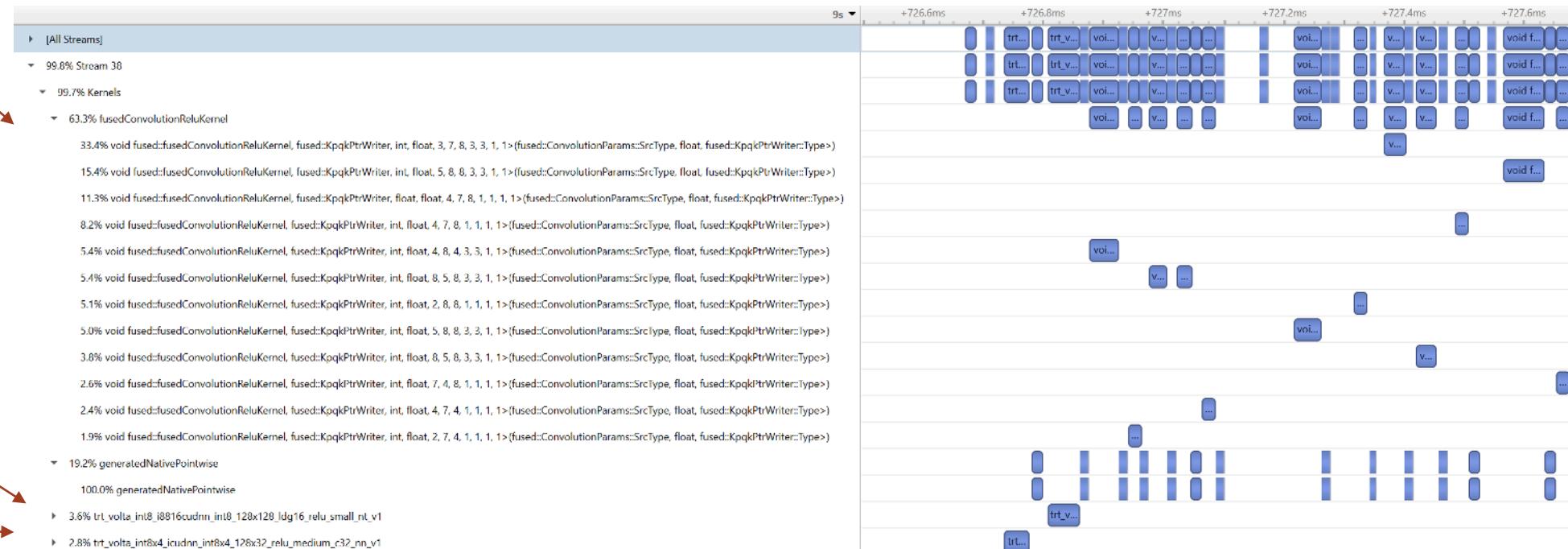
# Nsight System/trtexec

如果是利用Nsight system的话，我们可以查看到哪一个kernel的时间占用率最高，之后从kernel的名字取推测这个kernel是否在用Tensor Core。(从kernel名字推测kernel的计算设备需要经验)

这个使用的是CUDA Core

这个使用的是Tensor Core

这个使用的是CUDA Core



从kernel名字推测可以从kernel中的关键字去猜，比如

- h884 = HMMA = FP16 TensorCore
  - i8816 = IMMA = INT8 TensorCore
  - hcudnn = FP16 normal CUDA kernel (without TensorCore)
  - icudnn = INT8 normal CUDA kernel (without TensorCore)
  - scudnn = FP32 normal CUDA kernel (without TensorCore)



03

## Quantization (QAT-and- kernel-fusion)

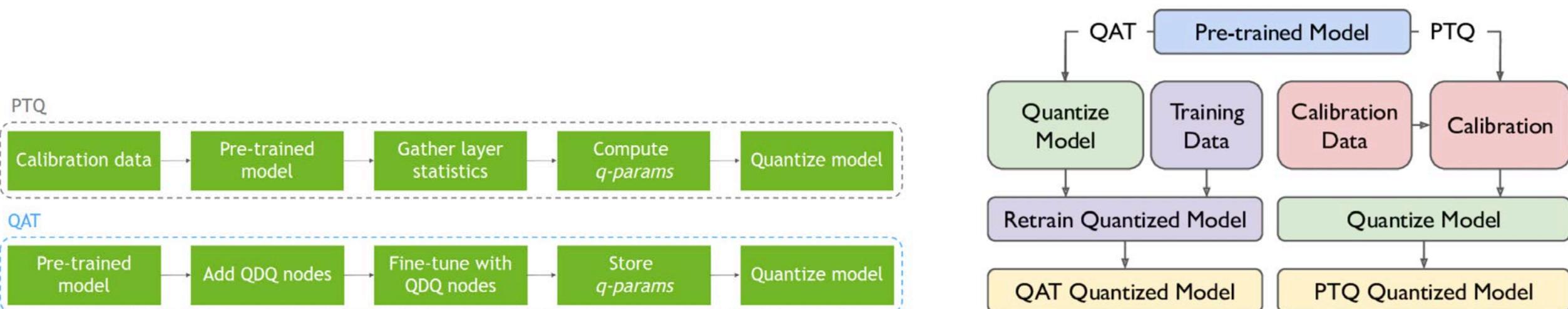
Goal: 理解什么是QAT，以及Q/DQ是什么， Q/DQ与算子的融合是如何做的

# 量化的基本原理: 基本术语(PTQ, QAT)

根据量化的时机，一般我们会把量化分为

- PTQ(Post-Training Quantization), 训练后量化
- QAT(Quantization-Aware Training), 训练时量化

PTQ一般是指对于训练好的模型，通过calibration算法等来获取dynamic range来进行量化。但量化普遍上会产生精度下降。所以QAT为了弥补精度下降，在学习过程中通过Fine-tuning权重来适应这种误差，实现精度下降的最小化。所以一般来讲，QAT的精度会高于PTQ。但并不绝对。



PTQ与QAT的workflow上的不同<sup>[1, 2]</sup>

[1] Accelerating Quantized Networks with the NVIDIA QAT Toolkit for TensorFlow and NVIDIA TensorRT

[2] Neural Network Quantization for Efficient Inference: A Survey

# QAT是什么

QAT(Quantization Aware Training)也被称作显式量化。我们明确的在模型中添加Q/DQ节点(量化/反量化)，来控制某一个算子的精度。并且通过fine-tuning来更新模型权重，让权重学习并适应量化带来的精度误差

## 2. Quantization Aware Training (QAT)

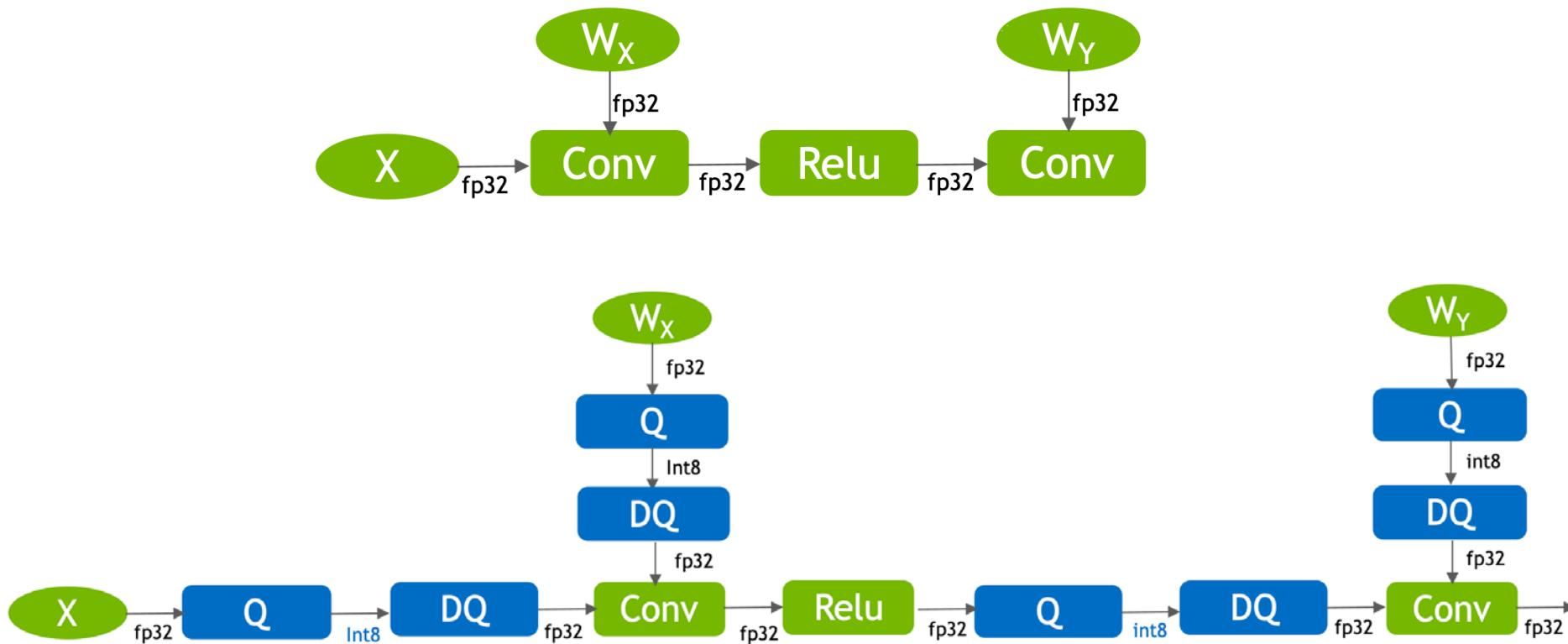
Quantization Aware Training aims at computing scale factors during training. Once the network is fully trained, Quantize (Q) and Dequantize (DQ) nodes are inserted into the graph following a specific set of rules. The network is then further trained for few epochs in a process called *Fine-Tuning*. Q/DQ nodes simulate quantization loss and add it to the training loss during fine-tuning, making the network more resilient to quantization. In other words, QAT is able to better preserve accuracy when compared to PTQ.



QAT的核心就是通过添加fake quantization，也就是Q/DQ节点，来模拟量化过程。在谈QAT的优缺点前，我们先看一下Q/DQ是什么，以及QAT的流程

# Q/DQ是什么

Q/DQ node也被称作fake quantization node，是用来模拟fp32->int8的量化的scale和shift(zero-point)，以及int8->fp32的反量化的scale和shift(zero-point)。QAT通过Q和DQ node里面存储的信息对fp32或者int8进行线性变换



Q/DQ节点的插入。上图为没有QAT的默认onnx模型架构，下图为带有QAT的onnx模型架构<sup>[1]</sup>

## Q/DQ是什么

我们回顾一下之前mapping range的小节讲的。不妨我们把Q/DQ按照公式的形式展现一下。这里面有几个符号，说明一下：

- $x$ : 表示量化前的值，是 $fp32$ 精度
- $x_q$ : 表示量化后的值，是 $int8$ 精度
- $\hat{x}$ : 表示通过量化反量化计算后得到的 $fp32$ 精度的值，因为浮点计算会有误差，所以近似等于 $x$
- $b$ : 表示 $b$ -bit的integer可以表示范围从 $\{-2^{b-1}, -2^{b-1} + 1, \dots, 2^{b-1} - 1\}$ 。
  - 比如 $int32$ 的话， $b = 32$ , 可以表示 $(-2,147,483,648, 2,147,483,647)$ 范围的数据
  - 比如 $int8$ 的话， $b = 8$ , 可以表示 $(-128, 127)$ 范围的数据
  - 比如 $int4$ 的话， $b = 4$ , 可以表示 $(-8, 7)$ 范围的数据
- $\beta$ : 表示量化前 $fp32$ 的 $range$ 的下限
- $\alpha$ : 表示量化前 $fp32$ 的 $range$ 的上限
- $s$ : 表示量化计算所需要的 $scale$
- $z$ : 表示量化计算所需要的 $shift$

$$s = \frac{2^b - 1}{\alpha - \beta}$$

那么Q的公式可以理解为：

$$z = -\text{round}(\beta \cdot s) - 2^{b-1}$$

$$x_q = \text{quantize}(x, b, s, z) = \text{clip}(\text{round}(s \cdot x + z), -2^{b-1}, 2^{b-1} - 1)$$

DQ的公式可以理解为：

$$\hat{x} = \text{dequantize}(x_q, s, z) = \frac{1}{s}(x_q - z)$$

$$\text{clip}(x, l, u) \begin{cases} l, & x < l \\ x, & l \leq x \leq u \\ u, & x > u \end{cases}$$

## Q/DQ是什么 + 可量化层的计算

在理解了上面的公式的基础上，我们在进一步理解这个：对于一个线性计算的op(conv或者linear)。我们把fp32精度的op的计算简化成

$$op_{FP32}(w, x) = w * x$$

既然x和w是fp32的，那么我们也可以这么表示

$$op_{FP32}(w, x) = dequantize(s_w, w_q) * dequantize(s_x, x_q)$$

展开

$$op_{FP32}(w, x) = \frac{1}{s_w} * w_q * \frac{1}{s_x} * x_q$$

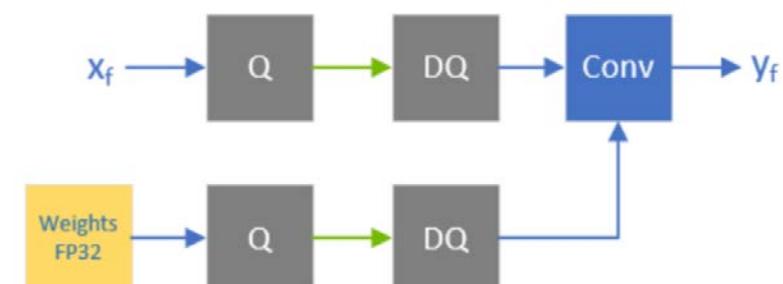
$$op_{FP32}(w, x) = \frac{1}{s_w * s_x} * w_q * x_q$$

因为计算量主要是 $w_q * x_q$ , 是int8计算，所以我们可以把这个公式写成：

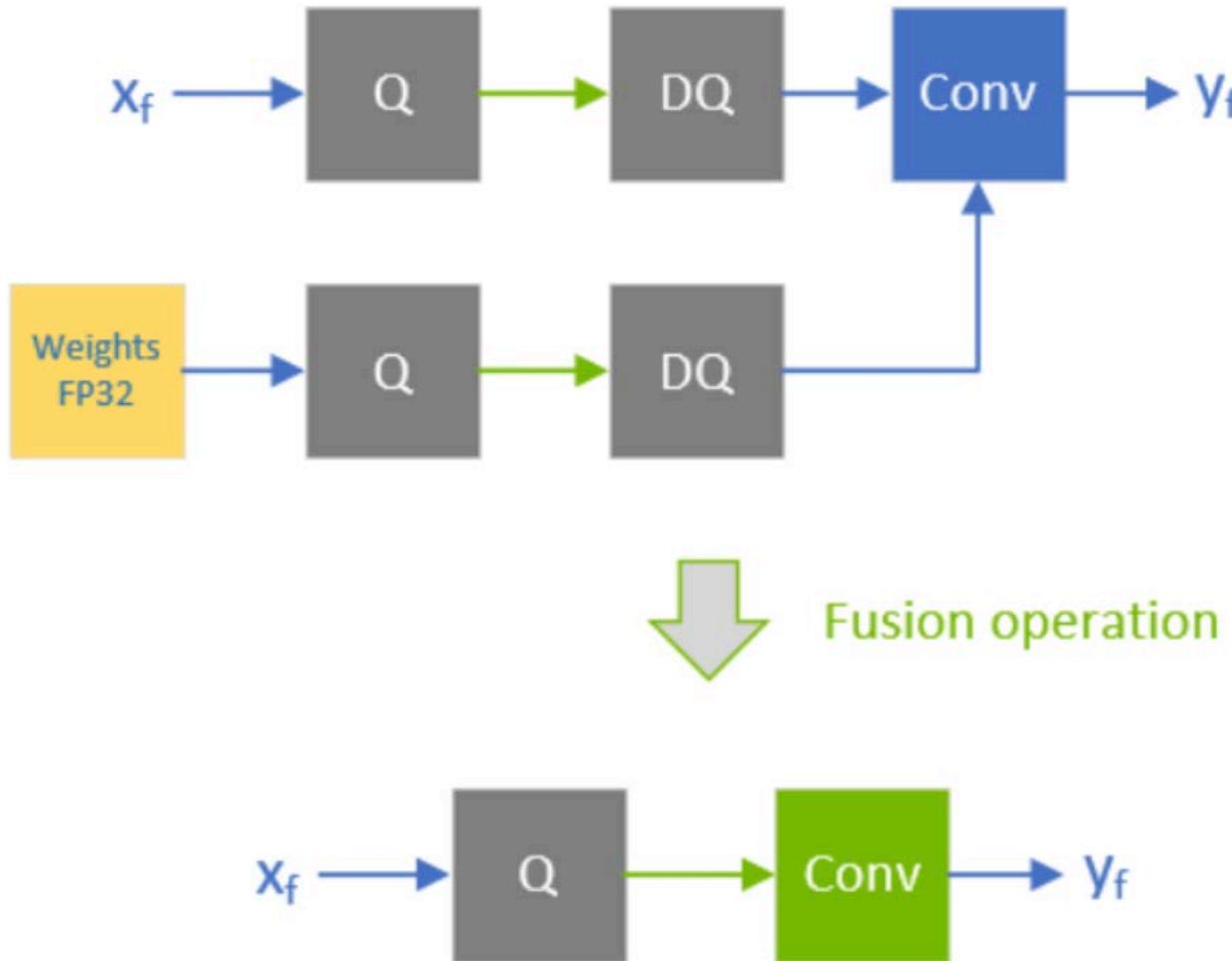
$$op_{int8}(w_q, x_q, s_w, s_x) = \frac{1}{s_w * s_x} * w_q * x_q$$

所以我们就知道DQ + fp32精度的op可以拼成一个int8精度的op

这里以NVIDIA采用的对称量化量化与反量化计算为例，计算过程没有涉及zero-shift



## Q/DQ是什么 + 可量化层的计算



将上一页的公式用图比较直观的表示出来就是这个样子：

- 蓝色conv表示是fp32的op
- 绿色conv表示是int8的op
- 蓝色arrow表示的是fp32的tensor
- 绿色arrow表示的是int8的tensor

## Q/DQ是什么 + 可量化层的计算

如果DQ + fp32精度op可以拼成一个int8精度的op， 那么DQ + fp32精度op + Q是不是也可以融合呢？

$$\text{quantization}(x') = s_{x'} * x'$$

由于 $x'$ 是来自于上一层计算，可以把 $x'$ 展开

$$\text{quantization}(x') = s_{x'} * \frac{1}{s_w * s_x} * w_q * x_q$$

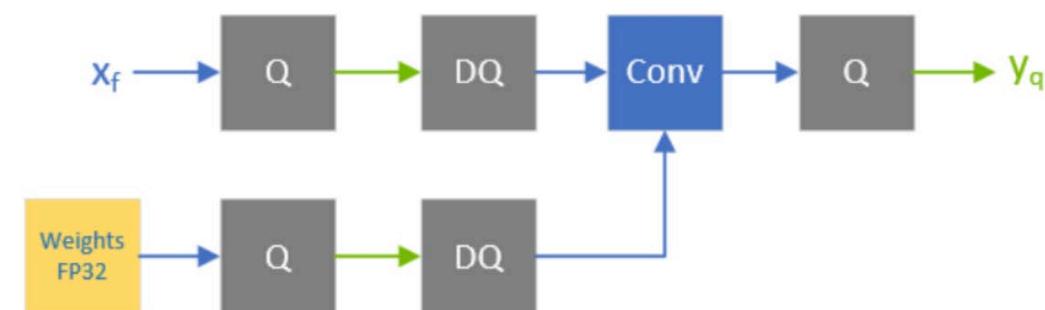
我们可以看到这个依然是一个线性变化。所以说DQ + fp32精度OP + Q可以融合在一起凑成一个int8的op，所以我们可以把这个公式替换成：

$$op_{int8}(x_q, w_q, s_w, s_x, s_{x'}) = s_{x'} * \frac{1}{s_w * s_x} * w_q * x_q$$

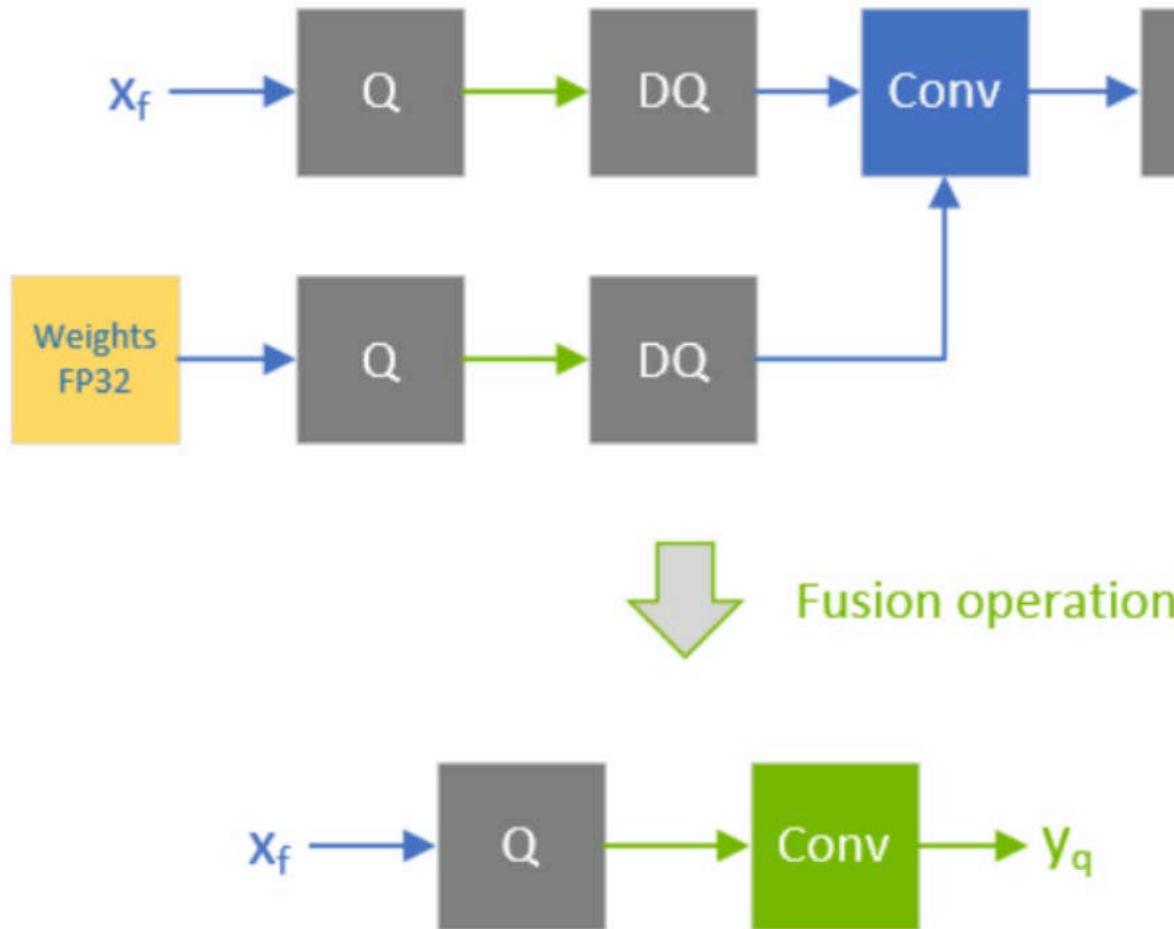
这里的 $x'$ 是来自于上一层的输出，是fp32

我们称这个op或者layer为quantizable layer，翻译为可量化层。

- 这个可量化层的输入和输出都是int8
- 计算的主体也是int8，可以节省带宽的同时，提高计算效率



# Q/DQ是什么 + 可量化层的计算

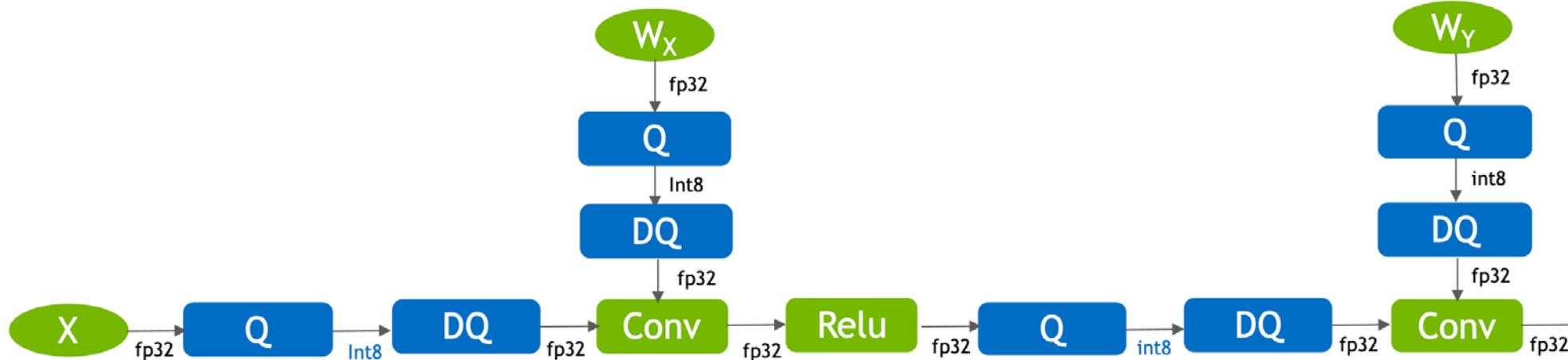


将上一页的公式用图比较直观的表示出来就是这个样子：

- 蓝色conv表示是fp32的op
- 绿色conv表示是int8的op
- 蓝色arrow表示的是fp32的tensor
- 绿色arrow表示的是int8的tensor

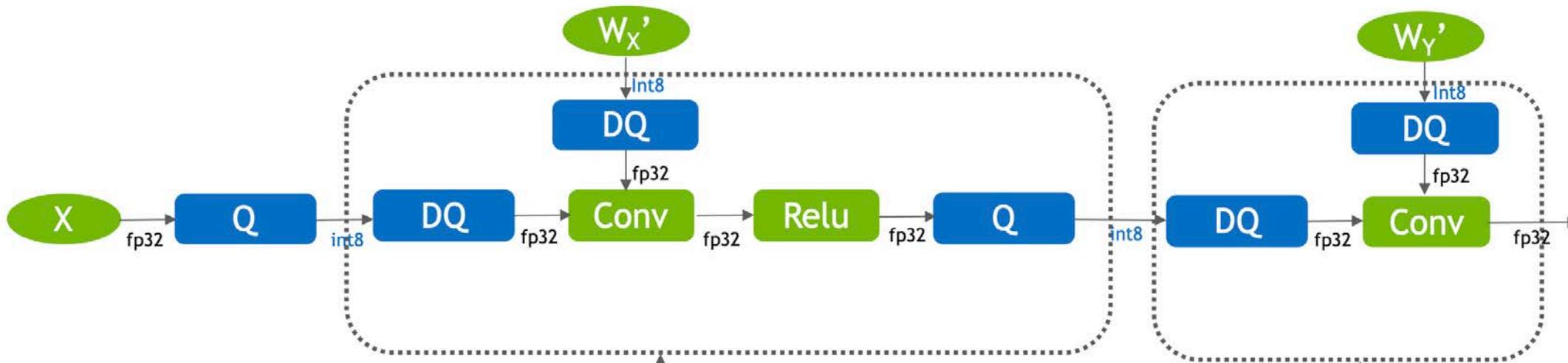
## Q/DQ是什么 + 可量化层的计算

我们理解了Q/DQ之后，我们再回到这张图看一下，

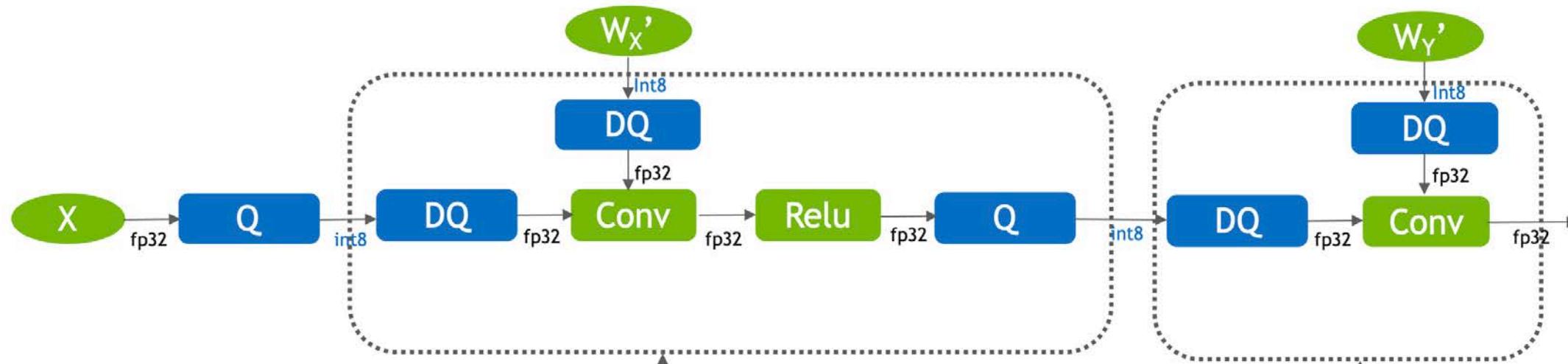


我们知道conv和Relu是可以融合在一起成为ConvReLU算子，同时根据之前的公式和图，我们知道：

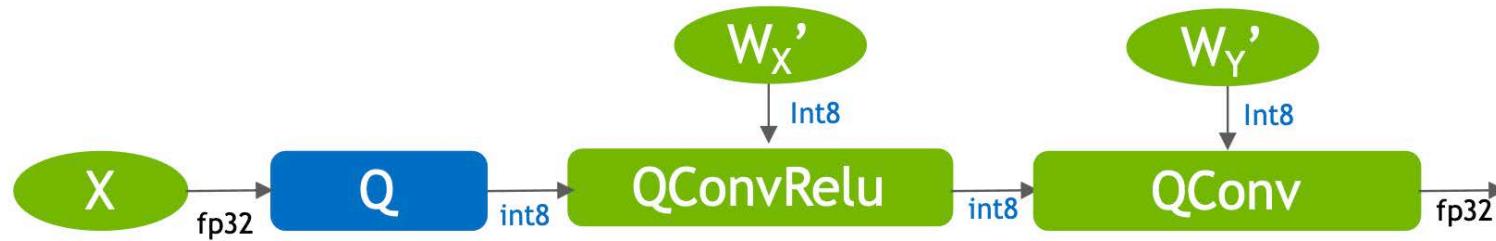
- DQ和fp32精度的conv组合在一起，可以融合成一个int8精度的conv
- fp32精度输出的conv和后面的Q也可以融合在一起，输出一个int8精度的activation value



## Q/DQ是什么 + 可量化层的计算



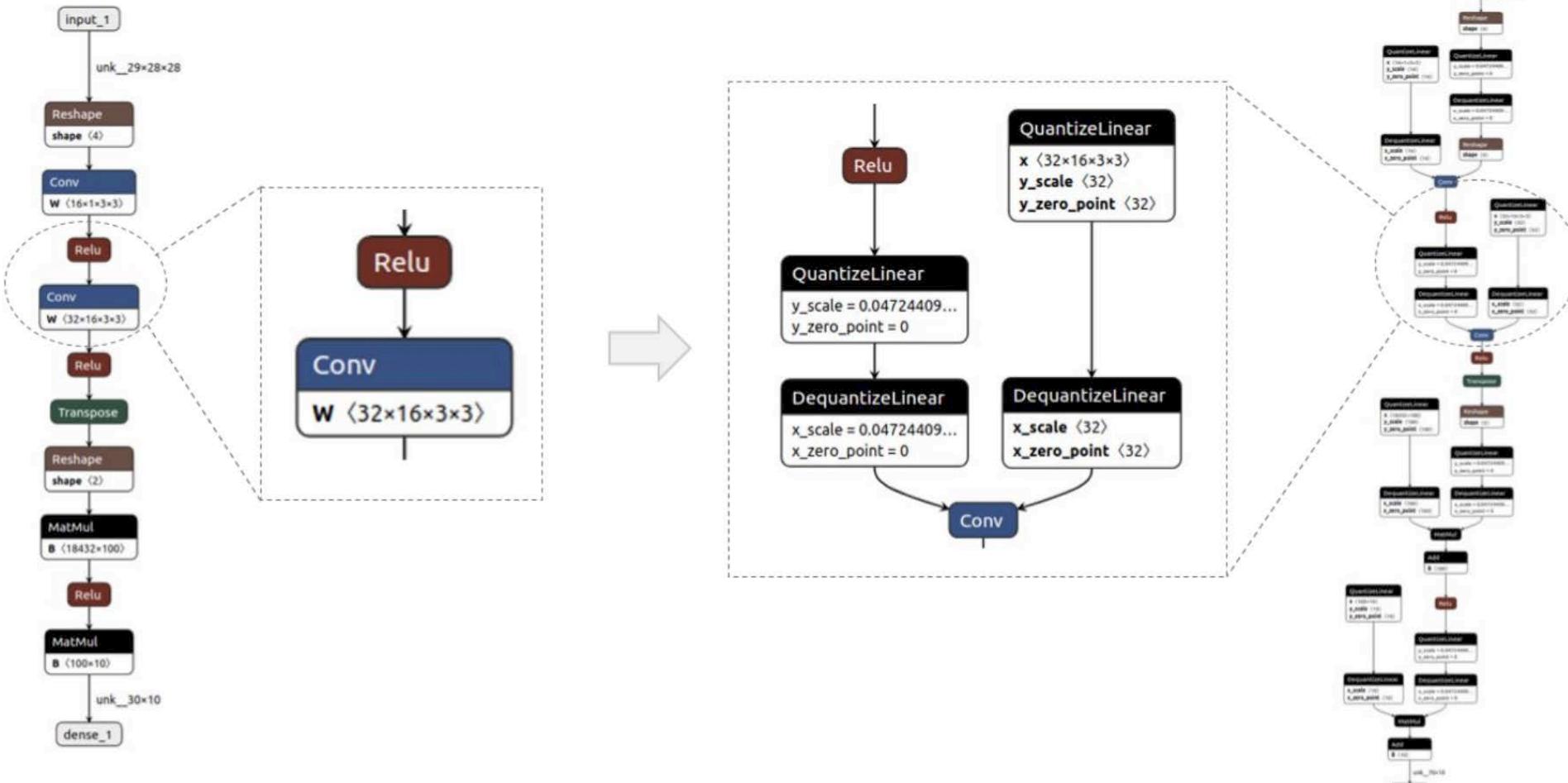
将这些虚线包围起来的算子融合在一起，用一个int8的op来替换后，整个网络就会变成这个样子



新生成的QConvRelu以及Qconv是int8精度的计算，速度很快并且TensorRT会很大几率分配tensor core执行这个计算。这个就是TensorRT中对量化节点的优化方法之一。还有一些其他优化方法在后面介绍

# QAT的工作流

理解了Q/DQ再去看QAT就非常容易了。QAT是一种Fine-tuning方式，通常对一个pre-trained model进行添加Q/DQ节点模拟量化，并通过训练来更新权重去吸收量化过程所带来的误差。添加了Q/DQ节点后的算子会以int8精度执行



pytorch支持对已经训练好的模型自动添加Q/DQ节点。详细可以参考  
<https://github.com/NVIDIA/TensorRT/tree/main/tools/pytorch-quantization>

# TensorRT中QAT的层融合的技巧

TensorRT对包含Q/DQ节点的onnx模型使用很多图优化，从而提高计算效率。主要分为

- Q/DQ fusion
  - 通过层融合，将Q/DQ中的线性计算与conv或者linear这种线性计算融合在一起，实现int8计算
- Q/DQ Propagation
  - 将Q节点尽量往前挪，将DQ节点尽量往后挪，让网络中int8计算的部分变得更长

# TensorRT中QAT的Q/DQ Fusion技巧

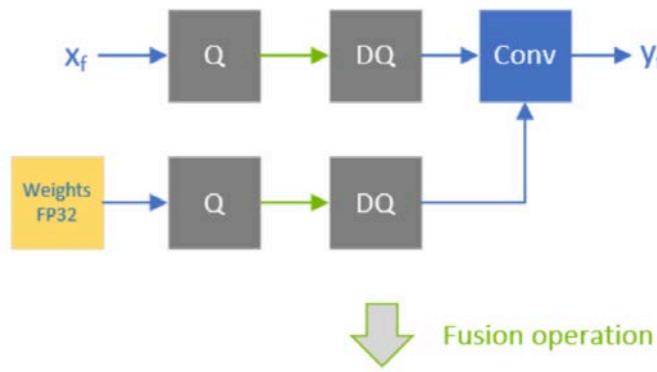


Fig.1 conv与DQ的融合

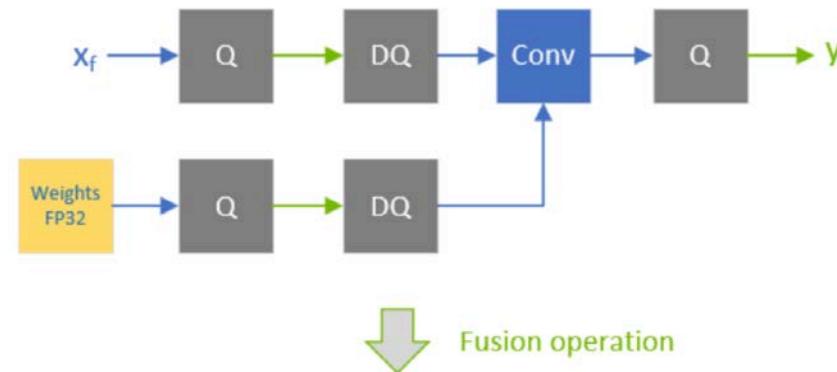


Fig.2 conv与Q/DQ的融合

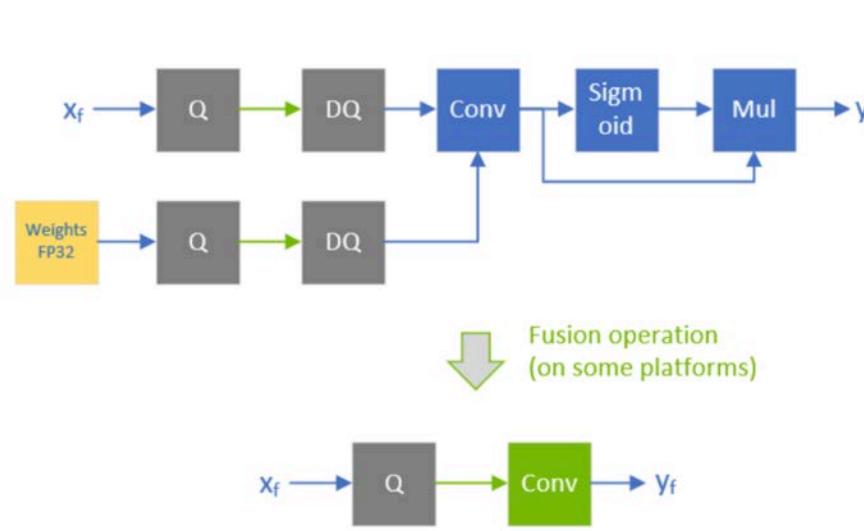


Fig.3 conv + SiLU + DQ的融合

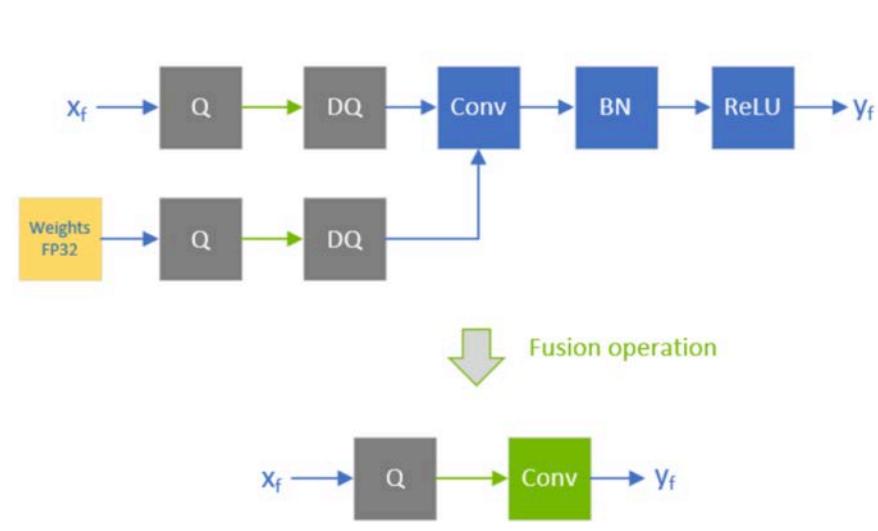


Fig.4 conv + BN + ReLU + DQ的融合

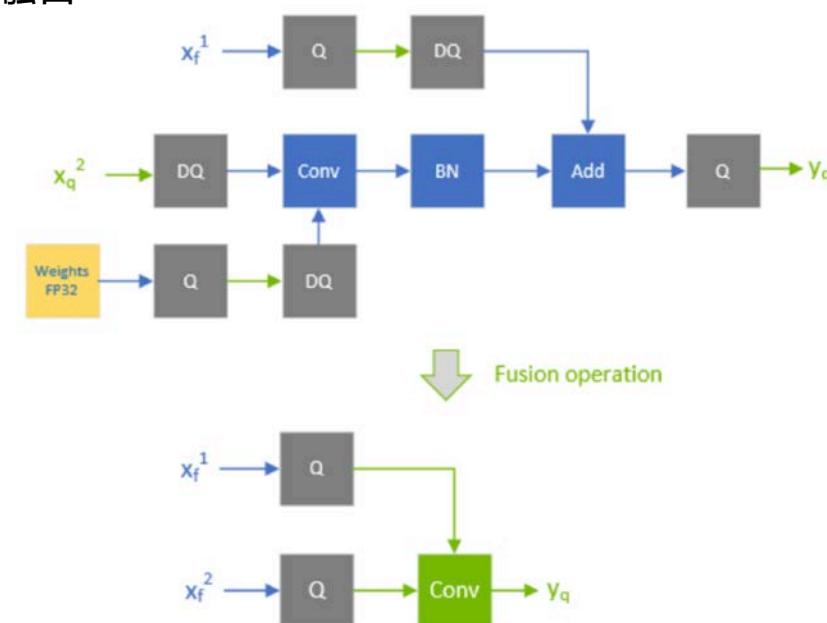


Fig.5 shortcut + DQ的融合

# TensorRT中QAT的Q/DQ Propagation技巧

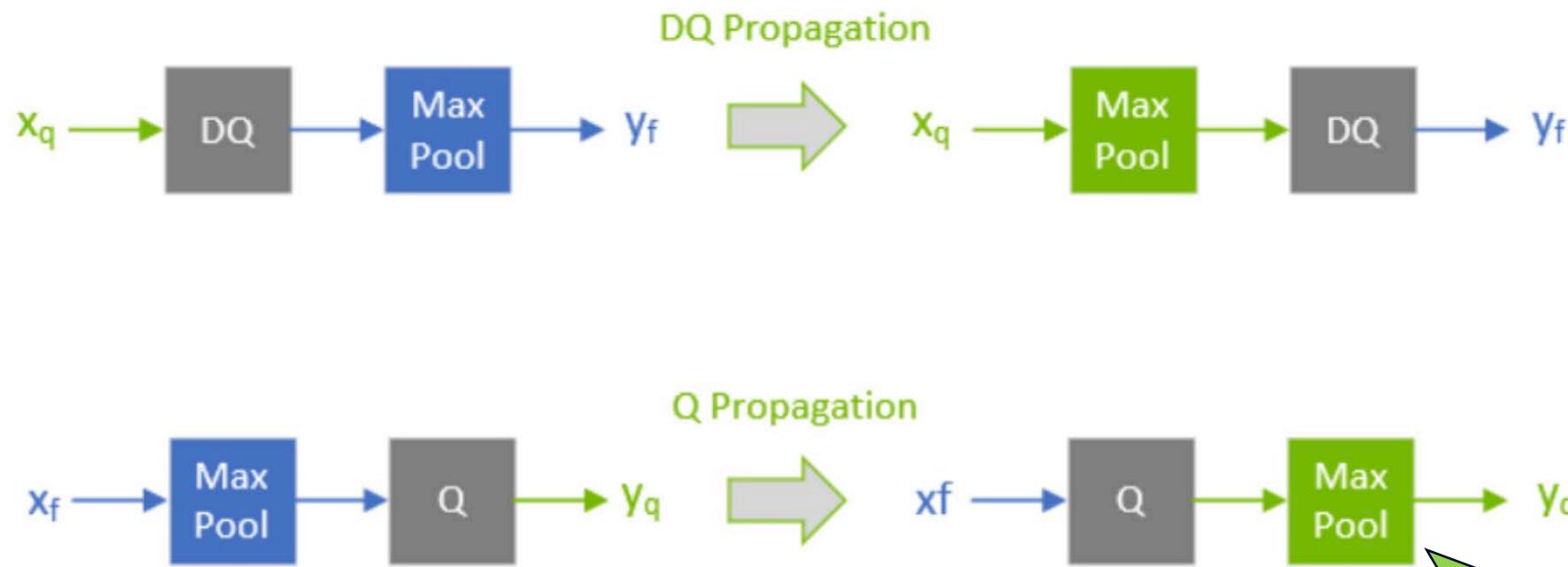


Fig.6 Max Pooling与Q/DQ的propagation  
 (由于maxpooling的结果在量化前后是没有变化，所以我们可以把fp32的maxpool节点转为int8的maxpool，从而达到加速)

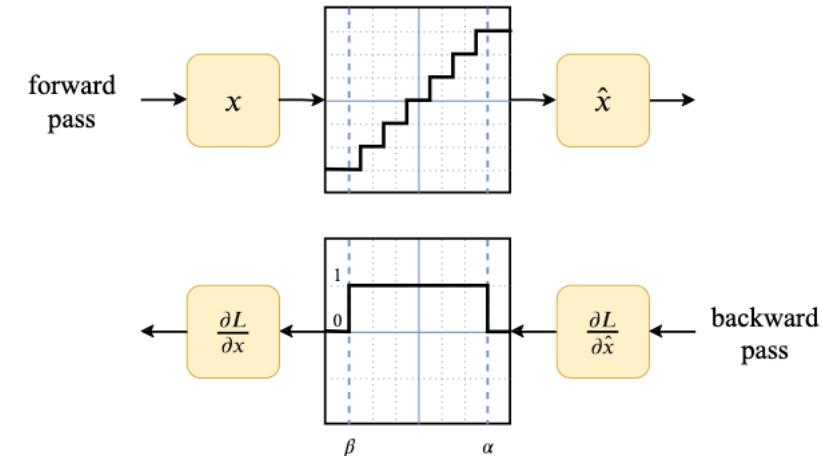
有的时候我们发现  
 TensorRT并没有帮我们做  
 到最好，这个时候我们  
 可以使用TensorRT API来  
 手动修改

# QAT的学习过程

- 主要是训练weight来学习误差
  - Q/DQ中的scale和zero-point也是可以训练的。通过训练来学习最好的scale来表示dynamic range
- 没有PTQ中那样人为的指定calibration过程
  - 不是因为没有calibration这个过程来做histogram的统计，而是因为QAT会利用fine-tuning的数据集在训练的过程中同时进行calibration，这个过程是我们看不见的。这就是为什么我们在pytorch创建QAT模型的时候需要选定calibration algorithm

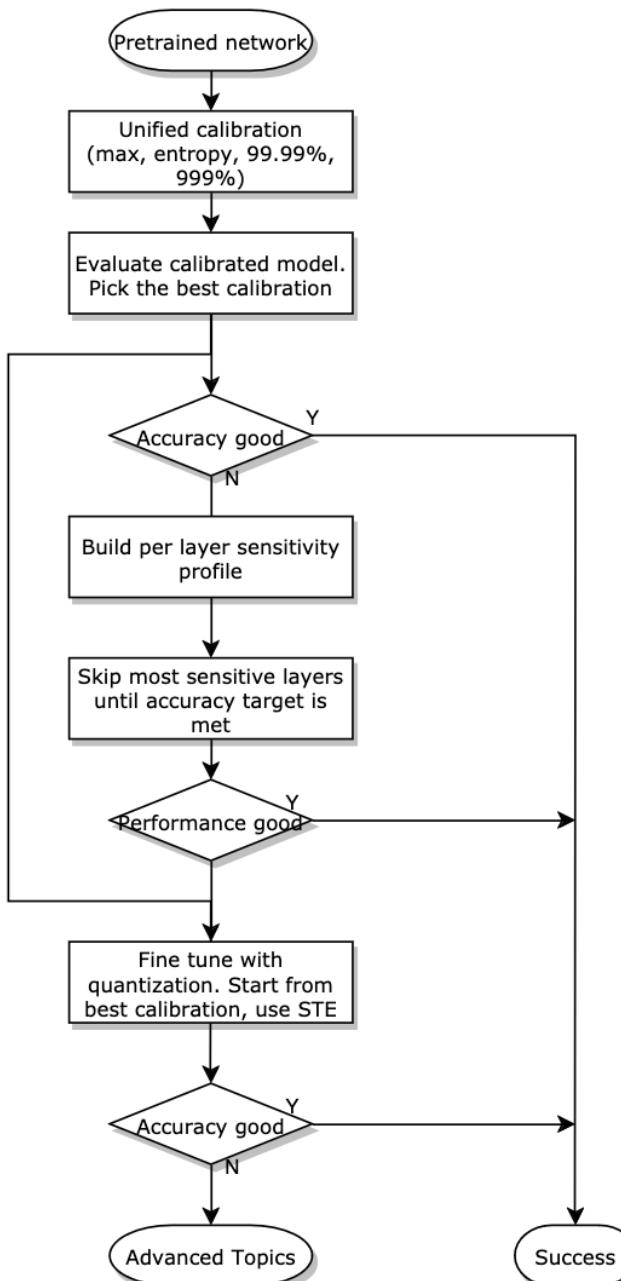
Models	fp32	Max	Entropy	99.9%	99.99%	99.999%	99.9999%
MobileNet v1	71.88	71.80	72.11	<b>72.07</b>	<b>72.14</b>	71.89	71.91
MobileNet v2	71.88	71.11	71.50	71.48	<b>71.56</b>	71.28	71.34
ResNet50 v1.5	76.16	76.68	<b>76.85</b>	76.59	76.67	76.77	76.81
ResNet152 v1.5	78.32	78.64	<b>78.61</b>	78.61	<b>78.69</b>	78.65	78.65
Inception v3	77.34	76.43	<b>78.43</b>	78.33	<b>78.49</b>	78.36	78.38
Inception v4	79.71	68.38	80.07	80.01	<b>80.14</b>	79.94	78.82
ResNeXt50	77.61	77.38	<b>77.67</b>	77.48	77.56	77.51	77.51
ResNeXt101	79.30	78.98	<b>78.99</b>	79.00	78.99	<b>79.01</b>	<b>79.04</b>
EfficientNet b0	76.85	76.16	<b>76.95</b>	76.85	<b>77.09</b>	76.98	76.63
EfficientNet b3	81.61	80.51	<b>80.63</b>	80.62	<b>81.07</b>	<b>81.09</b>	80.92
Faster R-CNN	36.95	36.62	<b>36.76</b>	36.31	36.76	<b>36.83</b>	36.76
Mask R-CNN	37.89	37.63	37.74	37.26	37.67	<b>37.76</b>	<b>37.75</b>
Retinanet	39.30	39.03	39.11	37.76	38.97	<b>39.25</b>	<b>39.20</b>
FCN	63.70	63.40	<b>64.10</b>	63.90	<b>64.20</b>	63.90	63.40
DeepLabV3	67.40	67.10	<b>67.30</b>	66.90	67.20	<b>67.50</b>	67.20
GNMT	24.27	<b>24.49</b>	<b>24.38</b>	24.35	24.41	24.48	24.35
Transformer	28.27	28.42	<b>28.46</b>	28.23	<b>28.21</b>	28.04	28.10
Jasper	96.09	<b>96.11</b>	<b>96.10</b>	95.23	95.94	96.01	96.08
BERT Large	91.01	90.29	90.14	89.97	90.50	<b>90.67</b>	90.60

使用不同的calibration algorithm进行QAT的精度比较。粗体表示使用PTQ中可以达到最好的calibration algorithm



对于activation value的scale进行学习的过程(上为forward，下为backward)

# 我们在部署过程中应该按照什么样的流程进行QAT



没有必要盲目的使用QAT，在使用QAT之前先看看PTQ是否已经达到了最佳。可以按照左边的图进行量化测试：

## 1. 先进行PTQ

1. 从多种calibration策略中选取最佳的算法
2. 查看是否精度满足，如果不满足再下一步。

## 2. 进行partial-quantization

1. 通过layer-wise的sensitive analysis分析每一层的精度损失
2. 尝试fp16 + int8的组合
3. fp16用在敏感层(网络入口和出口)，int8用在计算密集处(网络的中间)
4. 查看是否精度满足，如果不满足再下一步。
5. (注意，这里同时也需要查看计算效率是否得到满足)

## 3. 进行QAT来通过学习权重来适应误差

1. 选取PTQ实验中得到的最佳的calibration算法
2. 通过fine-tuning来训练权重(大概是原本训练的10%个epoch)
3. 查看是否精度满足，如果不满足查看模型设计是否有问题
4. (注意，这里同时也需要查看层融合是否被适用，以及Tensor core是否被用)

普遍来讲，量化后精度下降控制在相对精度损失 $\leq 2\%$ 是最好的。



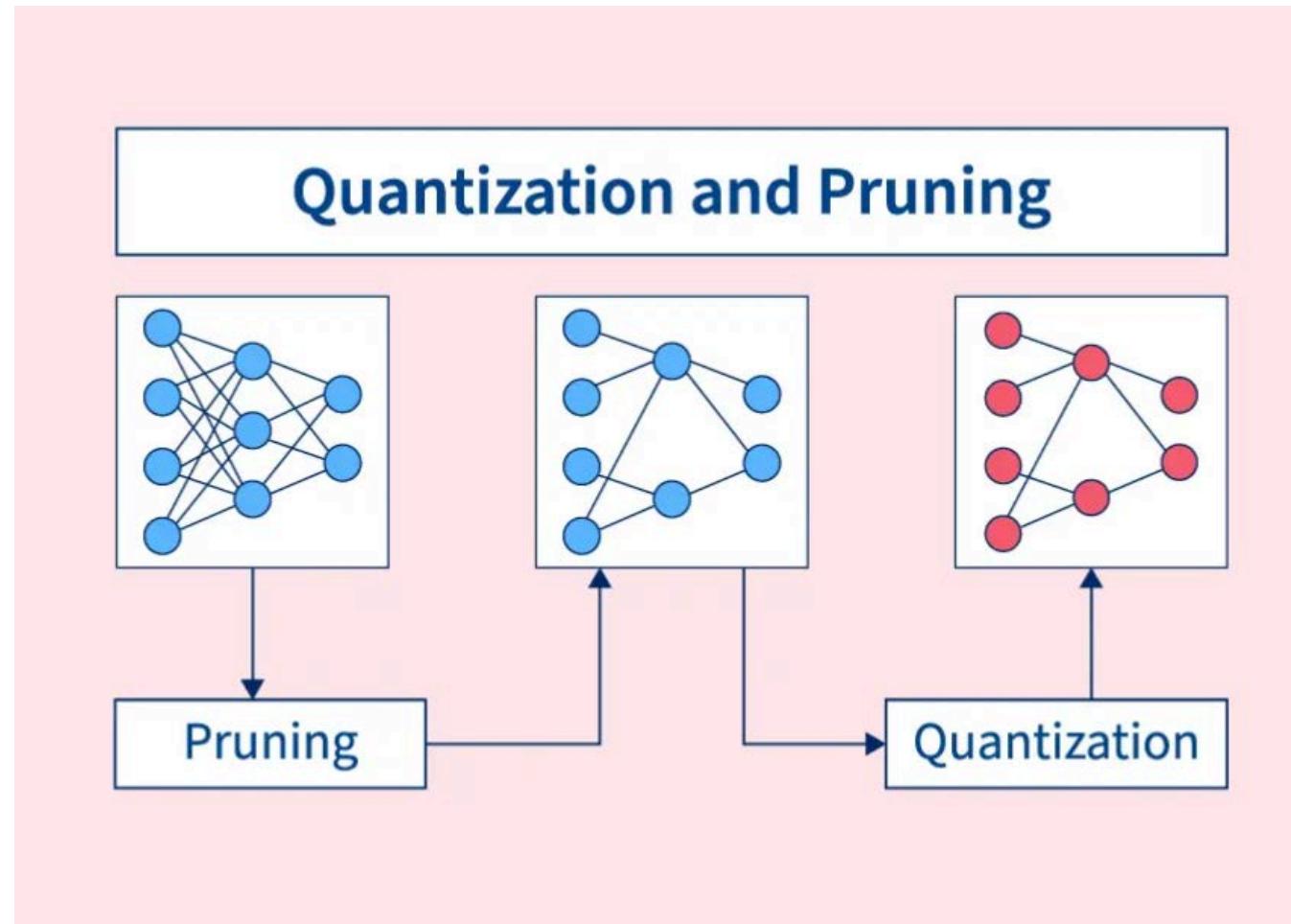
04

# Pruning (overview-and-pruning-granularity)

Goal: 理解什么是模型剪枝，模型剪枝的分类，以及各类剪枝的利弊都有哪些

# 模型剪枝(Pruning)

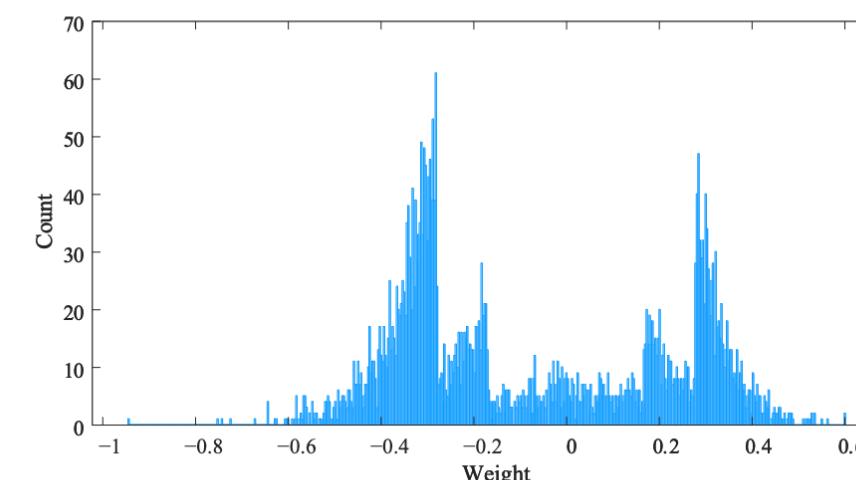
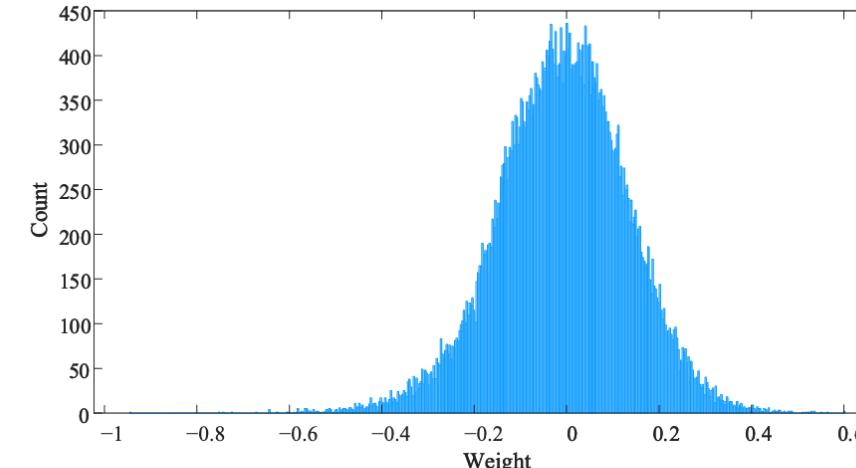
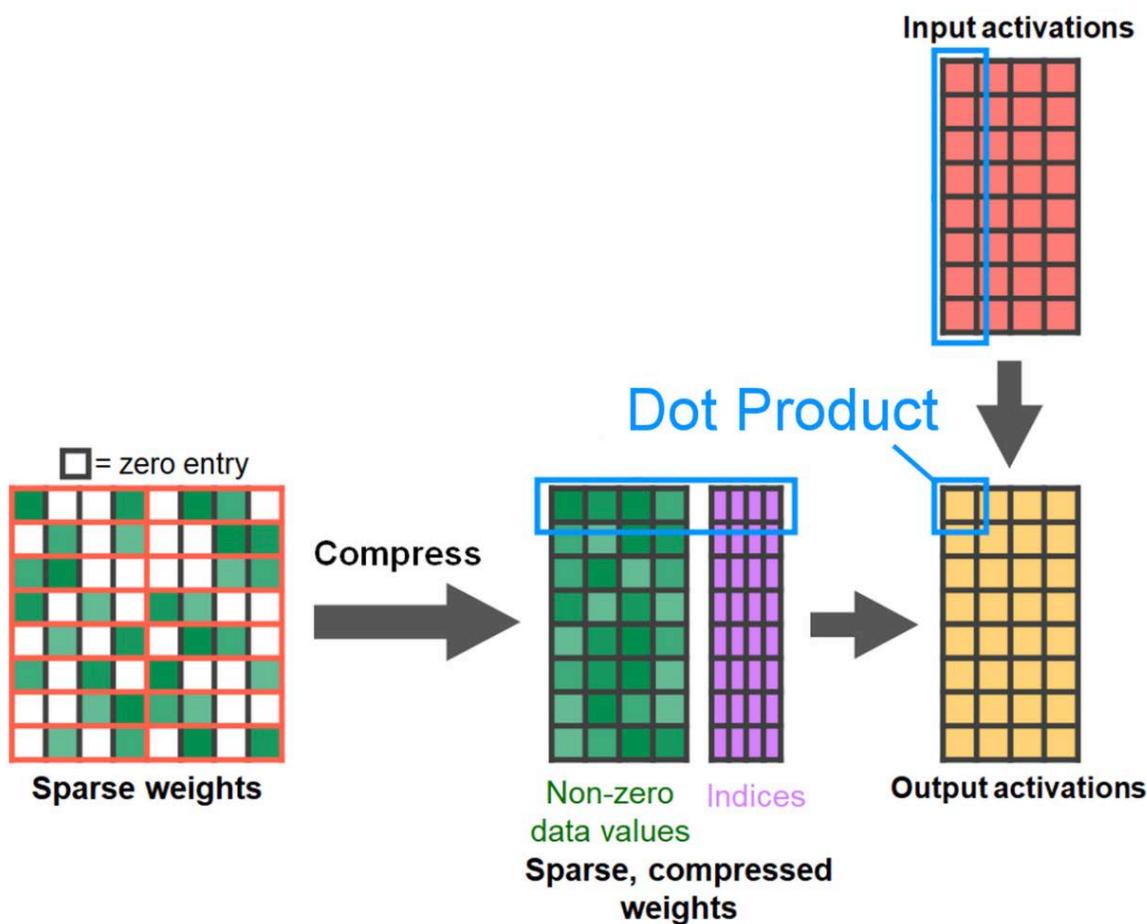
模型剪枝是不同于量化的另外一种模型压缩的方式。如果说“量化”是通过改变权重和激活值的表现形式从而让内存占用变小和计算变快的话，“剪枝”则是直接“删除”掉模型中没有意义的，或者意义较小的权重，来让推理计算量减少的过程。



更准确来说，是skip掉一些不必要的计算

# 模型剪枝(Pruning) – More background...

为什么我们需要剪枝？主要是因为学习的过程中会产生过参数化导致会产生一些意义并不是很大的权重，或者值为0的权重(ReLU)。对于这些权重所参与的计算是占用计算资源且没有作用的。我们需要想办法**找到这些权重**并让硬件去skip掉这些权重所参与的计算



[1] [Exploiting NVIDIA Ampere Structured Sparsity with cuSPARSELT](#)

[2] [Differential Evolution Based Layer-Wise Weight Pruning for Compressing Deep Neural Networks](#)

## 模型剪枝(Pruning) – More background...

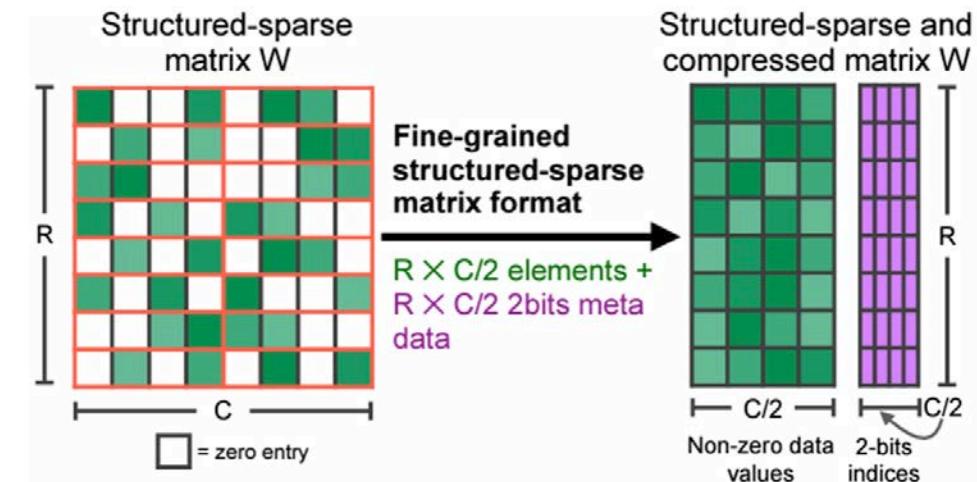
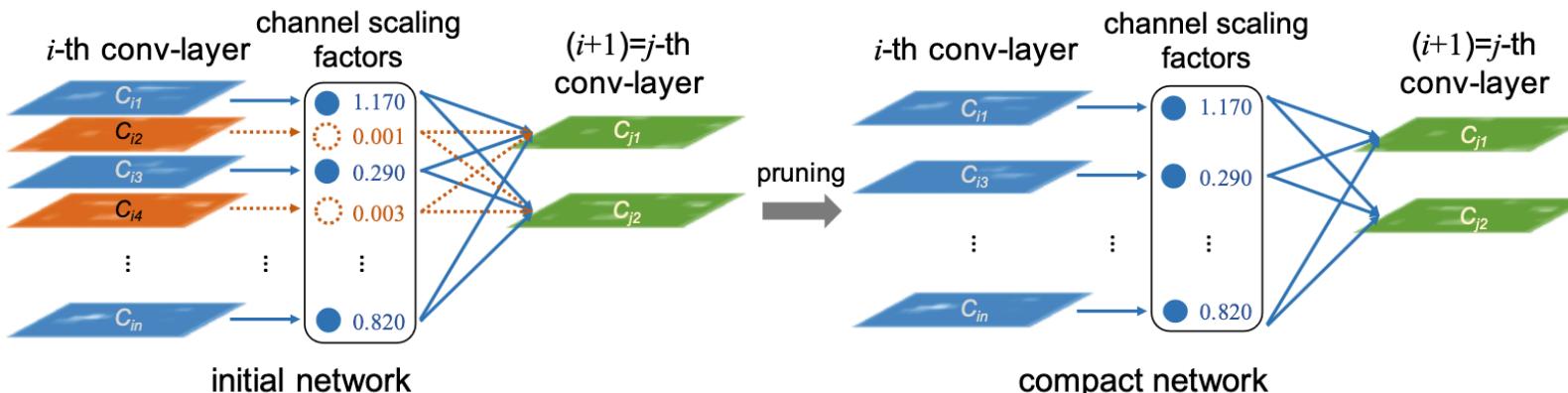
模型剪枝的过程其实大体上分为这么几个流程：

1. 获取一个已经训练好的初始模型
2. 对这个模型进行剪枝
3. 对剪枝后的模型进行fine-tuning
4. 获取到一个压缩的模型

# 模型剪枝(Pruning) – More background...

模型剪枝的过程其实大体上分为这么几个流程：

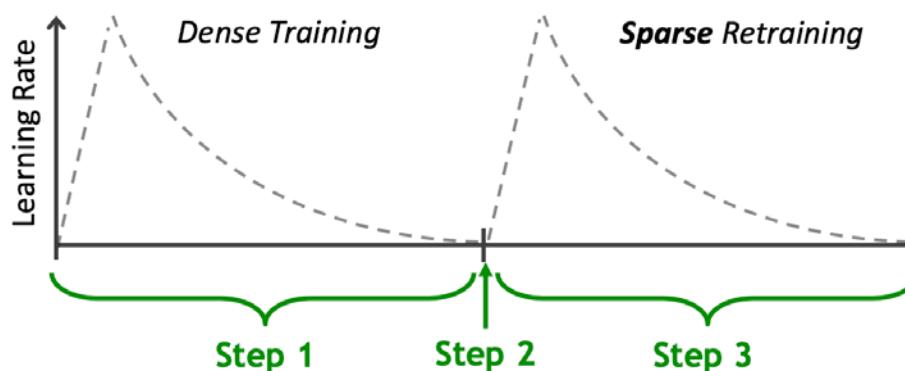
1. 获取一个已经训练好的初始模型
2. 对这个模型进行剪枝
  - 我们可以通过训练的方式让DNN去学习哪些权重是可以归零的
    - (e.g. 使用L1 regularization和BN中的scaling factor让权重归零)
  - 我们也可以通过自定义一些规则，手动的有规律的去让某些权重归零
    - (e.g. 对一个 $1 \times 4$ 的vector进行2:4的weight pruning)
3. 对剪枝后的模型进行fine-tuning
4. 获取到一个压缩的模型



# 模型剪枝(Pruning) – More background...

模型剪枝的过程其实大体上分为这么几个流程：

1. 获取一个已经训练好的初始模型
2. 对这个模型进行剪枝
  - 我们可以通过训练的方式让DNN去学习哪些权重是可以归零的
    - (e.g. 使用L1 regularization和BN中的scaling factor让权重归零)
  - 我们也可以通过自定义一些规则，手动的有规律的去让某些权重归零
    - (e.g. 对一个1x4的vector进行2:4的weight pruning)
3. 对剪枝后的模型进行fine-tuning
  - 有很大的可能性，在剪枝后初期的网络的精度掉点比较严重
  - 需要fine-tuning这个过程来恢复精度
  - Fine-tuning后的模型有可能会比之前的精度还要上涨
4. 获取到一个压缩的模型



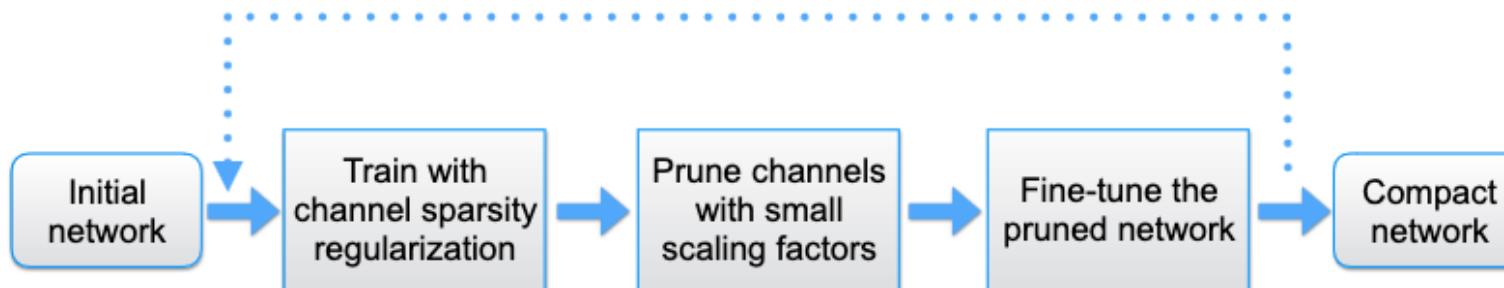
ImageNet

Network	Accuracy		
	Dense FP16	Sparse FP16	Sparse INT8
ResNet-34	73.7	73.9	0.2
ResNet-50	76.6	76.8	0.2
ResNet-101	77.7	78.0	0.3
ResNeXt-50-32x4d	77.6	77.7	0.1
ResNeXt-101-32x16d	79.7	79.9	0.2
DenseNet-121	75.5	75.3	-0.2
DenseNet-161	78.8	78.8	-
Wide ResNet-50	78.5	78.6	0.1
Wide ResNet-101	78.9	79.2	0.3
Inception v3	77.1	77.1	-
Xception	79.2	79.2	-
VGG-16	74.0	74.1	0.1
VGG-19	75.0	75.0	-

# 模型剪枝(Pruning) – More background...

模型剪枝的过程其实大体上分为这么几个流程：

1. 获取一个已经训练好的初始模型
2. 对这个模型进行剪枝
  - 我们可以通过训练的方式让DNN去学习哪些权重是可以归零的
    - (e.g. 使用L1 regularization和BN中的scaling factor让权重归零)
  - 我们也可以通过自定义一些规则，手动的有规律的去让某些权重归零
    - (e.g. 对一个 $1 \times 4$ 的vector进行2:4的weight pruning)
3. 对剪枝后的模型进行fine-tuning
  - 有很大的可能性，在剪枝后初期的网络的精度掉点比较严重
  - 需要fine-tuning这个过程来恢复精度
  - Fine-tuning后的模型有可能会比之前的精度还要上涨
4. 获取到一个压缩的模型
  - **其实如果到这个阶段对模型压缩还不够满足的话，可以回到step2循环**



# 模型剪枝(Pruning) – More background...

模型剪枝是可以配合着量化一起进行的。

FP32

0.587	2.56	0.487	-1.194	0.348	3.037	-3.087	-3.748
1.894	3.964	3.333	-3.984	3.771	3.112	1.596	-3.22
-0.782	0.995	-2.856	-3.018	-0.665	0.455	3.531	-0.726
2.754	0.277	2.062	-0.003	2.803	0.958	2.893	-2.146
3.763	3.149	3.539	1.138	0.917	-2.179	-0.112	2.458
1.374	0.494	0.34	3.148	2.742	-1.552	1.049	1.442
-2.908	-0.928	-1.436	-1.069	1.677	3.201	0.273	-2.022
3.68	1.596	3.999	-2.239	-1.112	1.919	3.972	-1.469

Dense Matrix

FP32

0.0	2.56	0.0	-1.194	0.0	0.0	-3.087	-3.748
0.0	3.964	0.0	-3.984	3.771	0.0	0.0	-3.22
0.0	0.0	-2.856	-3.018	0.0	0.0	3.531	-0.726
2.754	0.0	2.062	0.0	2.803	0.0	2.893	0.0
3.763	0.0	3.539	0.0	0.0	-2.179	0.0	2.458
1.374	0.0	0.0	3.148	2.742	-1.552	0.0	0.0
-2.908	0.0	-1.436	0.0	0.0	3.201	0.0	-2.022
3.68	0.0	3.999	0.0	0.0	1.919	3.972	0.0

Sparse Matrix

INT8

0	81	0	-38	0	0	-98	-119
0	126	0	-127	120	0	0	-102
0	0	-91	-96	0	0	112	-23
87	0	65	0	89	0	92	0
120	0	112	0	0	-69	0	78
44	0	0	100	87	-49	0	0
-92	0	-46	0	0	102	0	-64
117	0	127	0	0	61	126	0

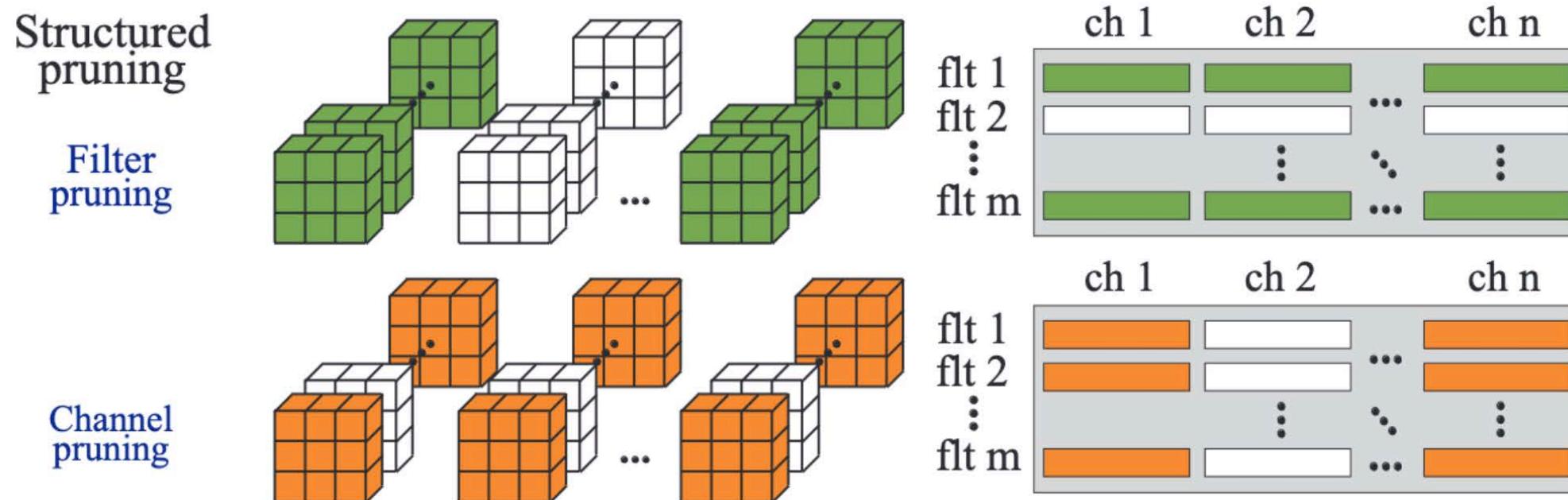
Sparse-Quantized Matrix

## 模型剪枝的分类

模型减枝可以按照减枝的方法按照一定规律与否可以分为结构化减枝，以及非结构化减枝。同时，模型减枝也可以按照减枝的粒度与强度分为粗粒度减枝，以及细粒度减枝。

# 模型剪枝的分类

- Coarse Grain Pruning (粗粒度剪枝)
  - 这里面包括Channel/Kernel Pruning
  - Channel/Kernel Pruning是**结构化减枝(Structured pruning)**
    - 这个是比较常见的，也就是直接把某些卷积核给去除掉。
    - 比较常见的方法就是通过L1 Norm寻找权重中影响度比较低的卷积核。

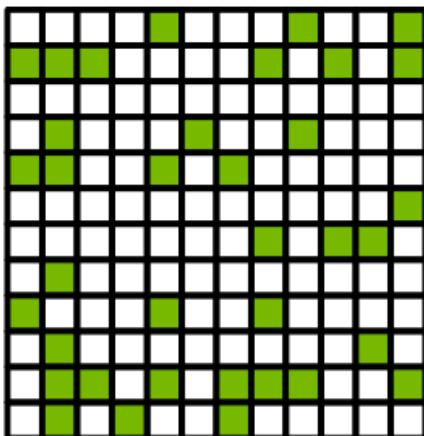


## 模型剪枝的分类

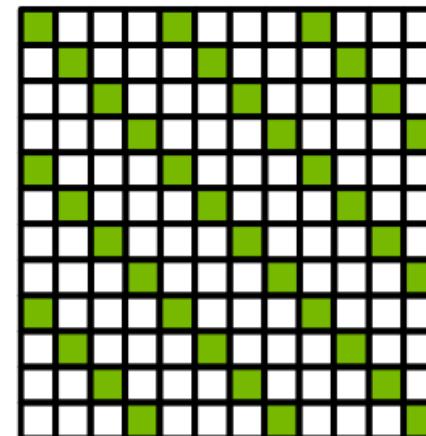
- 我们来看一下Coarse Grain Pruning的优势劣势
  - 优势：
    - 不依赖于硬件，可以在任何硬件上跑并且得到性能的提升
  - 劣势：
    - 由于减枝的粒度比较大(卷积核级别的)，所以有潜在的掉精度的风险
    - 不同DNN的层的影响程度是不一样的
    - 减枝之后有可能反而不适合硬件加速(比如Tensor Core的使用条件是channel是8或者16的倍数)

# 模型剪枝的分类

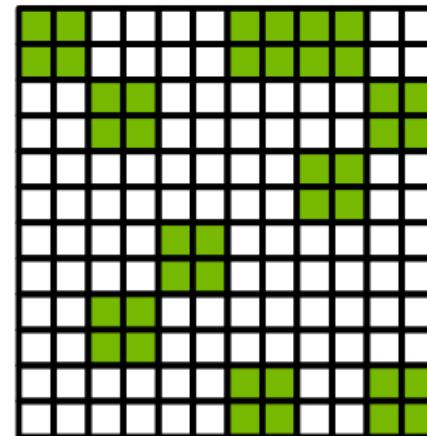
- Fine Grain Pruning(细粒度剪枝)
  - 主要是对权重的各个元素本身进行分析减枝
  - 这里面可以分为**结构化减枝(structed)**与**非结构化减枝(unstructured)**
    - 结构化减枝
      - Vector-wise的减枝: 将权重按照 $4 \times 1$ 的vector进行分组, 每四个中减枝两个的方式减枝权重
      - Block-wise的减枝: 将权重按照 $2 \times 2$ 的block进行分区, block之间进行比较的方式来减枝block
    - 非结构化减枝
      - Element-wise的减枝: 每一个每一个减枝进行分析, 看是不是影响度比较高



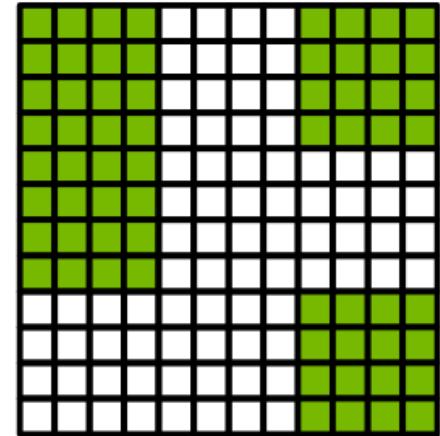
非结构化剪枝



非结构化剪枝



非结构化剪枝



非结构化剪枝

## 模型剪枝的分类

- 当然Fine Grain Pruning也有优势和劣势
  - 优势：
    - 相比于Coarse Grain Pruning，精度的影响并不是很大
  - 劣势：
    - 需要特殊的硬件的支持(Tensor Core可以支持sparse)
    - 需要用额外的memory来存储哪些index是可以保留计算的
    - memory的访问不是很效率(跳着访问)
    - 支持sparse计算的硬件内部会做一些针对sparse的tensor的重编，这个会比较耗时



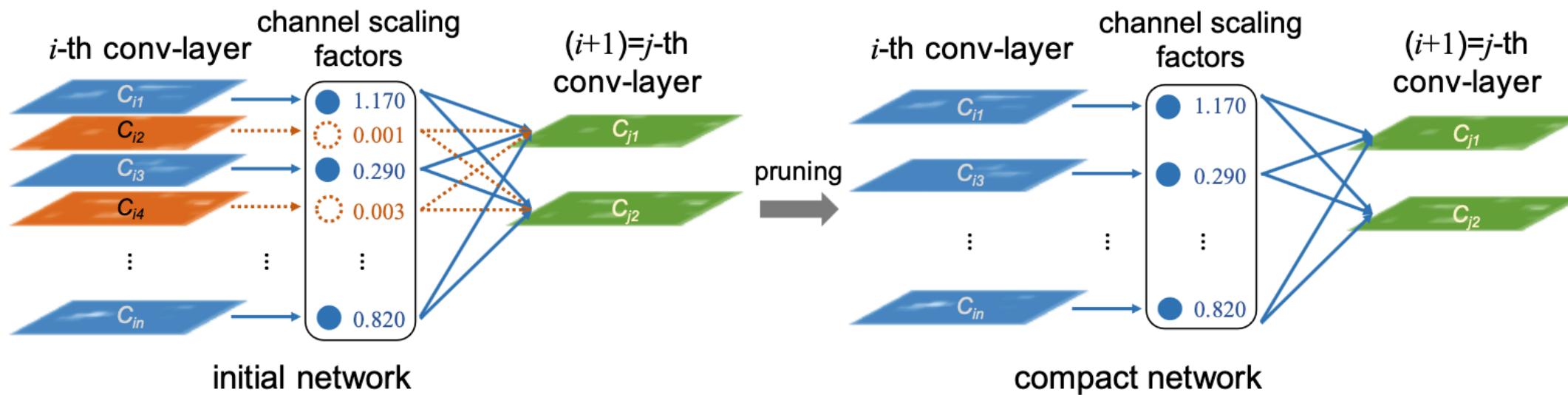
04

# Pruning (channel-level pruning)

Goal: 理解channel-level pruning的算法，以及如何使用L1-Norm来让权重稀疏

# channel-level pruning

结构化剪枝中比较常用以及使用起来比较简单的方式是channel-level pruning，不依赖于硬件的特性可以简单的实现粗粒度的剪枝。这里围绕着一篇关注度比较高的文章进行讲解。



整篇文章的核心点是围绕着通过使用BN中的scaling factor，与使用L1-regularization的训练可以让权重趋向零这一特点，找到conv中不是很重要的channel，实现channel-level的pruning。

(思考)L1 normalization为什么可以让权重趋向零？

## (快速复习)L1 & L2 regularization

两者都是通过在loss损失函数中添加L1/L2范数(L1/L2-norm)，实现对权重学习的惩罚(penalty)来限制权重的更新方式。根据L1/L2范数的不同，两者的作用也是不同的

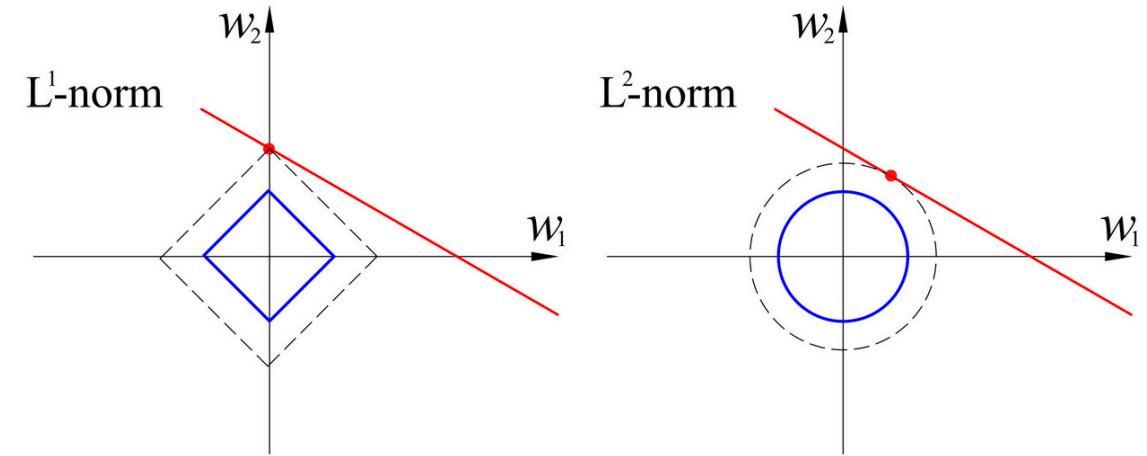
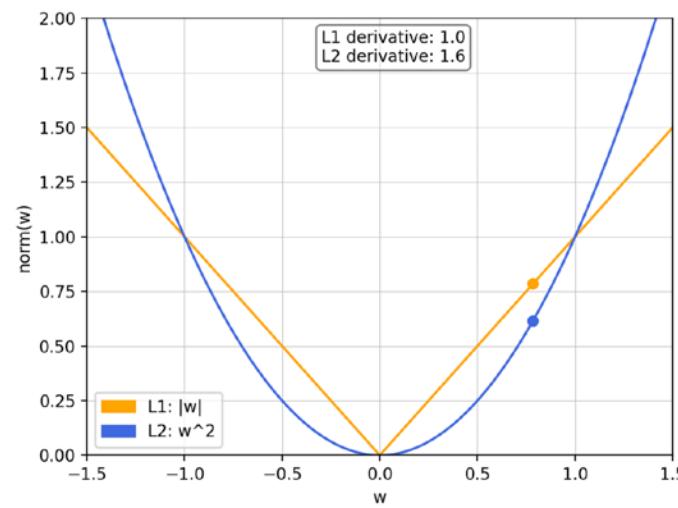
- L1 regularization: 可以用来**稀疏参数**，或者说让参数趋向零。Loss function的公式是：

$$\min f(x) + \lambda \sum_{i=1}^n |w_i|$$

- L2 regularization: 可以用来**减少参数值的大小**。Loss function的公式是：

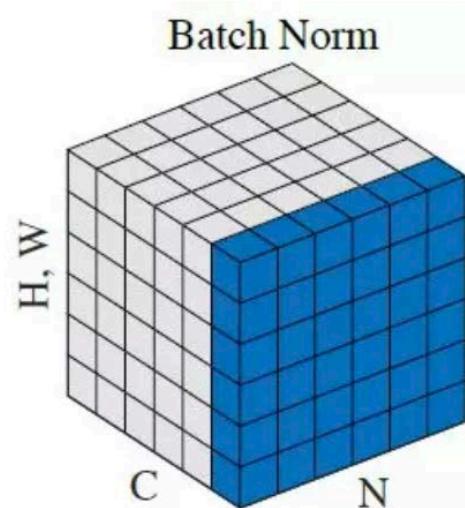
$$\min f(x) + \frac{\lambda}{2} \sum_{i=1}^n |w_i|^2$$

训练的目的是让loss function逐渐变小，所以我们可以看看这两个L1/L2-norm在back-propagation中的梯度变化：



## (快速复习)BN中的scaling factor

我们都知道Batch normalization一般放在conv之后，对conv的输出进行normalization。整个计算是channel-wise的，所以每一个channel都会有自己的BN参数(均值、方差、缩放因子、偏移因子)。那么我们可以考虑，如果BN之后发现某一个channel的scaling非常小，或者为零，是不是可以认为这个channel做参与的计算并没有非常大强度的改变/提取特征？是不是并不是那么重要？



$$\mu_B = \frac{1}{B} \sum_{i=1}^B x_i$$

$$\sigma_B^2 = \frac{1}{B} \sum_{i=1}^B (x_i - \mu_B)^2 + \epsilon$$

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$y_i = \gamma * \hat{x}_i + \beta$$

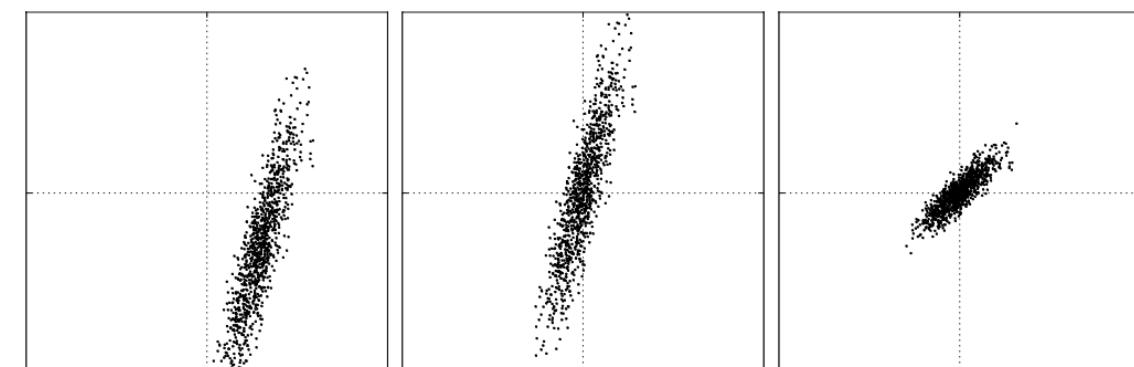
$\mu_B$ : 代表均值

$\sigma_B^2$ : 代表方差

$\epsilon$ : 一个大于0的浮点数，用于防止分母为0

$\gamma$ : scaling, 缩放因子

$\beta$ : shift, 偏移因子

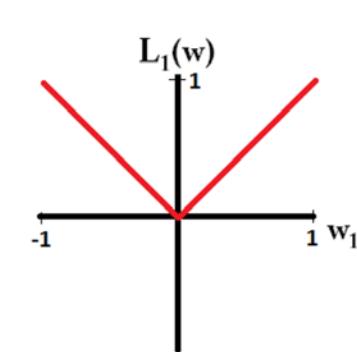


(a) Original data

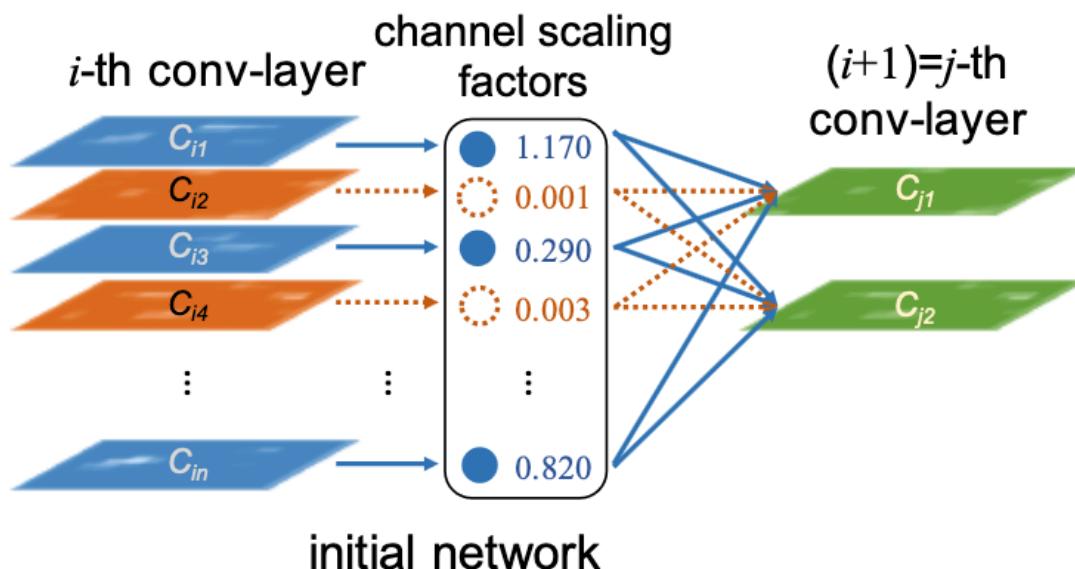
(b) Zero centered data (c) Standardized data

## 使用BN和L1-norm对模型的权重进行计算以及重要度排序

$$L = \sum_{(x,y)} l(f(x, W), y) + \lambda \sum_{\gamma \in \Gamma} g(\gamma) \quad g(s) = |s|$$

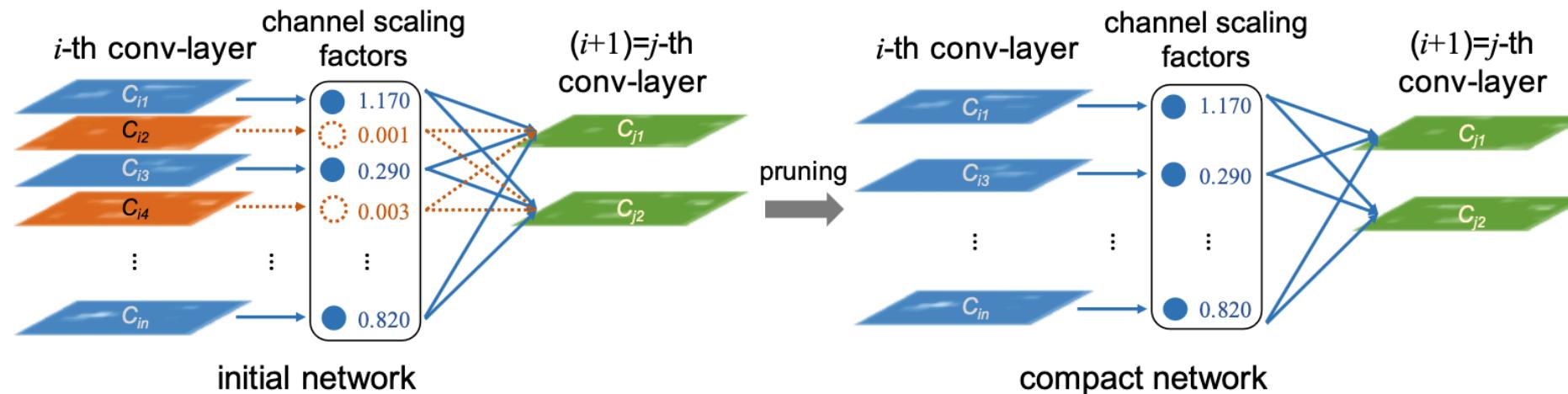


在channel-wise pruning中，同样使用L1-norm作为惩罚项添加到loss中，但是L1-norm的参数不再是每一个权重，而是BN中对于conv中每一个channel的scaling factor。从而在学习过程中让scaling factor趋向零，并最终变为零。(负的scaling factor会变大，正的scaling factor会变小)



通过对scaling factor进行L1正则，这里面的 $C_{i2}$ 和 $C_{i4}$ 会逐渐趋向零，我们可以认为这些channel不是很重要，可以称为pruning的候选

# Pruning和fine-tuning



对于scaling factor不是很大的channel，在pruning的时候可以把这些channel直接剪枝掉，但同时也需要把这些channel所对应的input/output的计算也skip掉。最终得到一个紧凑版的网络。这个方法比较方便去选择剪枝的力度，通过不断的实验找到最好的剪枝百分比

- 0% pruning
- 25% pruning
- 50% pruning
- 75% pruning

但是需要注意：

1. 刚剪枝完的网络，由于权重信息很多信息都没了，所以需要fine-tuning来提高精度(需要使用mask)
2. 剪枝完的channel size可能会让计算密度变低(64ch通过75% pruning后变成16ch)

# Pruning和fine-tuning

0.0123	0.2374	0.2987
0.9871	0.1291	0.5431
0.1354	0.2222	0.1567
0.2764	0.0174	0.0000
0.0000	0.1001	0.5431
0.0012	0.0002	0.0007
0.0012	0.0074	0.2017
0.0371	0.0000	0.0123
0.0345	0.1874	0.1023
02984	0.8751	0.9823
0.7412	0.1200	0.0131
0.4456	0.2987	0.7891

dence weight

 *channel-level  
pruning*

0.0123	0.2374	0.2987
0.9871	0.1291	0.5431
0.1354	0.2222	0.1567
0.000	0.0000	0.0000
0.0000	0.0000	0.0000
0.0000	0.0000	0.0000
02984	0.8751	0.9823
0.7412	0.1200	0.0131
0.4456	0.2987	0.7891

sparse weight

1	1	1
1	1	1
1	1	1
0	0	0
0	0	0
0	0	0
1	1	1
1	1	1
1	1	1

weight mask

# Pruning和fine-tuning

0.0123	0.2374	0.2987
0.9871	0.1291	0.5431
0.1354	0.2222	0.1567
0.000	0.0000	0.0000
0.0000	0.0000	0.0000
0.0000	0.0000	0.0000

*sparse weight*

0	0	0
0	0	0
0	0	0

0	0	0
0	0	0
0	0	0

*weight mask*

1	1	1
1	1	1
1	1	1

0.0000	0.0000	0.0000
0.0000	0.0000	0.0000
0.0000	0.0000	0.0000

*fine-tuning  
(update weight)*

0.0221	0.3375	0.3989
0.1872	0.3491	0.7432
0.2354	0.2342	0.3569
0.0012	0.0043	0.0091
0.0045	0.0013	0.0091
0.0098	0.0001	0.0001

*dence weight*



# Pruning和fine-tuning

0.0123	0.2374	0.2987
0.9871	0.1291	0.5431
0.1354	0.2222	0.1567
0.000	0.0000	0.0000
0.0000	0.0000	0.0000
0.0000	0.0000	0.0000
0.2984	0.8751	0.9823
0.7412	0.1200	0.0131
0.4456	0.2987	0.7891

sparse weight

1	1	1
1	1	1
1	1	1
0	0	0
0	0	0
0	0	0

weight mask

update



*fine-tuning  
(update weight)*

0.0221	0.3375	0.3989
0.1872	0.3491	0.7432
0.2354	0.2342	0.3569
0.0012	0.0043	0.0091
0.0045	0.0013	0.0091
0.0098	0.0001	0.0001
0.0030	0.0012	0.0001
0.0123	0.0081	0.0003
0.0035	0.034	0.0040
0.2424	0.9752	0.7824
0.7813	0.2391	0.0221
0.9126	0.0027	0.9111

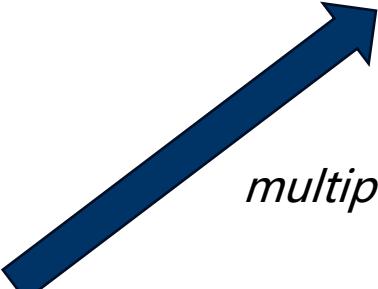
dense weight

*fine-tuning  
(update weight)*

0.0221	0.3375	0.3989
0.1872	0.3491	0.7432
0.2354	0.2342	0.3569
0.0000	0.0000	0.0000
0.2424	0.9752	0.7824
0.7813	0.2391	0.0221
0.9126	0.0027	0.9111
0.0000	0.0000	0.0000
0.0000	0.0000	0.0000
0.2424	0.9752	0.7824
0.7813	0.2391	0.0221
0.9126	0.0027	0.9111

sparse weight

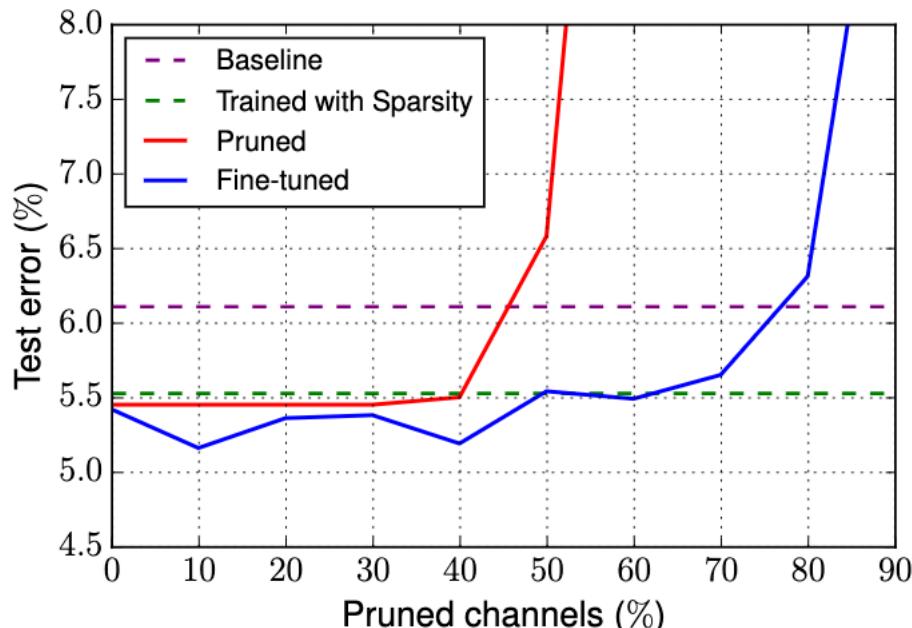
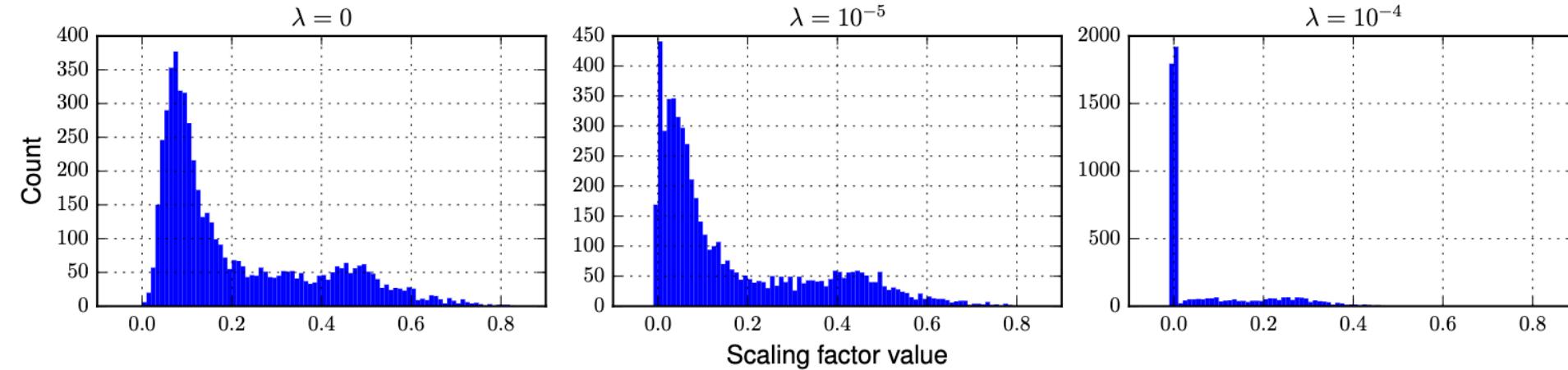
*multiply weight with mask*



整个fine-tuning的过程是通过sparse计算时得到的各个channel的mask来决定weight的更新方式。最终得到的weight依然是sparse，但已经通过调整过了

# channel-level pruning中的超参和技巧

论文中提到，整个pruning的过程中 $\lambda$ 和channel的剪枝力度是超参，需要不断的实验找到最优。 $\lambda$ 表示的是在loss中L1-norm这个penalty所占的比重。 $\lambda$ 越大就整个模型就会越趋近稀疏



同时，不同力度的channel pruning也会伴随着精度损失的不同。论文中虽然没有提及，但有一些pruning的经验：

- pruning后的channel尽量控制在64的倍数
  - 要记住最大化tensor core的使用
- 对哪些层可以大力度的pruning需要进行sensitive analysis
  - 要记住DNN中哪些层是敏感层



04

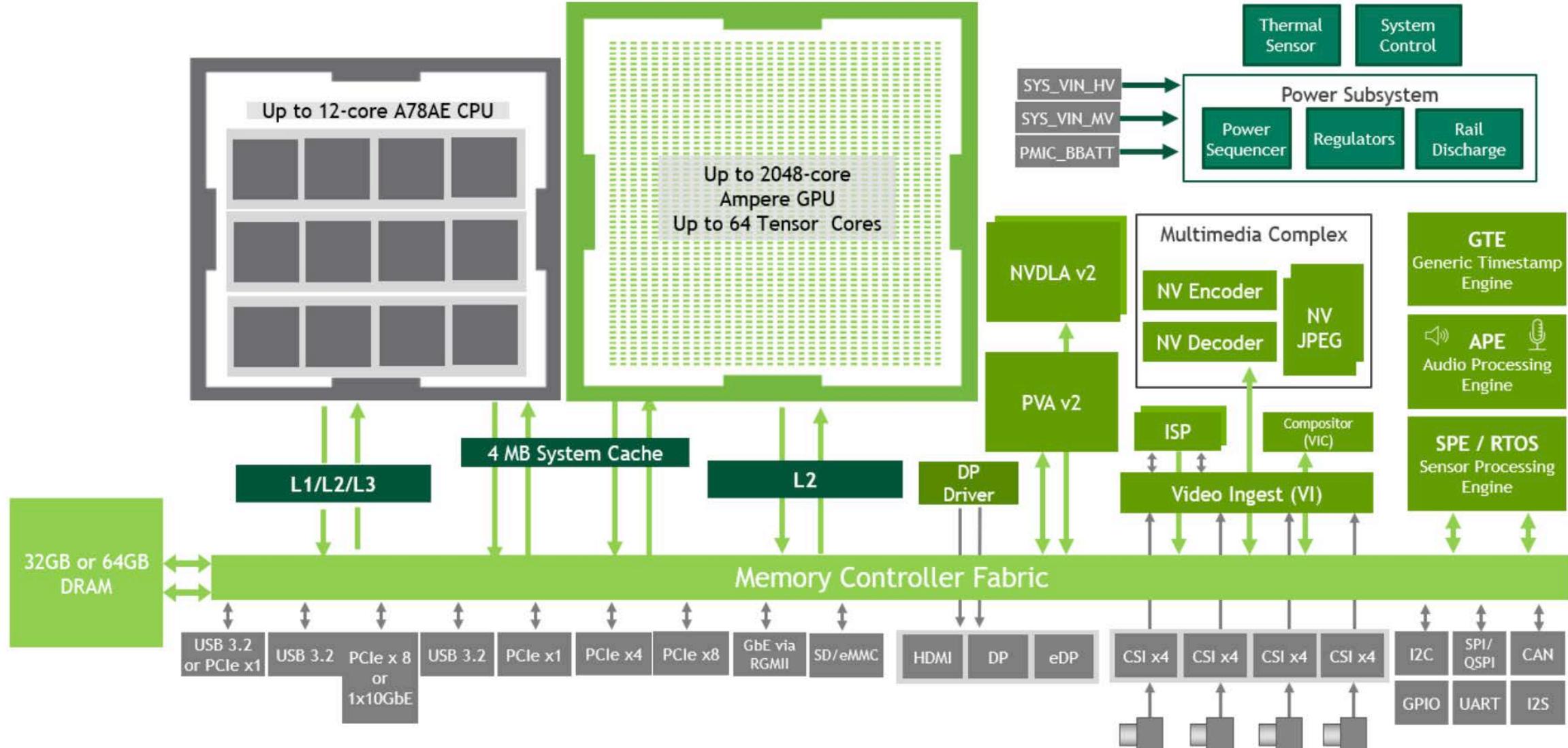
# Pruning (sparse-tensor-core)

Goal: 理解NVIDIA是如何使用sparse tensor core来处理带有稀疏性的矩阵乘法

# (快速回顾)自动驾驶中需要关注的电力消耗

NVIDIA Jetson AGX Orin

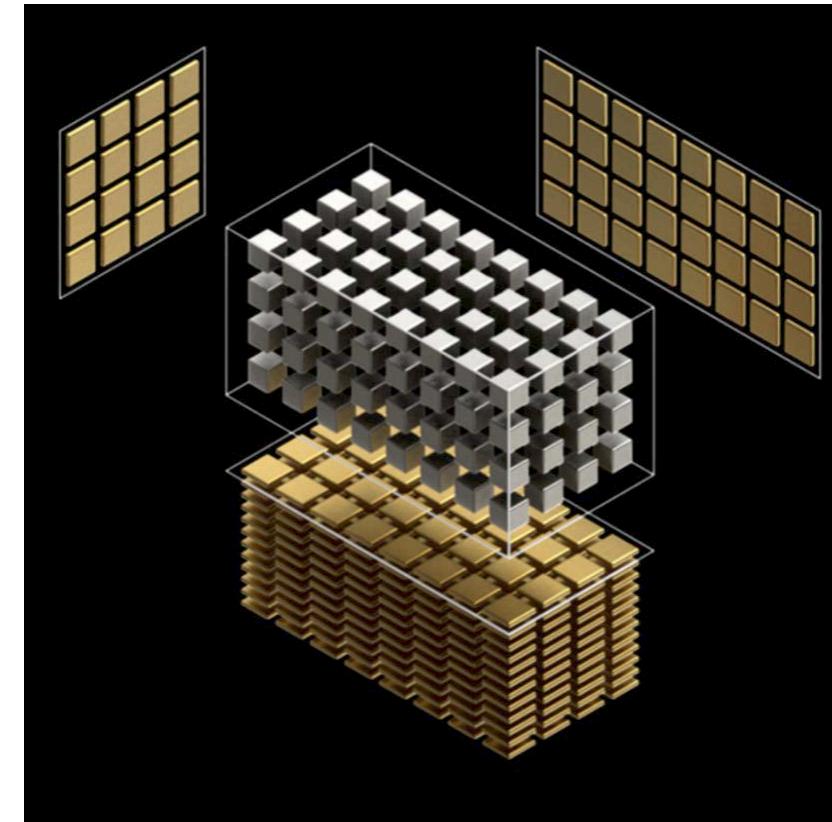
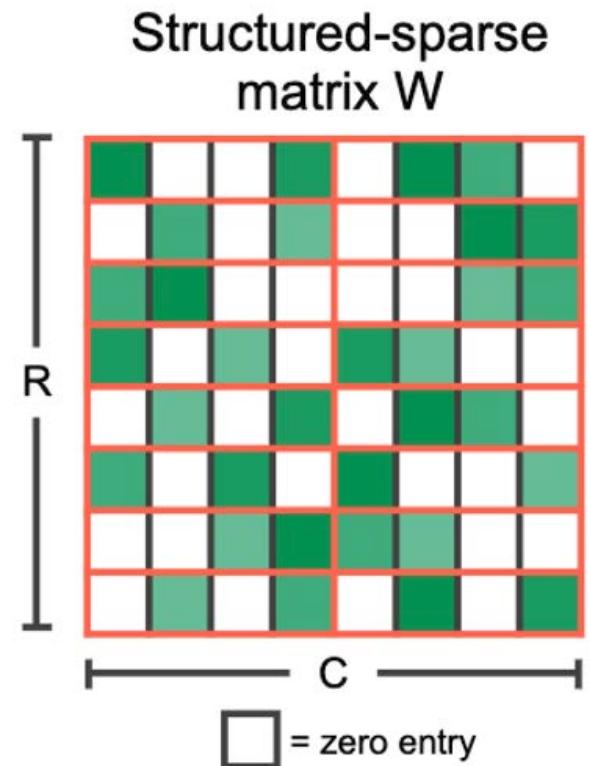
Peak performance  
GPU: 170 TOPS (Sparse INT8), 85TOPS (DENSE INT8)  
DLA0: 52.5 TOPS (Sparse INT8), 26.25TOPS (DENSE INT8)  
DLA1: 52.5 TOPS (Sparse INT8), 26.25TOPS (DENSE INT8)  
Power consumption ~60W



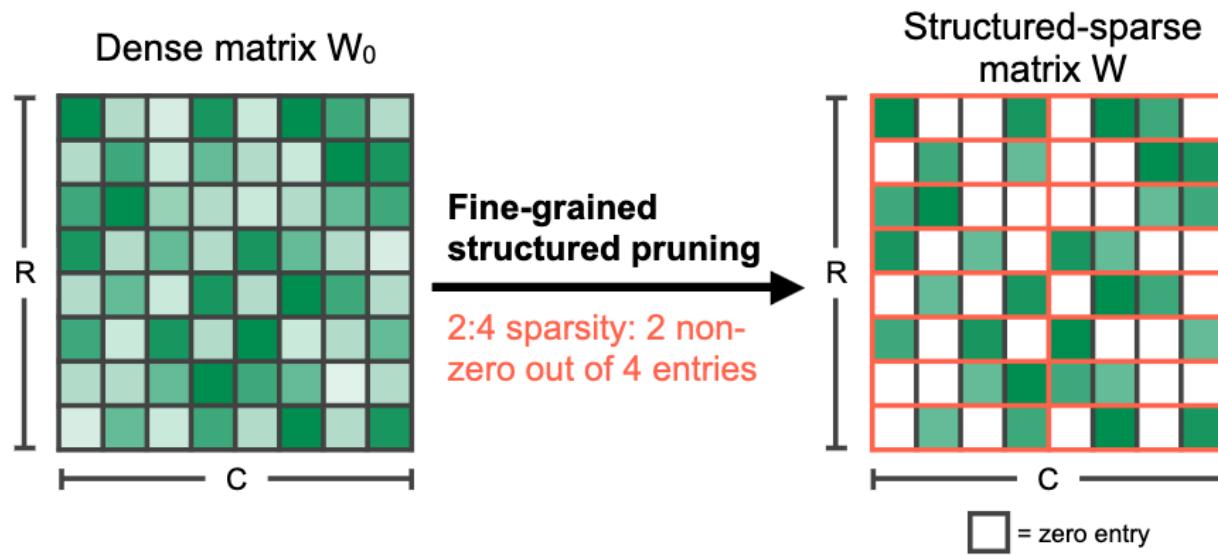
# Ampere架构中的3rd Generation Tensor core

在Ampere架构(e.g. A100, Jetson AGX Orin)中的第三代Tensor core支持带有sparsity的matrix计算。更准确来说：

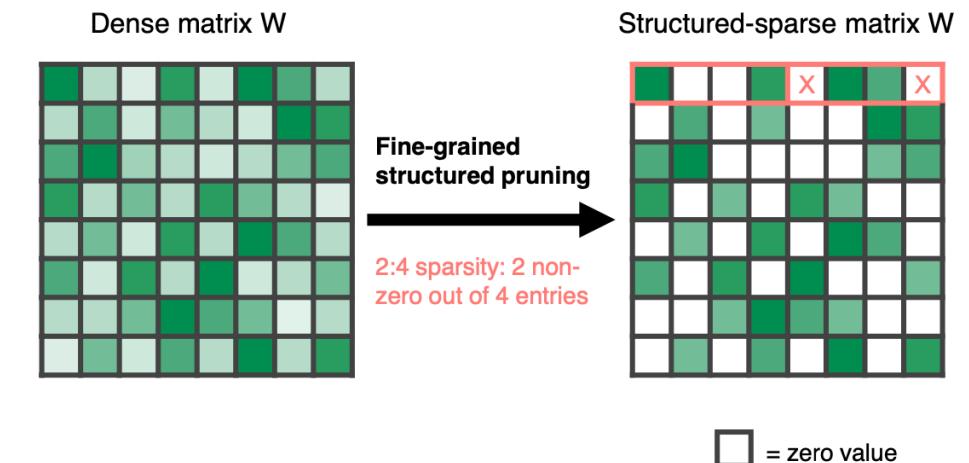
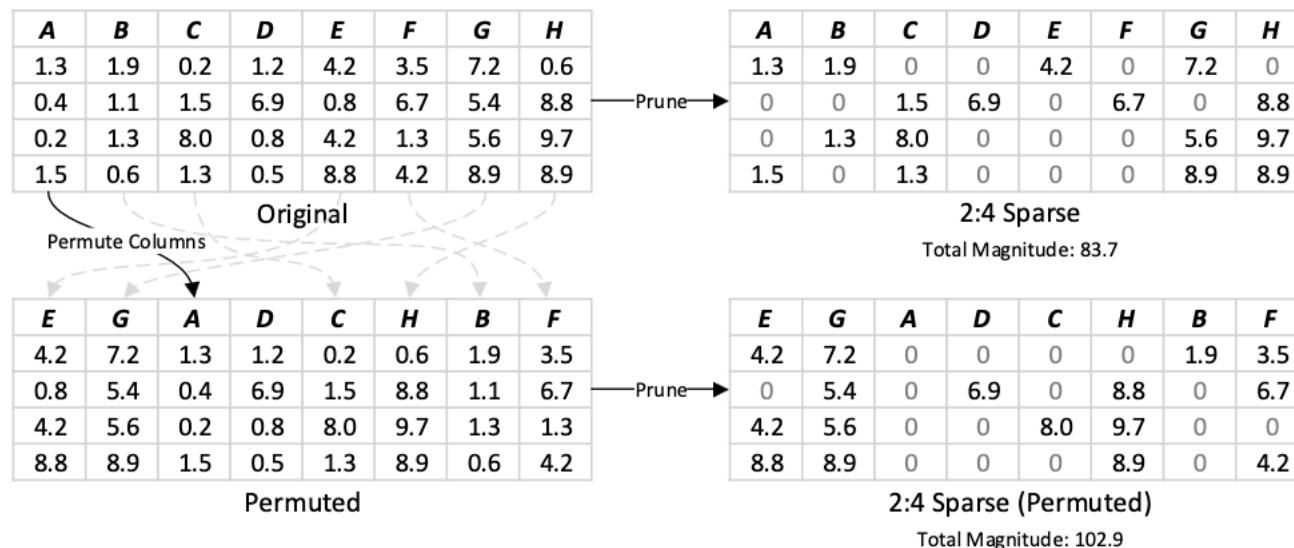
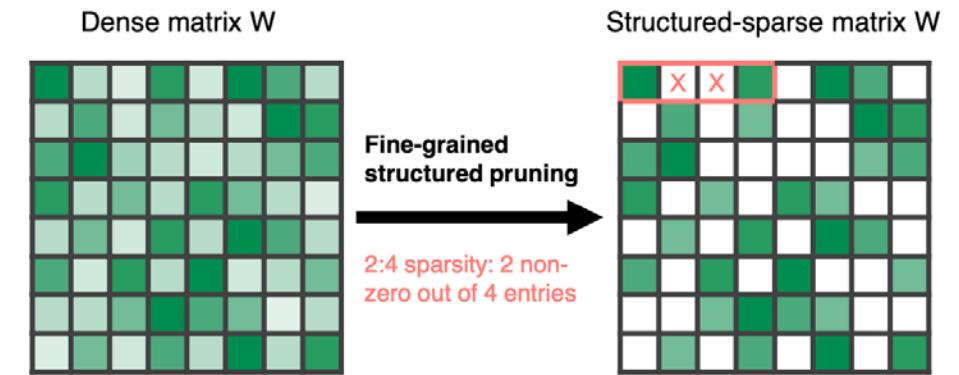
- 支持Fine-grained structured sparsity
- “structured”表现在，sparsity的pattern是以 $1 \times 4$  vector的大小进行2:4的归零(vector-wise pruning)
- 50%粒度的sparse pruning，理论上可以实现2x的吞吐量的提升



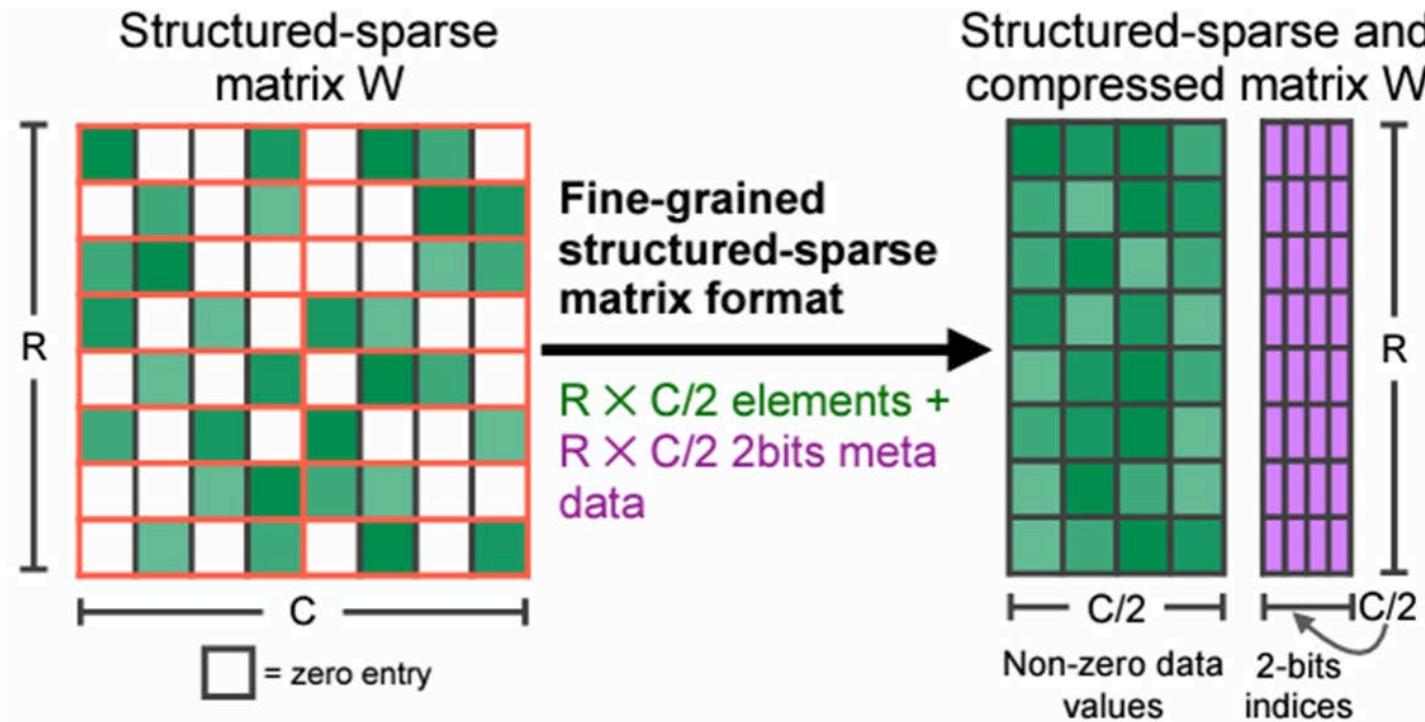
# Ampere架构中的3rd Generation Tensor core



一个dense的matrix会以2:4 sparsity的方式进行剪枝(每4个连续的权重中最小的两个归零)



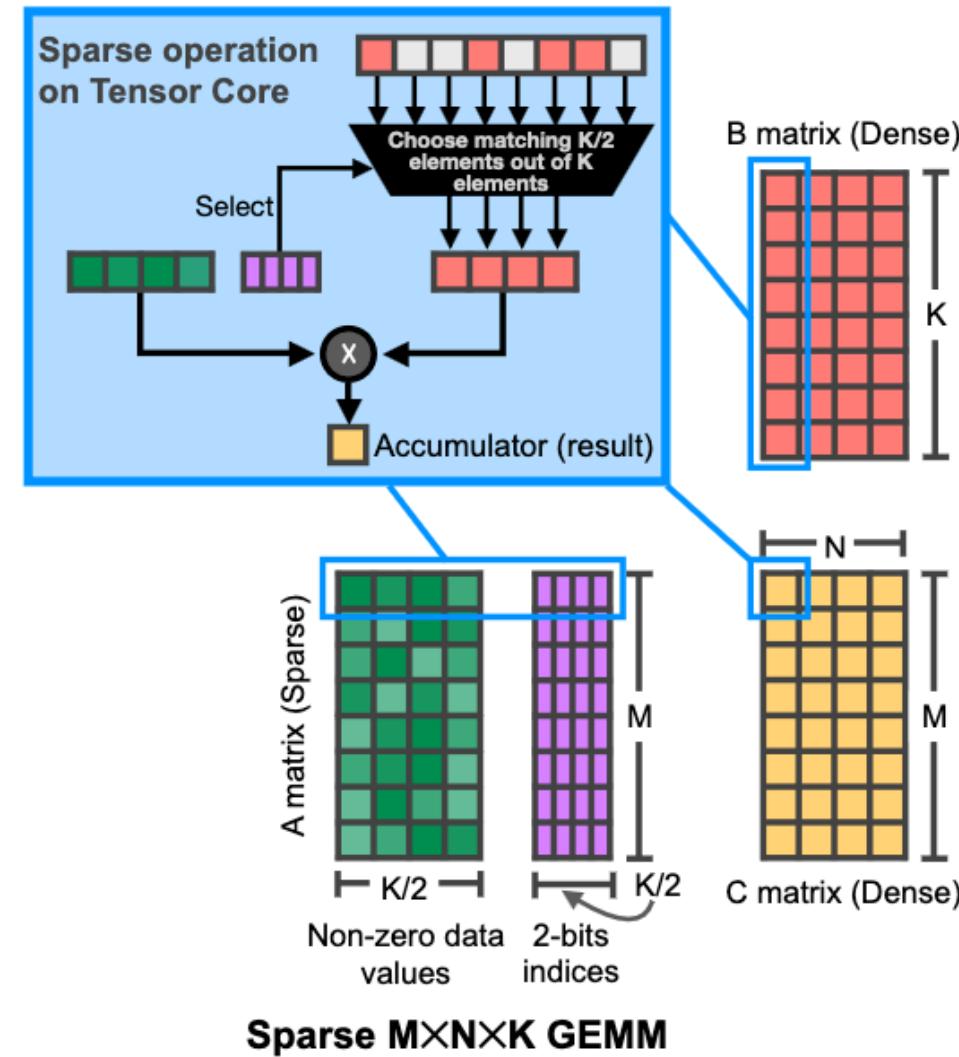
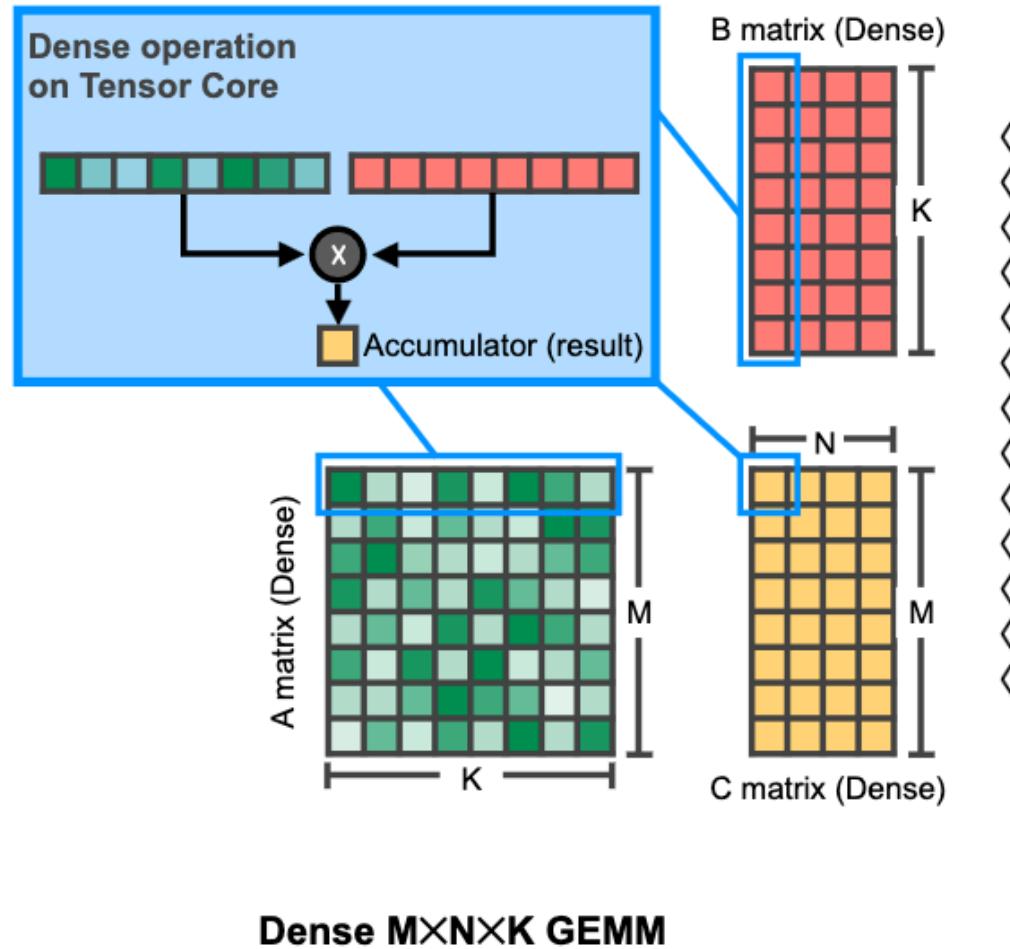
# Ampere架构中的3rd Generation Tensor core



对于已经sparse pruning过的matrix，可以进行压缩。在memory中只保存非零的weight，至于哪些weight是零，哪些是非零用一个2-bits indices来保存(可以把它理解为一种索引)

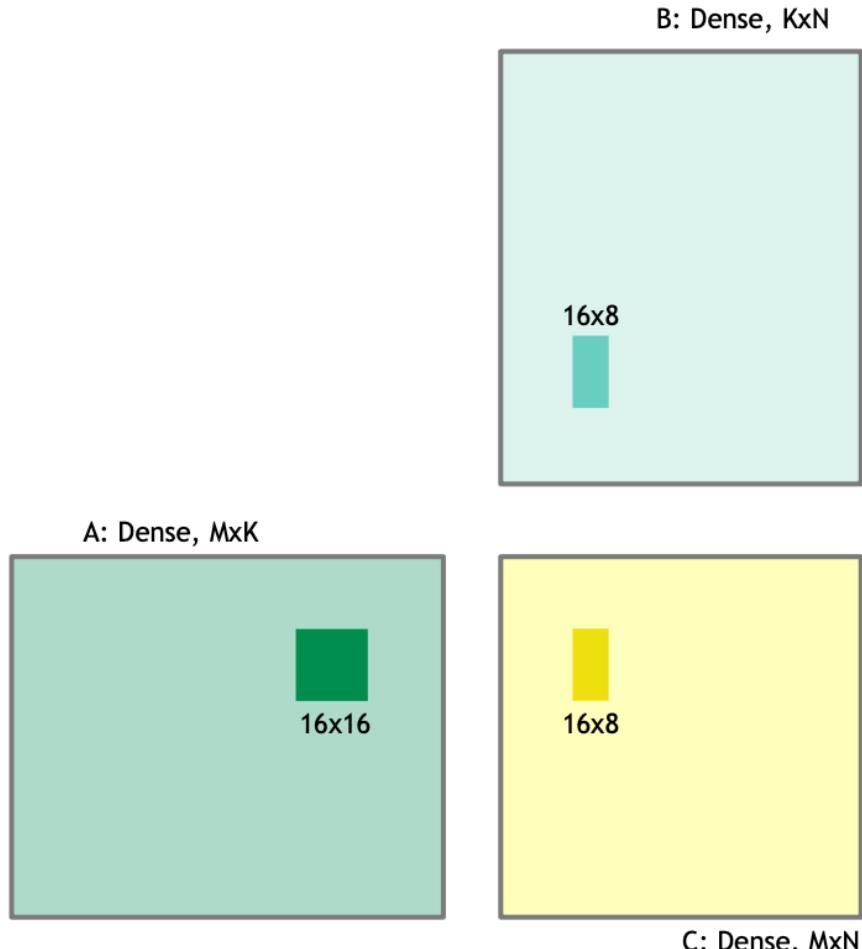
# Ampere架构中的3rd Generation Tensor core

这样一来，weight的大小减半。同时对于activation values，可以通过2-bits indices来决定activation values中哪些值是参与计算的，哪些是skip掉的(这个过程需要特殊的硬件unit来实现)，从而实现2x的计算吞吐量的提升。



# Sparse tensor core做矩阵乘法

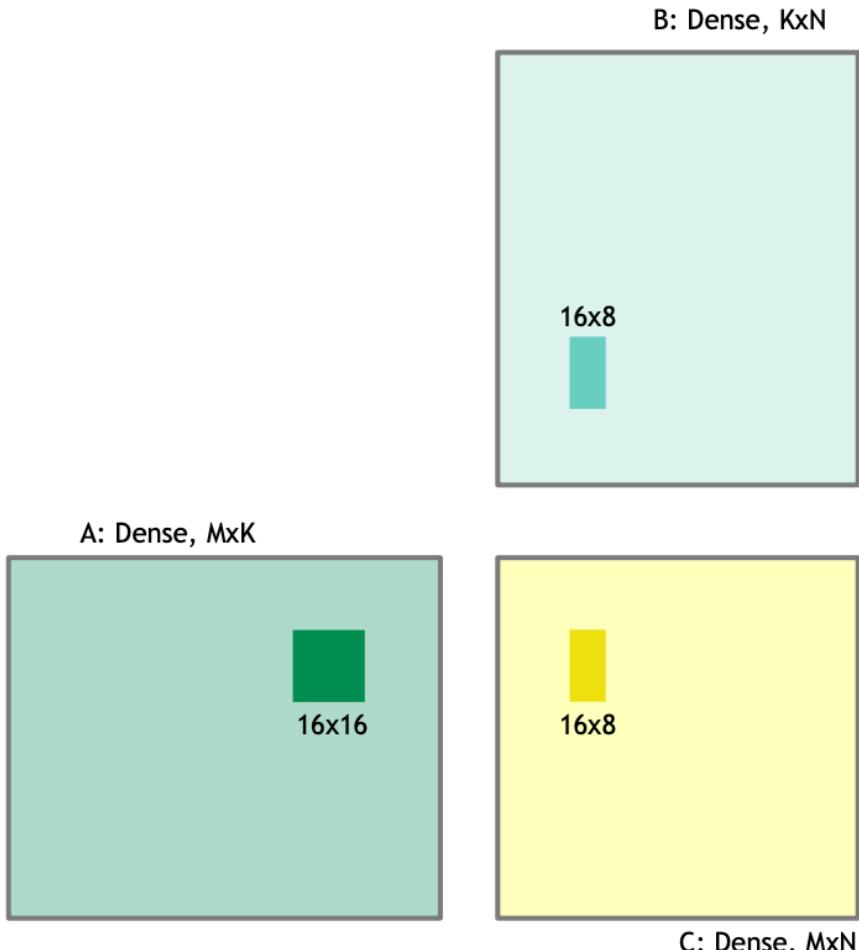
Dence Tensor core(FP16)的计算  $A(M, K) * B(K, N) = C(M, N)$  的过程



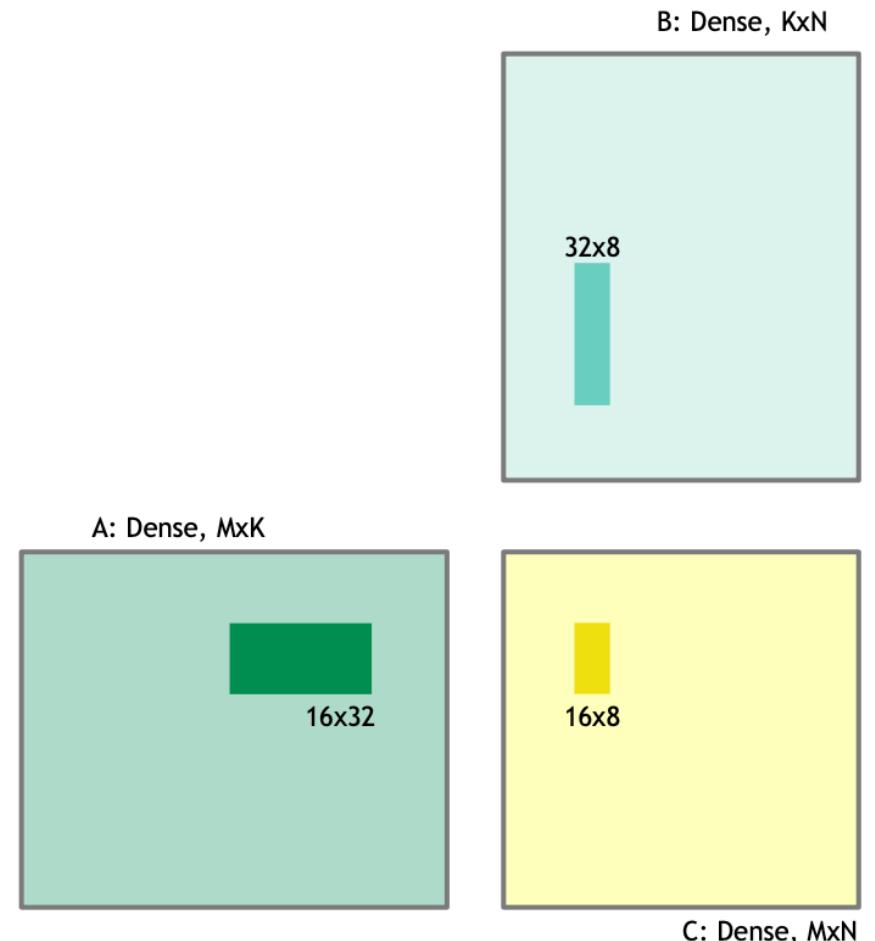
可以用1 cycle完成一个 $16 \times 16 * 16 \times 8 = 16 * 8$ 的矩阵乘法

# Sparse tensor core做矩阵乘法

Dence Tensor core(FP16)的计算  $A(M, K) * B(K, N) = C(M, N)$  的过程



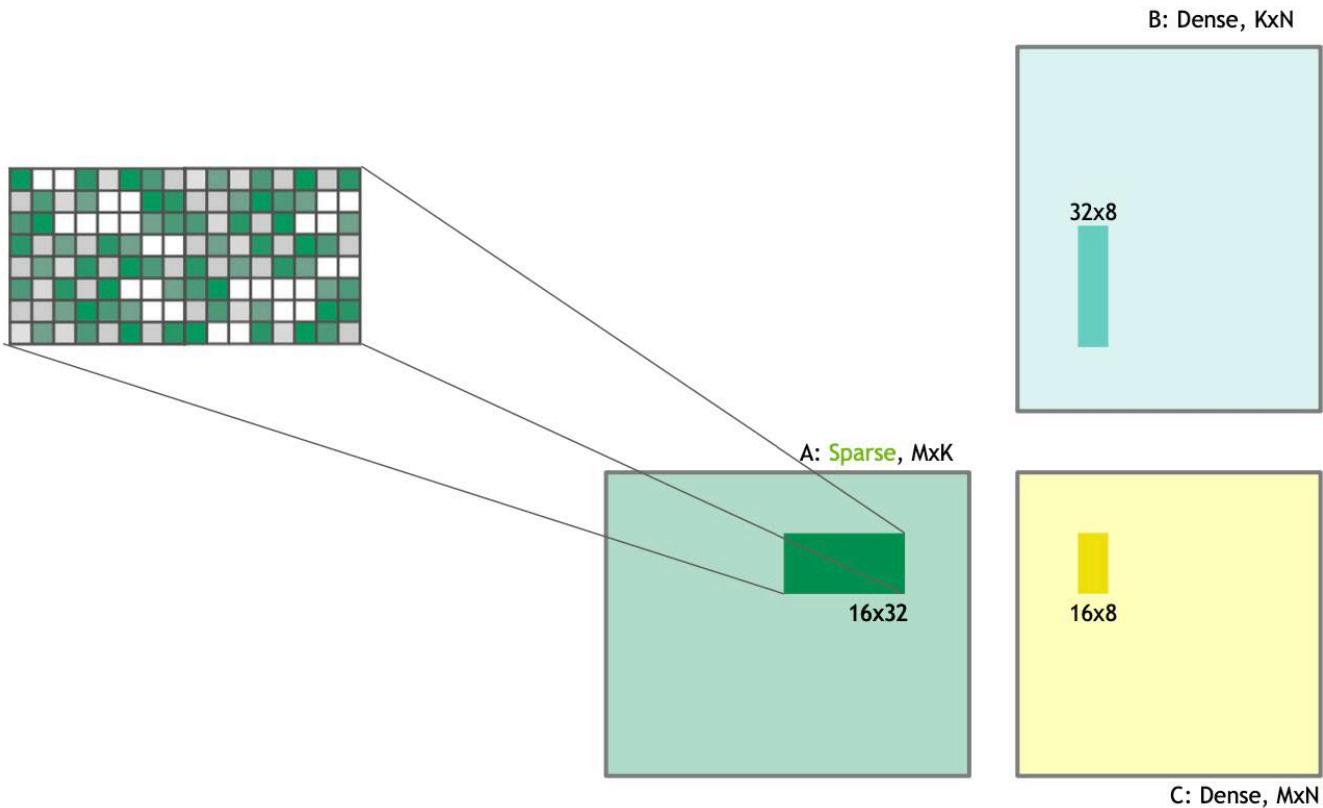
可以用1 cycle完成一个 $16 \times 16 * 16 \times 8 = 16 * 8$ 的矩阵乘法



需要用2 cycle完成一个 $16 \times 32 * 32 \times 8 = 16 * 8$ 的矩阵乘法

# Sparse tensor core做矩阵乘法

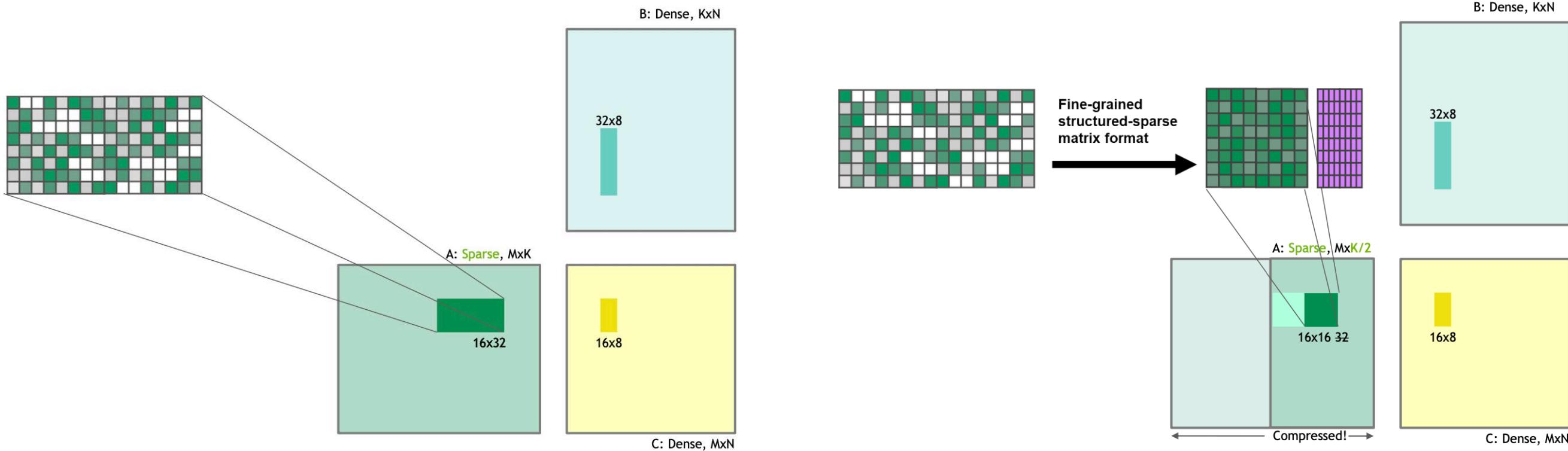
Sparse Tensor core(FP16)的计算  $A(M, K) * B (K, N) = C(M, N)$  的过程



但如果说A中的matrix拥有sparsity，是按照  
2:4的Pattern进行pruning，我们可以重构A

# Sparse tensor core做矩阵乘法

Sparse Tensor core(FP16)的计算  $A(M, K) * B(K, N) = C(M, N)$  的过程

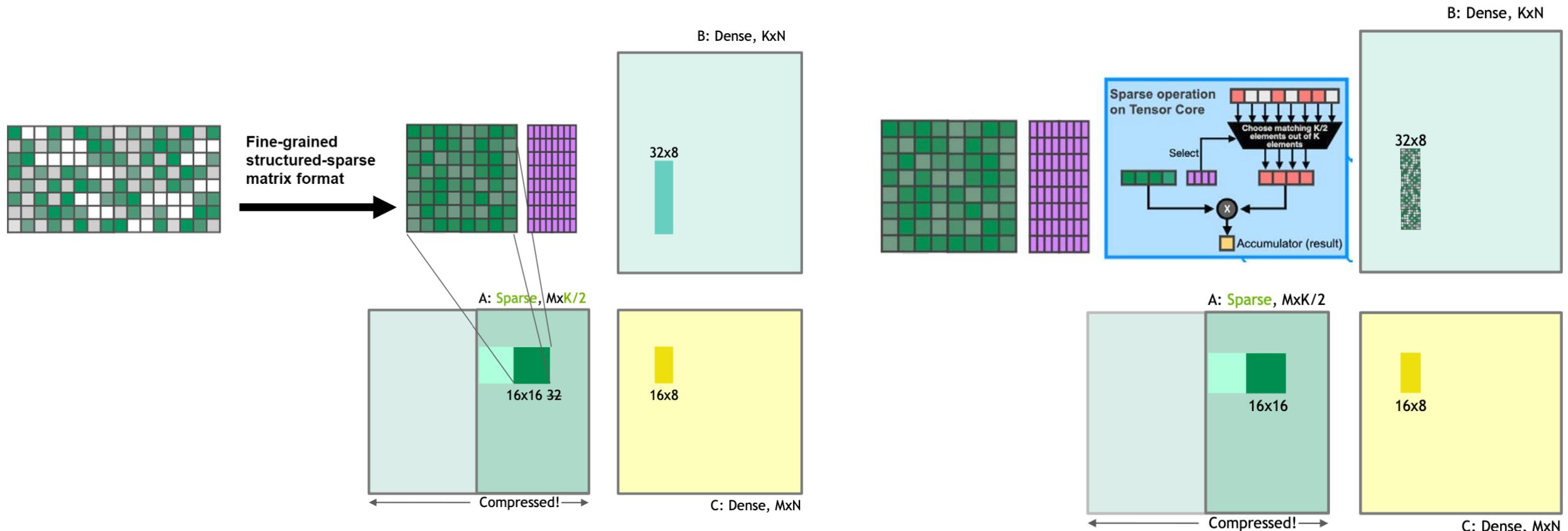


但如果说A中的matrix拥有sparsity，是按照2:4的Pattern进行pruning，我们可以重构A

重构后的A的memory占用空间直接减半达到压缩的效果。我们可以把这里的A理解为conv和FC中的weight

# Sparse tensor core做矩阵乘法

Sparse Tensor core(FP16)的计算  $A(M, K) * B (K, N) = C(M, N)$  的过程

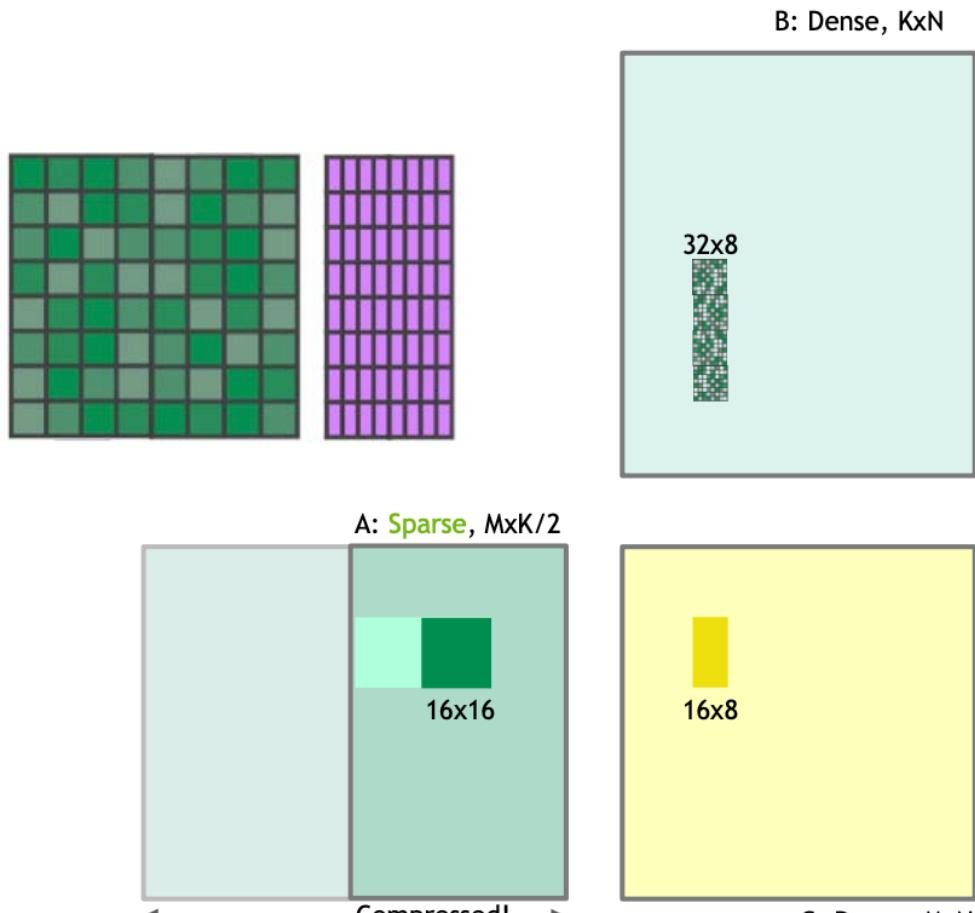
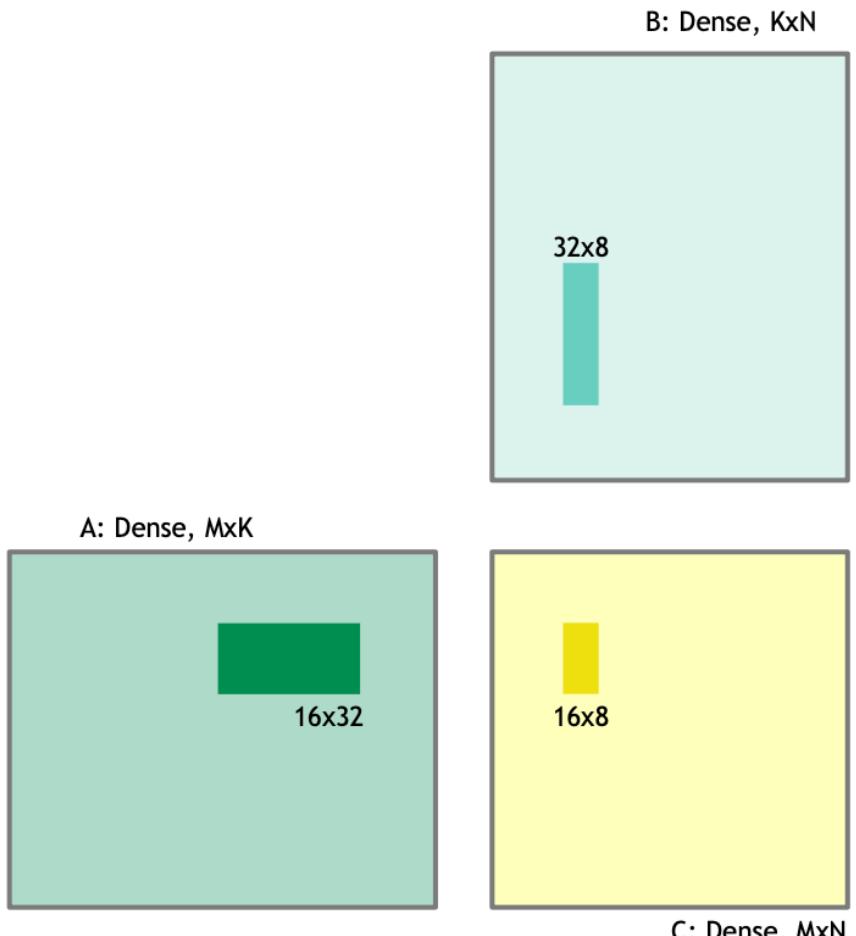


重构后的A的memory占用空间直接减半达到压缩的效果。我们可以把这里的A理解为conv和FC中的weight

那么对于B，可以通过索引对B中参与计算的值进行筛选(也可以理解为对B的重构)。我们可以把这里的B理解为conv和FC中的activation values

# Sparse tensor core做矩阵乘法

Sparse Tensor core(FP16)的计算  $A(M, K) * B(K, N) = C(M, N)$  的过程



使用dense tensor core需要用2 cycle完成一个  
 $16 \times 32 * 32 \times 8 = 16 * 8$ 的矩阵乘法

用sparse tensor core值需要用1 cycle就可以  
完成一个 $16 \times 32 * 32 \times 8 = 16 * 8$ 的矩阵乘法

## Ampere架构中的3rd Generation Tensor core

然而这里面很容易忽视的一点就是，为了实现sparse的计算而添加的额外操作的overhead

- compress weight的overhead
- reconstruct activation values的overhead

虽然这些是在硬件上可以完成，从而自动的将0参与的计算全部skip掉，然而这些多余的操作有时会比较凸显，尤其是当模型并不是很大，参与sparse的计算的激活值不是很大时，使用sparsity的特性做计算效果不是那么好。目前认为sparse tensor core在NLP领域的加速可能会比较可观。

