嵌入式图网络

具体方向：边缘设备多模态大模型推理优化(基于 Nvidia 设备)

一、阶段计划

（一）调研并学习相关模型推理优化算法

这方面技术内容目前大模型相对于视觉模型做的比较好，可以先从大模型推理优化入手，图神经网络作为视觉模型在边缘设备推理优化的过程中有很多可以借鉴的地方。

（二）推理优化算法实现

1. 计划现在先在3090（24G显存）上实现Llama3-8B等模型的设备离线推理部署，测试比较先进的一些模型量化以及推理优化方法（如：AWQ，SmoothQuant等），测试其推理准确率和加速比。
2. 在从Llama3-8B 模型过渡到常用的图神经网络模型（YOLO，ViT，Diffusion等模型），在3090上测试推理效果。
3. 过渡到在64G jetson-orin 上实现主流图网络算法，进行嵌入式离线推理部署，以及推理部署优化，优化方向有三个：

- 在维持模型架构的条件下，加快推理速度，做到较快推理出结果，减少推理时延。
- 考虑到嵌入式设备的显存大小有限，但是目前AI模型架构越来越大，部署时需要尽可能量化被部署模型的模型大小。
- 尽量减小模型量化后的精度损失。

这三者需要做到兼顾和平衡，才能呈现比较好的嵌入式系统AI推理。

二、模型量化技术原理

模型压缩主要分为如下几类：

- 剪枝（Pruning）

- 知识蒸馏（Knowledge Distillation）
- 量化

模型的量化技术主要分为两类：

1. 量化感知训练（QAT）
2. 训练后量化（PTQ）

模型量化面临着下面这些问题：

量化感知训练（QAT）由于训练成本较高并不实用，而训练后量化（PTQ）在低比特场景下面临较大的精度下降。

这里有一些基本的大模型量化技术原理的介绍：
大模型量化技术原理

什么是模型的context stage 和generate stage？如何理解？

答：类似于大模型当中的prefill阶段和decode阶段。prefill阶段用于处理一次性输入的prompts，计算出其KV矩阵，存储在KV缓存当中使用，这样方便后面的自适应循环生成新tokens（即decode阶段）使用，总的来说LLM推理过程分为两个阶段，PD（prefill和decode），并出现了一种较为有名的优化方法，就是**PD分离**。

对TFLOPS的认识：

✅ 一句话定义

> **TFLOPS（Tera FLOPS）表示每秒可以进行一万亿次浮点运算。**
> “FLOPS” = Floating Point Operations Per Second。

📌 名词解释

- **Tera** = 10^{12}（万亿）
- **FLOPS** = 浮点运算次数 / 秒（Floating Point Operations Per Second）

因此：

$$1\,\text{TFLOPS} = 10^{12}\,\text{浮点运算 / 秒}$$

📊 举个例子：

如果一个 GPU 标注为 20 TFLOPS，它能在 1 秒内执行约：

$$20 \times 10^{12} = 20\,000\,000\,000\,000\ \text{次浮点运算}$$

2.1 AWQ、AutoAWQ

AWQ（AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration）是一种对大模型仅权重量化方法。通过保护更“重要”的权重不进行量化，从而在不进行训练的情况下提高准确率。

原理是：权重对于LLM的性能并不同等重要”的观察，存在约（0.1%-1%）显著权重对大模型性能影响太大，通过跳过这1%的显著权重（salient weight）不进行量化，可以大大减少量化误差。但是这种方法由于是混合精度的，对硬件并不是很友好，这里AWQ技术主要是针对解决这个问题，所有权重都量化，但是是通过激活感知缩放保护显著权重。

具体细节在上面文章链接当中查看

主要应用： 可利用AWQ方法在jetson orin上部署llma2-70B参数的模型

论文名称：AWQ: ACTIVATION-AWARE WEIGHT QUANTIZATION FOR ON-DEVICE LLM COMPRESSION AND ACCELERATION

作者：Mit 韩松团队

会议：MLSys

时间：2024 Best Paper Award

AWQ对应在边缘设备上的应用是**TinyChat**：是一种尖端的聊天机器人界面，其设计可在 GPU 平台上实现轻量级资源消耗和快速推理。

LLaMA-3-8B 在 jetson-orin上获得了2.9倍的加速 (2.9x faster than FP16)，比纯FP16精度快2.9倍。性能提升对比如下：

Jetson Orin Results

Model	FP16 latency (ms)	INT4 latency (ms)	Speedup
LLaMA-3-8B	96.00	32.53	2.95x
LLaMA-2-7B	83.95	25.94	3.24x
LLaMA-2-13B	162.33	47.67	3.41x
Vicuna-7B	84.77	26.34	3.22x
VILA-7B	86.95	28.09	3.10x
VILA-13B	OOM	57.14	--
NVILA-2B	24.22	22.25	1.09x
NVILA-8B	86.24	30.48	2.83x

2.1.1 基础知识

（GEMM）General Matrix-Matrix Multiplication

2.1.2 技术内容

基于激活值来选择“重要的”权重通道比基于权重大小或者权重的L1、L2范式来选择“重要的”权重的效果要好得多

首先分析仅权重量化产生的误差。考虑一个权重为 w 的组/块；线性运算可写成 $y = wx$，量化后的对应运算为 $y = Q(\mathbf{w})\mathbf{x}$. 量化函数定义如下：

$$Q(\mathbf{w}) = \Delta \cdot \text{Round}(\frac{\mathbf{w}}{\Delta}), \quad \Delta = \frac{\max(|\mathbf{w}|)}{2^{N-1}}$$

其中，N 是量化比特数，Δ 是由绝对最大值(abs value)确定的量化缩放比例。

现在考虑一个权重元素 $w \in \mathbf{w}$，如果我们将 w 与 s （s>1）相乘， 然后，再用 x 除以 s （其idea来源于之前的工作SmoothQuant），我们将得到 $y = Q(\mathbf{w})\mathbf{x} = Q(w \cdot s)(x/s)$，即：

$$Q(w \cdot s) \cdot \frac{x}{s} = \Delta' \cdot \text{Round}(\frac{ws}{\Delta}) \cdot x \cdot \frac{1}{s}$$

其中，Δ' 是应用 s 后的新的量化缩放（scaler）因子。

2.2 LLM.int8()

属于：RTN（round-to-nearest (RTN) 量化）

（论文：LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale）是一种采用混合精度分解的量化方法。该方案先做了一个矩阵分解，对绝大部分权重和激活用8bit量化（vector-wise）。对离群特征的几个维度保留16bit，对其做高精度的矩阵乘法。

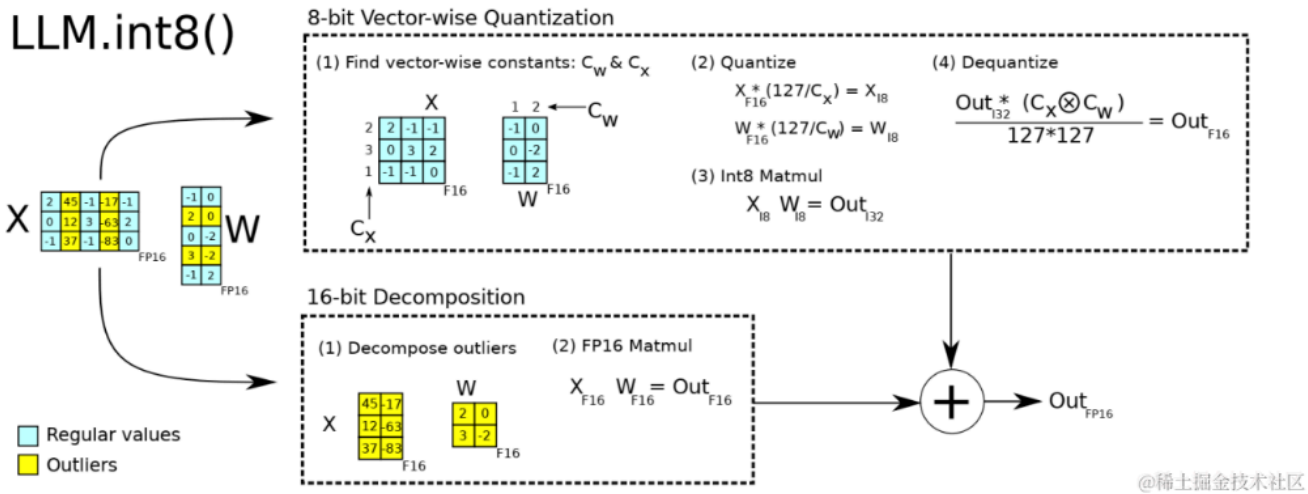


image.png

LLM.int8() 通过三个步骤完成矩阵乘法计算:

1. 从输入的隐含状态中，按列提取异常值 (离群特征，即大于某个阈值的值)。
2. 对离群特征进行 FP16 矩阵运算，对非离群特征进行量化，做 INT8 矩阵运算;
3. 反量化非离群值的矩阵乘结果，并与离群值矩阵乘结果相加，获得最终的 FP16 结果。

实验结果表明该方法效果良好。可以通过使用 LLM.int8() 的量化过程来恢复全部性能。您可以清楚地看到随着模型参数量逐渐变多 8 比特基线（即 vector-wise quantization）的性能大幅下降。 而 LLM.int8() 方法使用**vector-wise quantization**和**混合精度分解**来恢复全部性能。

但是这种分离计算的方式拖慢了推理速度

int8量化是量化 任何模型最简单的方法之一，这里对于了离群值的处理回大大拖慢了推理速度，因为在计算过程中分成两部分计算，离群值部分用fp16来计算，被int8量化的非离群值用int8来计算。

2.3 GPTQ（2022）

GPTQ(论文：GPTQ: ACCURATE POST-TRAINING QUANTIZATION FOR GENERATIVE PRE-TRAINED TRANSFORMERS) 采用 int4/fp16 (W4A16) 的混合量化方案，其中模型权重被量化为 int4 数值类型，而激活值则保留在 float16，是一种仅权重量化方法。在推理阶段，模型权重被动态地反量化回 float16 并在该数值类型下进行实际的运算；同 OBQ 一样，GPTQ还是从单层量化的角度考虑，希望找到一个量化过的权重，使的新的权重和老的权重之间输出的结果差

别最小。

技术原理

GPTQ(论文：**GPTQ: ACCURATE POST-TRAINING QUANTIZATION FOR GENERATIVE PR E-TRAINED TRANSFORMERS**) 采用 int4/fp16 (W4A16) 的混合量化方案，其中模型权重被量化为 int4 数值类型，而激活值则保留在 float16，是一种仅权重量化方法。在推理阶段，模型权重被动态地反量化回 float16 并在该数值类型下进行实际的运算；同 OBQ 一样，GPTQ还是从单层量化的角度考虑，希望找到一个量化过的权重，使的新的权重和老的权重之间输出的结果差别最小。

GPTQ 将权重分组（如：128列为一组）为多个子矩阵（block）。对某个 block 内的所有参数逐个量化，每个参数量化后，需要适当调整这个 block 内其他未量化的参数，以弥补量化造成的精度损失。因此，GPTQ 量化需要准备校准数据集。

GPTQ 量化过程如下图所示。首先，使用 Cholesky 分解中 Hessian 矩阵的逆，在给定的step中对连续列的块（粗体）进行量化，并在step结束时更新剩余的权重（蓝色）。量化过程在每个块内递归应用：白色中间列表示当前正在被量化。

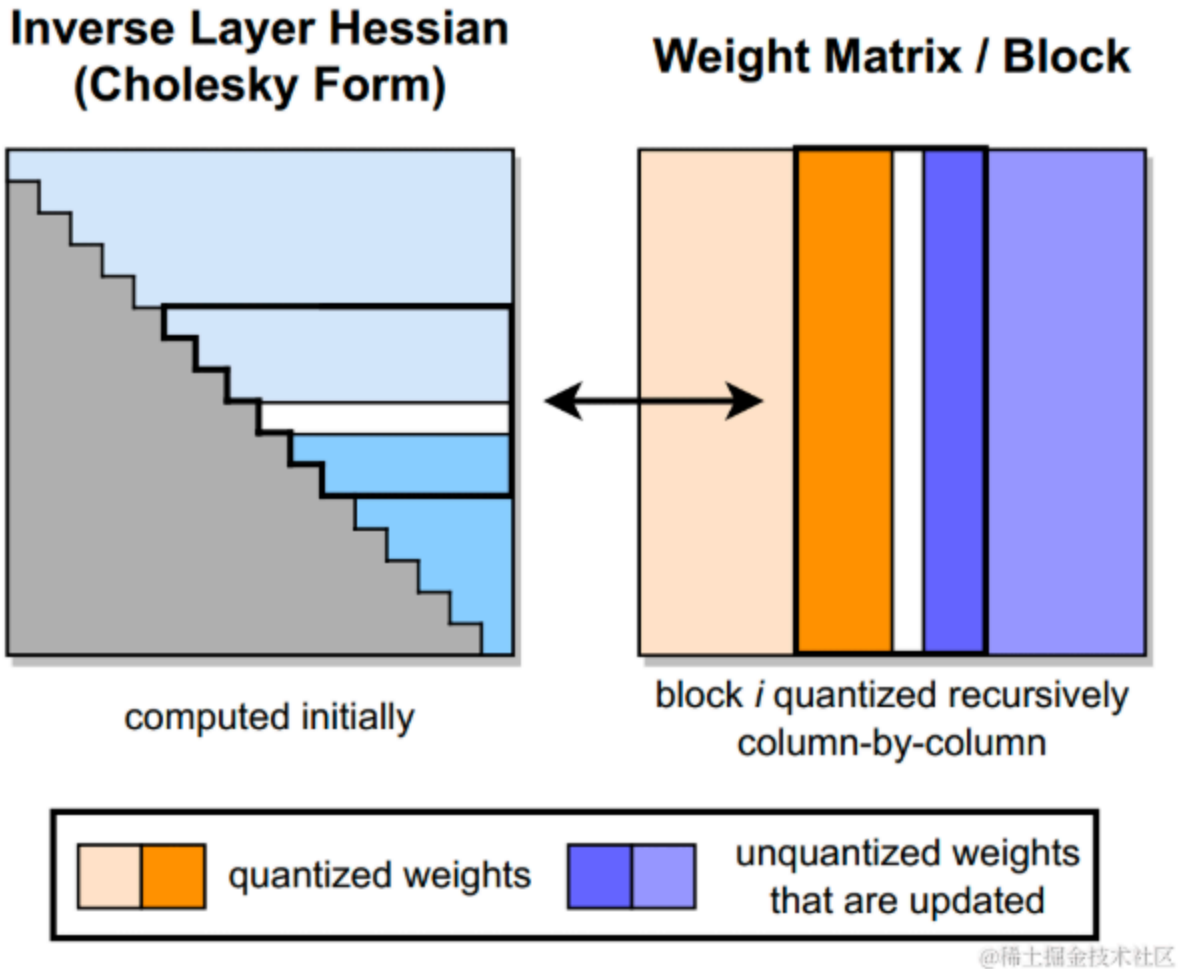


image.png

那GPTQ有什么优势呢？

GPTQ 的创新点如下：

- **取消贪心算法**：OBS 采用贪心策略，先量化对目标影响最小的参数；但 GPTQ 发现直接按顺序做参数量化，对精度影响也不大。这项改进使得**参数矩阵每一行的量化可以做并行的矩阵计算**（这意味着我们可以独立地对每一行执行量化。即所谓的 per-channel quantization）。对于大模型场景，这项改进使得量化速度快了一个数量级；
- **Lazy Batch-Updates**：OBQ 对权重一个个进行单独更新，作者发现性能瓶颈实际在于GPU的内存带宽，而且同一个特征矩阵W不同列间的权重更新是不会互相影响的。因此作者提出了延迟批处理的方法，通过延迟一部分参数的更新，一次处理多个（如：128）列，来**缓解带宽的压力**，大幅提升了计算速度。
- **Cholesky 分解**：用 Cholesky 分解求海森矩阵的逆，提前计算好所有需要的信息，在**增强数值稳定性**的同时，后续更新的过程中再计算，进一步减少了计算量。

GPTQ的伪代码如下所示，包括了上面讨论的一些优化：

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

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

image.png

2.4 SmoothQuant

驱动安装

Nvidia 驱动

cuda 驱动

三、Ampere（A100） 架构学习（Orin 用的GPU同为Ampere架构）

每个A100 有108个SM（流式多处理器），每个SM有64个Cuda核心（int32，fp32），一个block最多对应对应1024个线程（这个是硬件当中预先定义好的，无法改变，所以cuda编程时，每个block的线程设置不能超过1024个），多个block对应我们A100架构的一个SM处理单元，block被分到某个SM上，则会保存到该SM上直到执行解说，同一时间段一个SM可以同时容纳多个block，每个SM中有1024个FMA独立计算单元，对应2048个独立的浮点运算，等效为2048个线程（这里不是SM的cuda core总数，而是最大活跃线程，即一个时钟周期可以执行2048个线程，block内线程的个数设置成1024，即最大活跃线程的一半），至于为什么是2048，因为一个SM有4个warp scheduler，最多能同时管理 64 个 warps（64*32=2048*）*A100总共108个SM，所以A100总共存在108*2048=221184个并发线程（最大活跃线程）。
所以cuda编程当中，一个block只能写1024个线程即：

dim3 blockdim(1024,1,1)或者
dim3 blockdim(32,32,1)....

反正最后相乘起来结果为1024.

至于grid：
硬件限制由 CUDA 运行时和 GPU 的硬件寄存器固定，具体为：

gridDim.x 最大值： $2^{31} - 1 = 2147483647$

gridDim.y 最大值： 65535

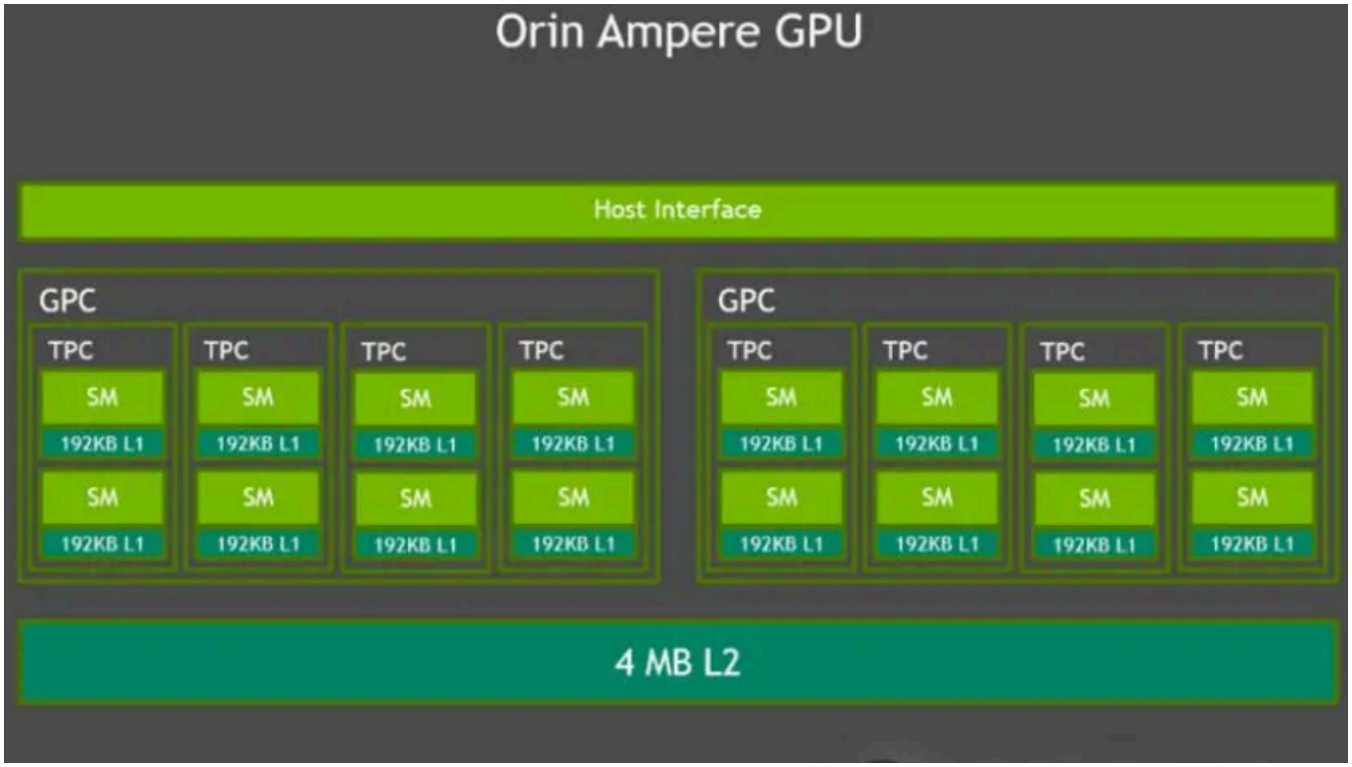
gridDim.z 最大值： 65535


SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
FP64
TENSOR CORE
LD/ ST
SFU
192KB L1 Data Cache / Shared Memory
Tex

1. Orin GPU结构

Orin采用NVIDIA Ampere GPU，具有两个GPC（Graphics Processing Clusters）和128个CUDA Core。总计2048个CUDA Core和64个Tensor Core，INT8稀疏算力高达170 TOPS。Ampere GPU支持CUDA语言，提供高级并行处理计算能力，并在图形处理和深度学习方面表现卓越。

	Jetson AGX Orin 系列				Jetson Orin NX 系列		Jetson Orin Nano 系列		
	Jetson AGX Orin 开发者套件	Jetson AGX Orin 64GB	Jetson AGX Orin 工业版	Jetson AGX Orin 32GB	Jetson Orin NX 16GB	Jetson Orin NX 8GB	Jetson Orin Nano Super 开发者套件	Jetson Orin Nano 8GB	Jetson Orin Nano 4GB
AI 性能	275 TOPS		248 TOPS	200 TOPS	157 TOPS	117 TOPS	67 TOPS	67 TOPS	34 TOPS
GPU	搭载 64 个 Tensor Core 的 2048 核 NVIDIA Ampere 架构 GPU			搭载 56 个 Tensor Core 的 1792 核 NVIDIA Ampere c GPU	搭载 32 个 Tensor Core 的 1024 核 NVIDIA Ampere 架构 GPU		搭载 32 个 Tensor Core 的 1024 核 NVIDIA Ampere 架构 GPU		搭载 16 个 Tensor Core 的 512 核 NVIDIA Ampere 架构 GPU
GPU 最大频率	1.3 GHz		1.2 GHz	930 MHz	1173MHz	1173MHz	1020MHz	1020MHz	1020MHz



Cuda Core 和Tensor Core 的使用方式是不一样的，cuda core 是单元素计算，而tensor core是直接输入矩阵进行计算

使用的API 为WMMA API

除了WMMA API, 也可以通过使用 NVIDIA 库来来间接使用 Tensor Cores. 例如cuBLAS 和 cuDNN. cuBLAS 利用 Tensor Cores 加速密集矩阵乘法(GEMM)计算; cuDNN 则利用 Tensor

Cores 加速卷积和循环神经网络(RNNs)的计算.

下面这篇博客详细讲解了Tensor Core的使用方法：

https://blog.csdn.net/CV_Autobot/article/details/138460383