



TensorRT-LLM应用与部署最佳实践

By Yuxin

Agenda

- TensorRT-LLM简介
- TensorRT-LLM全流程初体验
 - Hello World
 - 准确度与性能测试
- 多卡部署大语言模型
 - 张量并行 (Tensor Parallelism)
 - 流水线并行 (Pipeline Parallelism)
- 低精度量化：降本增效
 - 模型量化
 - ✓ INT4/INT8 weight-only
 - ✓ INT4 AWQ/GPTQ
 - ✓ INT8 SmoothQuant
 - ✓ FP8
 - KV Cache量化
 - ✓ INT8
 - ✓ FP8
- 性能调优选项
 - In-flight Batching
 - Multi-block Mode
- TensorRT-LLM优化效果

TensorRT-LLM简介

生态定位与主要特性

丰富的预定义模型:

Baichuan、BART、BERT、Blip2、BLOOM、ChatGLM、FairSeq NMT、Falcon、Flan-T5、GPT、GPT-J、GPT-Nemo、GPT-NeoX、InternLM、LLaMA、LLaMA-v2、Mamba、mBART、Mistral、MPT、mT5、OPT、Phi-1.5/Phi-2、Qwen、Replit Code、RoBERTa、SantaCoder、StarCoder1、StarCoder2、T5、Whisper、持续扩充中.....

多样的量化方法:

- INT4/INT8 Weight-Only
- INT4 AWQ/GPTQ
- INT8 SmoothQuant
- FP8
- INT8/FP8 KV Cache

高效的工程实现:

- In-flight Batching
- Tensor/Pipeline Parallelism
- Fused Multi Head Attention
- Multi-Block Mode
- Horizontal Fusion in Gated-MLP
- NCCL
- Cutlass
- 基于TensorRT的网络优化
-

TensorRT-LLM

TensorRT

CUDA

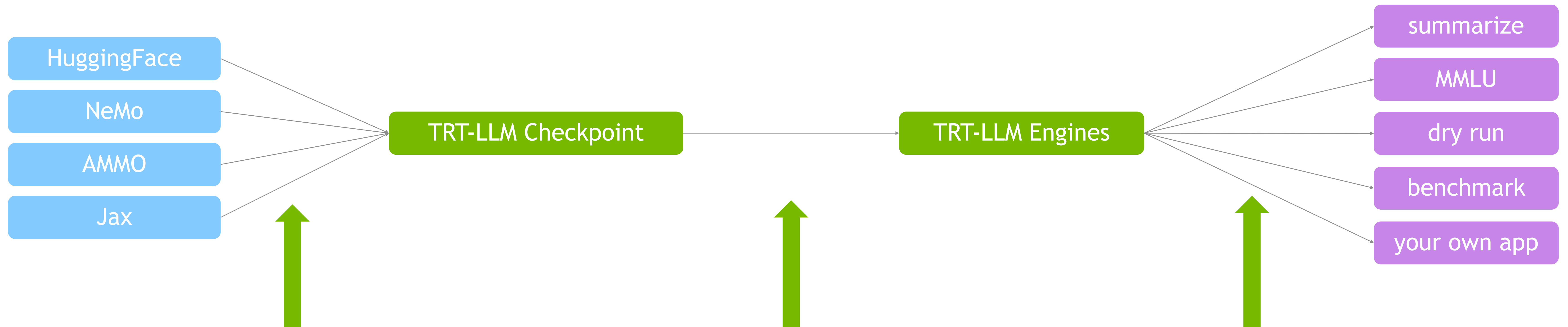
提供一组API以定义大型语言模型（LLM），并将LLM转为充分优化的TensorRT引擎，以在NVIDIA GPU上高效执行LLM推理。此外还为流行的LLMs做了充分优化，并支持多种量化方法，做到了高性能LLM的**开箱即用**。

深度神经网络推理（inference）优化器和运行时，专注于在 NVIDIA GPU上快速高效地运行已经训练好的网络。优化原理包括图优化、层融合、算子择优、量化等。

GPU通用编程SDK，与NVIDIA GPU紧密协同设计（co-design），能充分发挥GPU并行计算性能。

TensorRT-LLM全流程初体验

概览



使用脚本**convert_checkpoint.py**或**quantize.py**，将多种外部格式的模型转为TRT-LLM定义的checkpoint格式。这一步确定了逻辑层面的参数：

- 量化方式
- 并行方式
-

使用**trtllm-build**命令，将checkpoint转化为TensorRT Engines并加以优化。这一步确定了实现层面的参数：

- max_batch_size
- max_input_len
- max_output_len
- max_beam_width
- plugin_config
-

注意，大量的自动优化发生在这一步，优化与特定参数、特定硬件平台紧密相关，不建议跨场景、跨硬件平台混用。

使用C++/Python API做二次开发。TensorRT-LLM内置了若干工具，也可作为二次开发的参考：

- summarize.py做文本总结
- mmlu.py做准确度测试
- run.py运行一次推理验证模型可行性
- benchmark做性能测试

运行时可选项：

- temperature
- top K
- top P
-

TensorRT-LLM全流程初体验

环境搭建 (以v0.8.0为例)

- 准备源代码:

```
git lfs install
git clone https://github.com/NVIDIA/TensorRT-LLM.git
cd TensorRT-LLM
git checkout v0.8.0
git submodule update --init --recursive
git lfs pull
```

- 构建docker镜像:

```
make -C docker release_build
```

```
yuxinz@ipp2-1600:~/scratch/TensorRT-LLM$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
tensorrt_llm/release	latest	6c4d79e1b884	2 minutes ago	33.9GB

- 启动docker容器:

```
make -C docker release_run DOCKER_RUN_ARGS="-v /home/scratch.trt_llm_data/llm-models/:/llm-models"
```

```
root@ipp2-1600:/app/tensorrt_llm# ls
README.md  benchmarks  docs  examples  include  lib
root@ipp2-1600:/app/tensorrt_llm# ls benchmarks/
bertBenchmark  gptManagerBenchmark  gptSessionBenchmark
root@ipp2-1600:/app/tensorrt_llm# ls /llm-models/
Baichuan-13B-Chat  bart-large-cnn  falcon-180b  hugging-face-cache
Baichuan-7B        bloom           falcon-40b   int4-quantized-gptq-awq
Baichuan2-13B-Chat  bloom-3b       falcon-7b-instruct  internlm-20b
Baichuan2-7B-Chat   bloom-560m     falcon-rw-1b  internlm-7b
```

Note:

此目录存放了从HuggingFace上提前下载好的预训练模型，映射到容器内。或者进入容器再下载也OK。

TensorRT-LLM全流程初体验

Hello World: 运行LLaMA 7B, 跟大家说Hello~

- 将下载的Hugging Face模型转为TensorRT-LLM定义的checkpoint:

```
cd examples/llama
python convert_checkpoint.py \
    --model_dir /llm-models/llama-models/llama-7b-hf/ \
    --dtype float16 \
    --output_dir ./checkpoint_1gpu_fp16
```

```
[root@ipp2-1600:/app/tensorrt_llm/examples/llama# ls ./checkpoint_1gpu_fp16/
config.json  rank0.safetensors
```

- 从checkpoint构建engine:

```
trtllm-build \
    --checkpoint_dir ./checkpoint_1gpu_fp16 \
    --gemm_plugin float16 \
    --output_dir ./engine_1gpu_fp16
```

```
[root@ipp2-1600:/app/tensorrt_llm/examples/llama# ls ./engine_1gpu_fp16/
config.json  rank0.engine
```

- 运行engine:

```
python ../run.py \
    --engine_dir ./engine_1gpu_fp16 \
    --tokenizer_dir /llm-models/llama-models/llama-7b-hf/ \
    --max_output_len 100 \
    --input_text "Hello!"
```

```
[TensorRT-LLM] TensorRT-LLM version: 0.8.0Input [Text 0]: "<s> Hello!"
Output [Text 0 Beam 0]: "I'm a 20 year old college student from the United States. I'm currently studying English and Japanese at a university in Japan.
I'm a very friendly and outgoing person, and I love to meet new people. I'm also a very hard worker, and I'm always looking for new ways to improve myself. I'm very interested in Japanese culture, and I'm always looking for new ways to learn about it. I'm also very interested"
```


TensorRT-LLM全流程初体验

MMLU准确度测试

MMLU (Massive Multitask Language Understanding) 是一项用于衡量大语言模型性能的指标。在examples目录下自带了mmlu.py脚本用来测试准确度。

PS: 测试时往往会输入较大的数据量, 建议用较大的 `max_batch_size`、`max_input_len`和`max_output_len`构建engine:

```
trtllm-build \
    --checkpoint_dir ./checkpoint_1gpu_fp16 \
    --output_dir ./engine_1gpu_fp16 \
    --gemm_plugin float16 \
    --max_batch_size 32 \
    --max_input_len 4096 \
    --max_output_len 4096
```

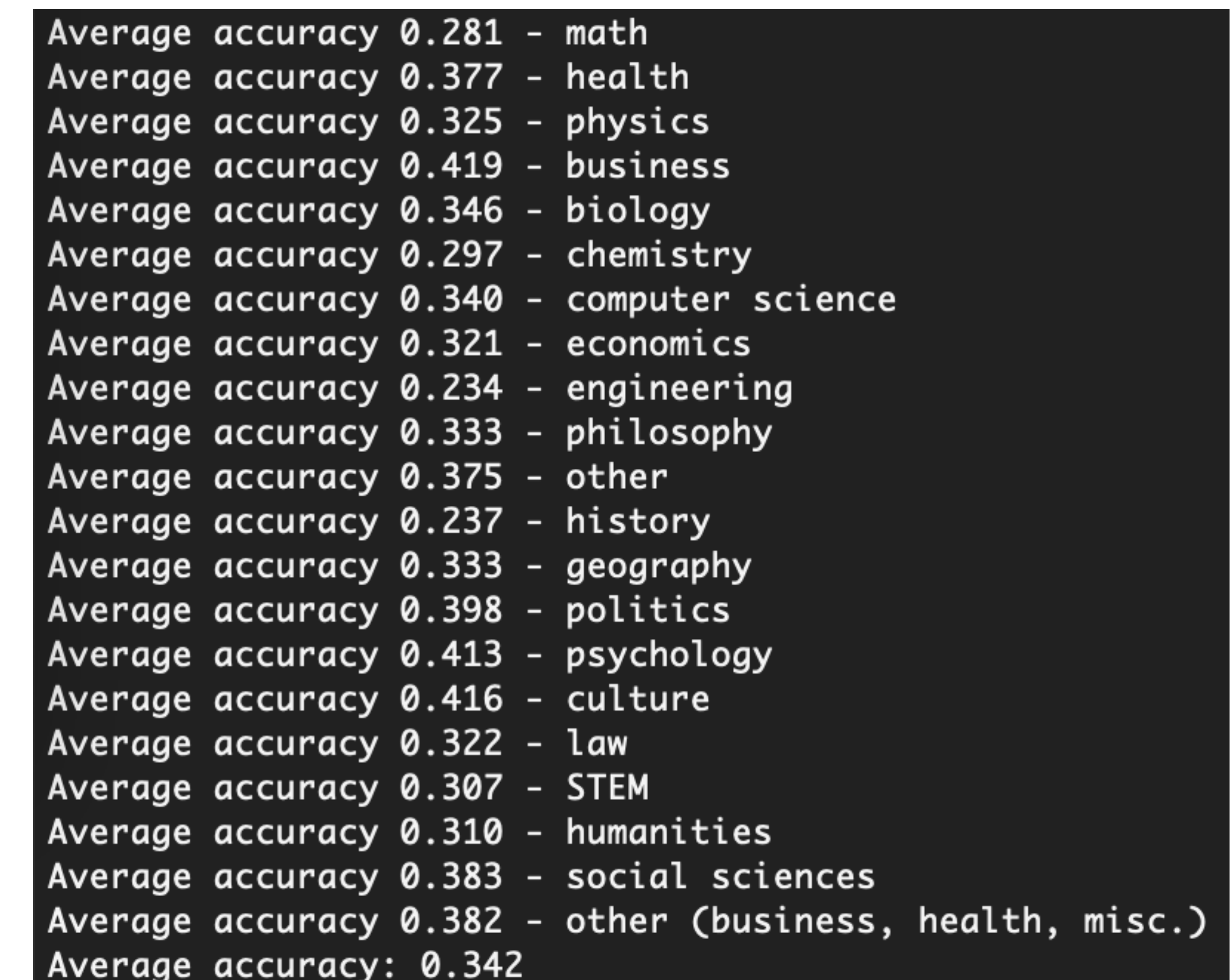
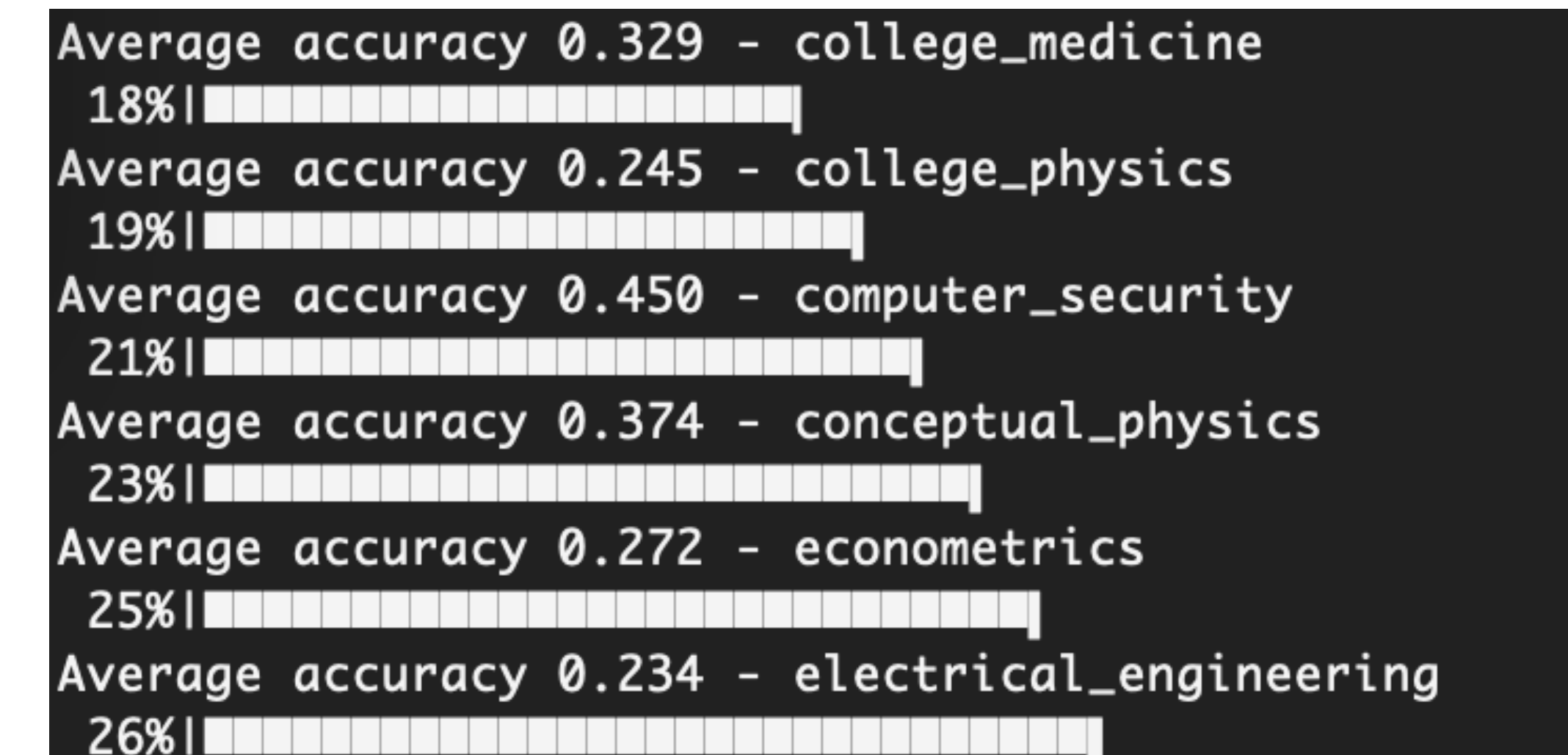
- MMLU准确度测试:

```
python ../mmlu.py \
  --hf_model_dir /llm-models/llama-models/llama-7b-hf \
  --engine_dir ./engine_1gpu_fp16 \
  --data_dir ~/mmlu_data \
  --test trt llm
```

Note:

<https://people.eecs.berkeley.edu/~hendrycks/data.tar>

下载MMLU测试数据集，解压至~/mmlu_data
即可，详见`python ../mmlu.py --help`。




TensorRT-LLM全流程初体验

性能测试

除了模型准确度以外，另一个重要指标是吞吐和延迟。可以使用benchmarks目录下的gptSessionBenchmark工具测试。

● 吞吐、延迟测试：

```
../.. /benchmarks/gptSessionBenchmark \
--model llama \
--engine_dir ./engine_1gpu_fp16 \
--batch_size "1;4;8" \
--input_output_len "128,256;2048,1024"
```

Batch Size		Input Length	Output Length
1		128	256
4		2048	1024
8			

```
Benchmarking done. Iteration: 21, duration: 61.28 sec.
Latencies: [2871.35, 2875.32, 2884.59, 2899.68, 2898.96, 2904.56, 2907.28, 2909.71, 2915.06, 2916.96 ... 2926.44, 2928.90, 2935.29, 2933.63, 2939.82, 2936.00, 2948.38, 2949.53, 2955.81]
[BENCHMARK] batch_size 1 input_length 128 output_length 256 latency(ms) 2918.14 tokensPerSec 87.73 gpu_peak_mem(gb) 93.76
Benchmarking done. Iteration: 19, duration: 60.12 sec.
Latencies: [3143.31, 3147.25, 3145.06, 3150.71, 3151.77, 3155.55, 3154.69, 3156.81, 3161.72, 3163.14, 3167.84, 3168.77, 3176.23, 3176.22, 3176.82, 3176.33, 3181.45, 3183.38, 3186.71]
[BENCHMARK] batch_size 4 input_length 128 output_length 256 latency(ms) 3164.41 tokensPerSec 323.60 gpu_peak_mem(gb) 93.76
Benchmarking done. Iteration: 18, duration: 60.40 sec.
Latencies: [3320.23, 3319.27, 3322.72, 3325.68, 3327.34, 3327.06, 3333.64, 3337.44, 3338.79, 3342.92, 3379.91, 3358.46, 3364.76, 3372.57, 3384.88, 3381.41, 3386.26, 3476.00]
[BENCHMARK] batch_size 8 input_length 128 output_length 256 latency(ms) 3355.52 tokensPerSec 610.34 gpu_peak_mem(gb) 93.76
Benchmarking done. Iteration: 10, duration: 138.74 sec.
Latencies: [13402.53, 13528.52, 13818.45, 14012.44, 13892.45, 14005.48, 14125.52, 13872.07, 14003.50, 14077.10]
[BENCHMARK] batch_size 1 input_length 2048 output_length 1024 latency(ms) 13873.81 tokensPerSec 73.81 gpu_peak_mem(gb) 93.76
Benchmarking done. Iteration: 10, duration: 179.32 sec.
Latencies: [17948.45, 17777.35, 18028.21, 17917.76, 17917.68, 18012.46, 17766.62, 18096.03, 18040.86, 17813.74]
[BENCHMARK] batch_size 4 input_length 2048 output_length 1024 latency(ms) 17931.91 tokensPerSec 228.42 gpu_peak_mem(gb) 93.76
Benchmarking done. Iteration: 10, duration: 233.14 sec.
Latencies: [23264.92, 23461.17, 23106.46, 23329.73, 23190.00, 23471.56, 23391.62, 23291.49, 23350.85, 23285.87]
[BENCHMARK] batch_size 8 input_length 2048 output_length 1024 latency(ms) 23314.37 tokensPerSec 351.37 gpu_peak_mem(gb) 93.76
```

Tips: 在运行run.py和mmlu.py时，除了指定engine目录外，还需要指定原始模型目录，而在benchmark时却不需要。这是因为前两者需要经历文本到token的过程与token到文本的逆过程，也就是需要用到tokenizer，而tokenizer位于原始模型目录。而benchmark则跳过了tokenizer，输入是token，输出也是token，测试的是engine本身的性能。

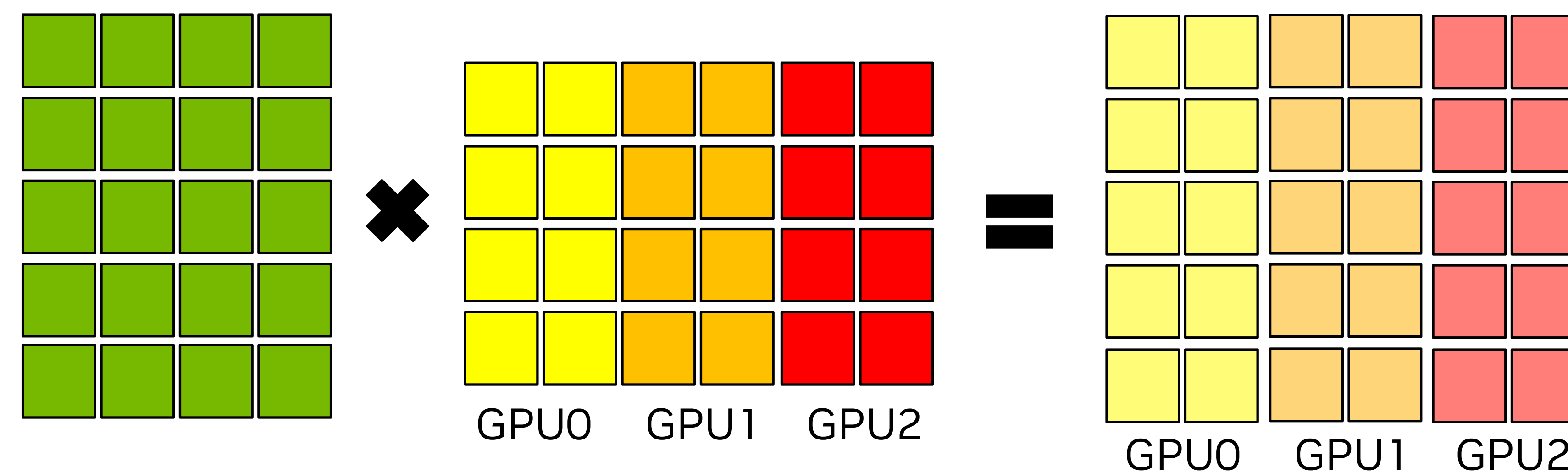
多卡部署大语言模型

多卡并行原理

当一个模型大到单卡无法容纳时，就需要分布在多卡。多卡如何协作就是并行方式。在 TensorRT-LLM 中可以使用 Tensor 和 Pipeline 两种并行模式。

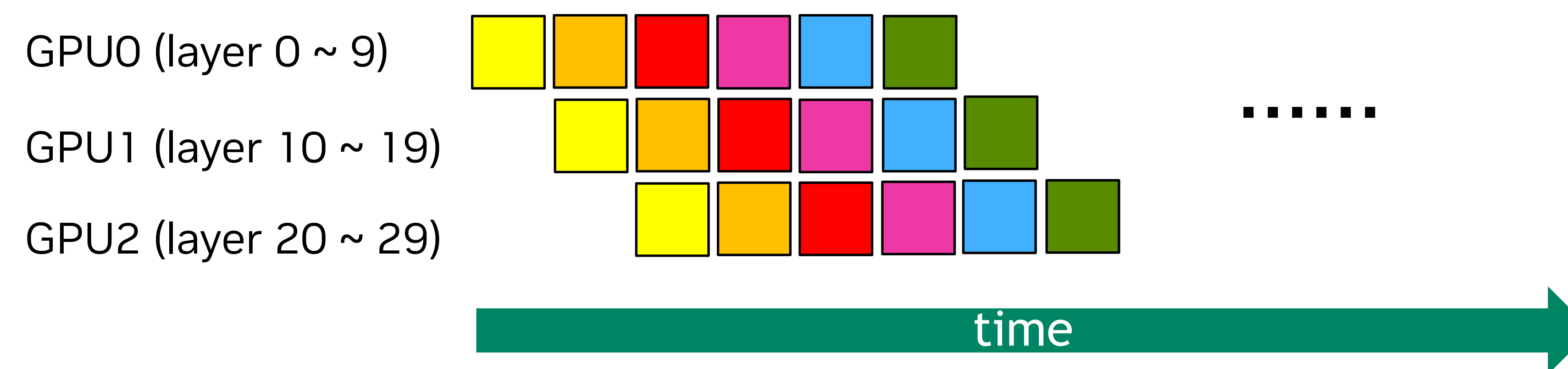
- 张量并行 (Tensor Parallelism)

利用矩阵运算可以分块的原理，将一个大矩阵运算拆分到多个GPU上的小矩阵运算，然后再合并结果。



- 流水线并行 (Pipeline Parallelism)

将网络按层分割，分布到多张GPU上，一个batch经过一张GPU处理进入下一张GPU，而对于某个特定GPU，把当前batch送入下一张GPU后，便处理下一个batch。



多卡部署大语言模型

使用Tensor Parallelism与Pipeline Parallelism

在使用convert_checkpoint.py生成checkpoint时，可以启用多卡部署。参数--tp_size控制张量并行（Tensor Parallelism），参数--pp_size控制流水线并行（Pipeline Parallelism）。如下命令将LLaMA 65B部署到8张GPU上，并采用张量并行：

```
python convert_checkpoint.py \
    --model_dir /llm-models/llama-models/llama-65b-hf/ \
    --output_dir ./checkpoint_8gpu_fp16 \
    --dtype float16 \
    --tp_size 8 --pp_size 1
```

如果--tp_size 1 --pp_size 8则使用流水线并行。当然，--tp_size和--pp_size是正交的，也就是说可以同时使用，比如--tp_size 4 --pp_size 2，那么也是用了4x2=8张卡。

查看checkpoint_8gpu_fp16目录，可以看到有8个safetensors文件，分别对应8张卡需要加载的数据。

```
root@a4u8g-0140:/app/tensorrt_llm/examples/llama# ls checkpoint_8gpu_fp16/
config.json      rank1.safetensors rank3.safetensors rank5.safetensors rank7.safetensors
rank0.safetensors rank2.safetensors rank4.safetensors rank6.safetensors
```

trtllm-build命令则无需额外参数，它根据checkpoint_8gpu_fp16目录下的config.json和8个safetensors文件就会为8张GPU各生成一个优化后的engine文件。

```
root@a4u8g-0140:/app/tensorrt_llm/examples/llama# ls engine_8gpu_fp16/
config.json  rank1.engine rank3.engine rank5.engine rank7.engine
rank0.engine rank2.engine rank4.engine rank6.engine
```

多卡模型运行时加上mpirun前缀即可，-n参数指定使用的GPU数量，需与build时保持一致。

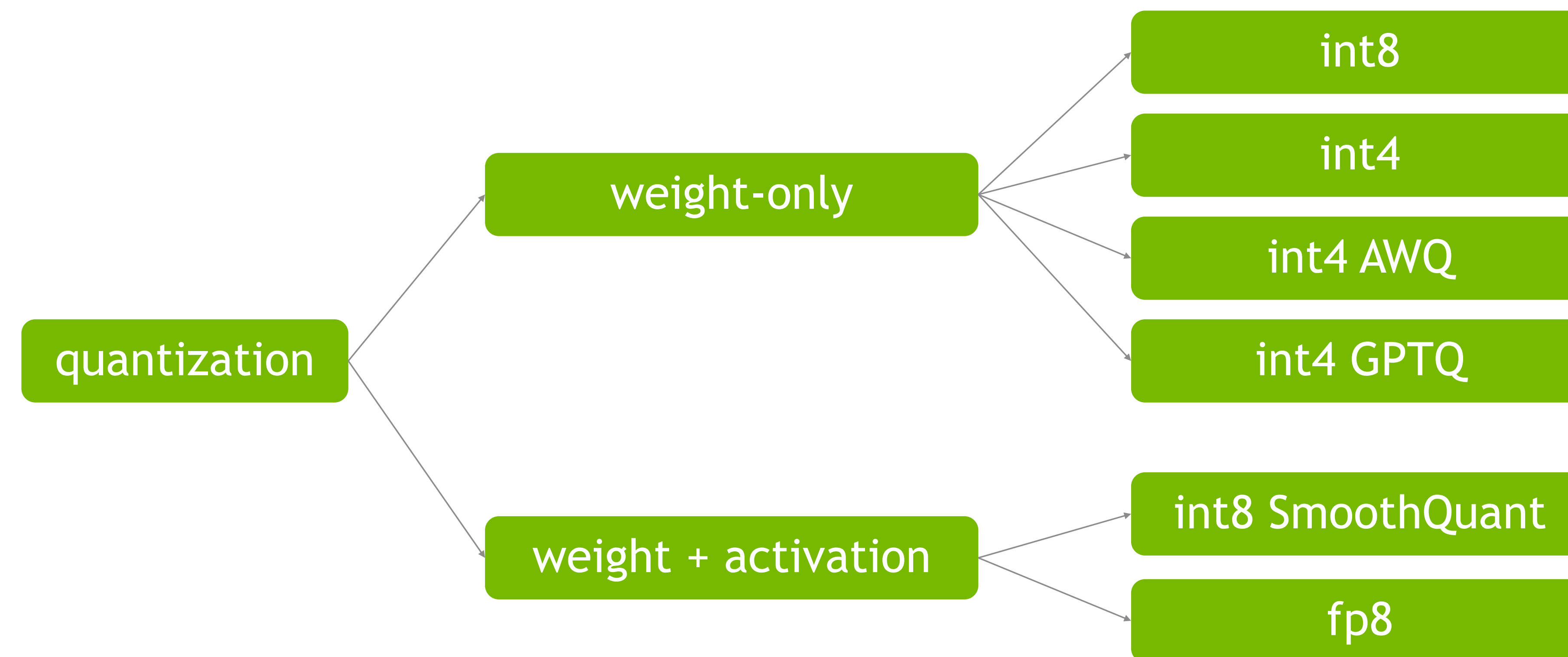
```
mpirun --allow-run-as-root -n 8 \
python ../run.py \
    --engine_dir ./engine_8gpu_fp16 \
    --tokenizer_dir /llm-models/llama-models/llama-65b-hf/ \
    --max_output_len 100 \
    --input_text "Hello, everyone!"
```


低精度量化：降本增效

TensorRT-LLM支持的量化方式

之前的示例中，模型使用的都是FP16数据类型。如果能够使用更窄的数据类型（比如8位整数INT8、8位浮点数FP8，甚至4位整数INT4），那么模型权重就能减少一半的存储空间，所需要的GPU显存量与个数能够大幅减少，计算时也能节省内存带宽，提高吞吐、降低延迟。这称为“量化”。

量化难免会对模型准确性造成一定影响，学界业界关于量化方法有很多的研究与创新，主要分为如下方式：



Weight-only只量化模型的权重，运行时的中间结果（activation）仍为fp16。而weight + activation方法则把两者都量化为8位，能够节省更多显存。在有些文档中会看到W4A16、W8A16或W8A8这样的标识，W为权重的位数，A为激活值的位数。

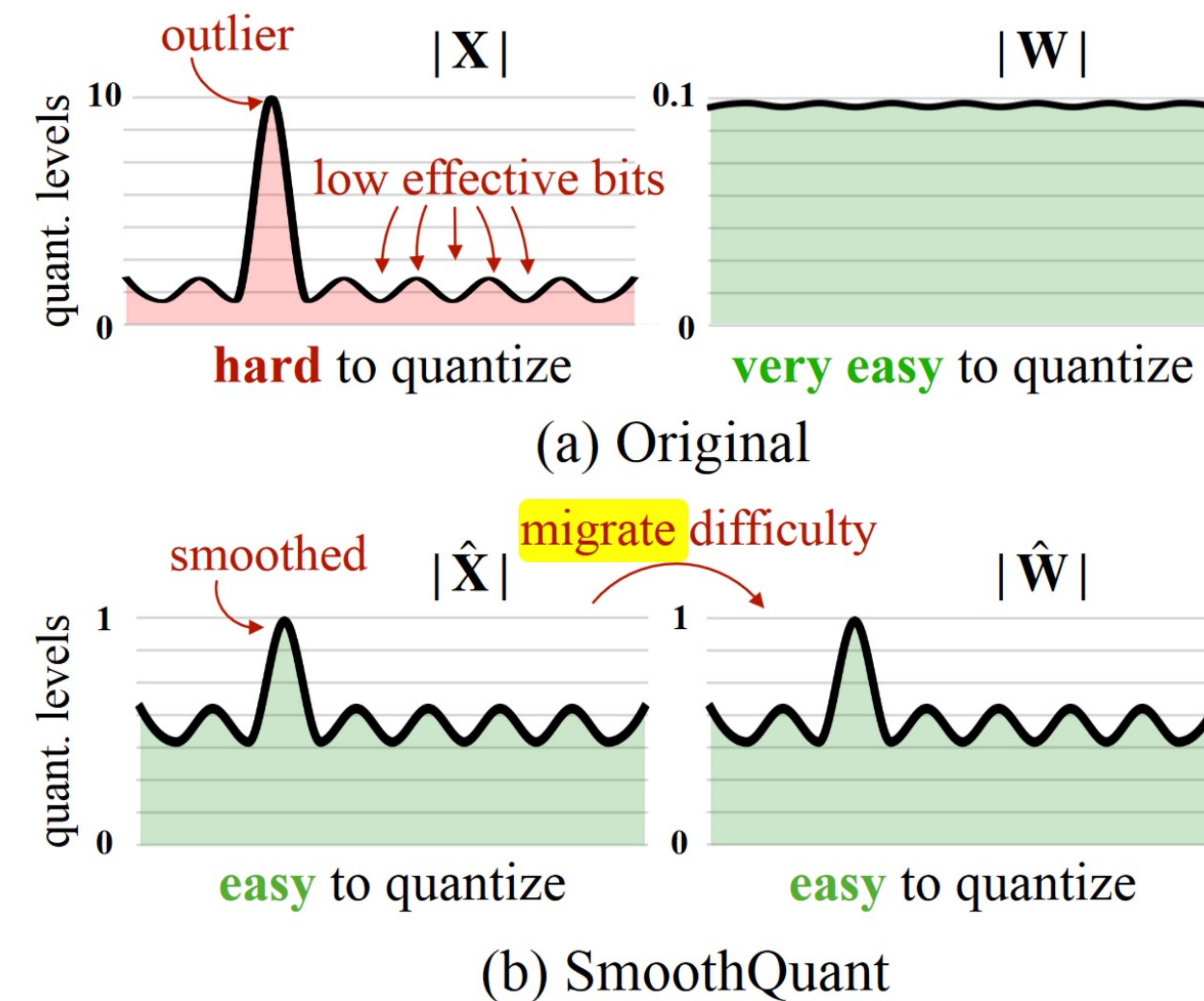
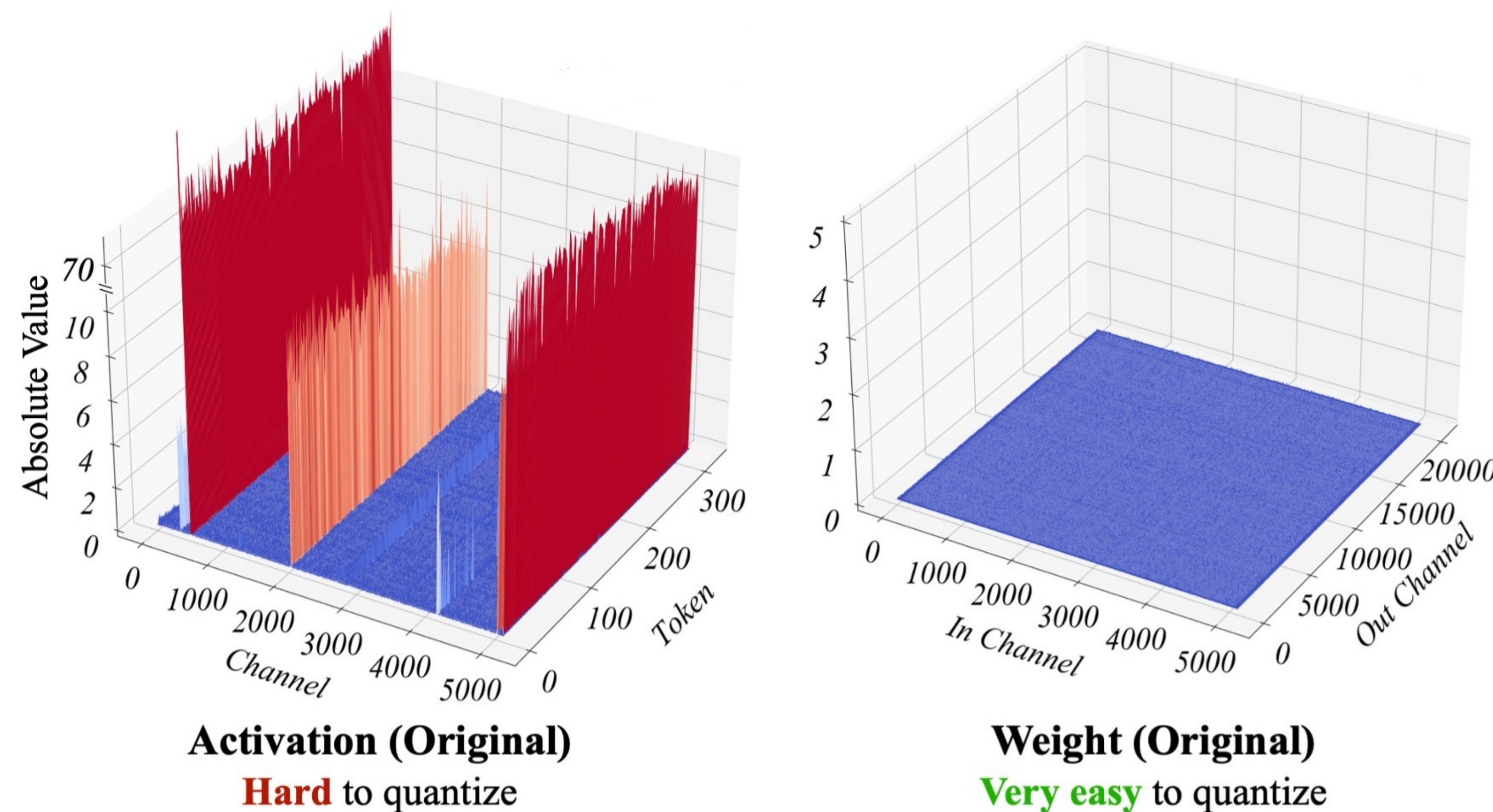
参考论文：

1. Int4 AWQ: [《AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration》](#)
2. Int8 GPTQ: [《GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers》](#)
3. Int8 SmoothQuant: [《SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models》](#)
4. FP8: [《FP8 Formats for Deep Learning》](#)

低精度量化：降本增效

SmoothQuant量化的原理

值得一提的是，量化方法之所以会分为weight-only和weight+activation两种，是基于这么一个事实：大语言模型中的权重分布比较均匀，很容易量化，而激活值的数值分布相差悬殊，有很多离群点，难以量化。下图展示了大语言模型中的数值分布：

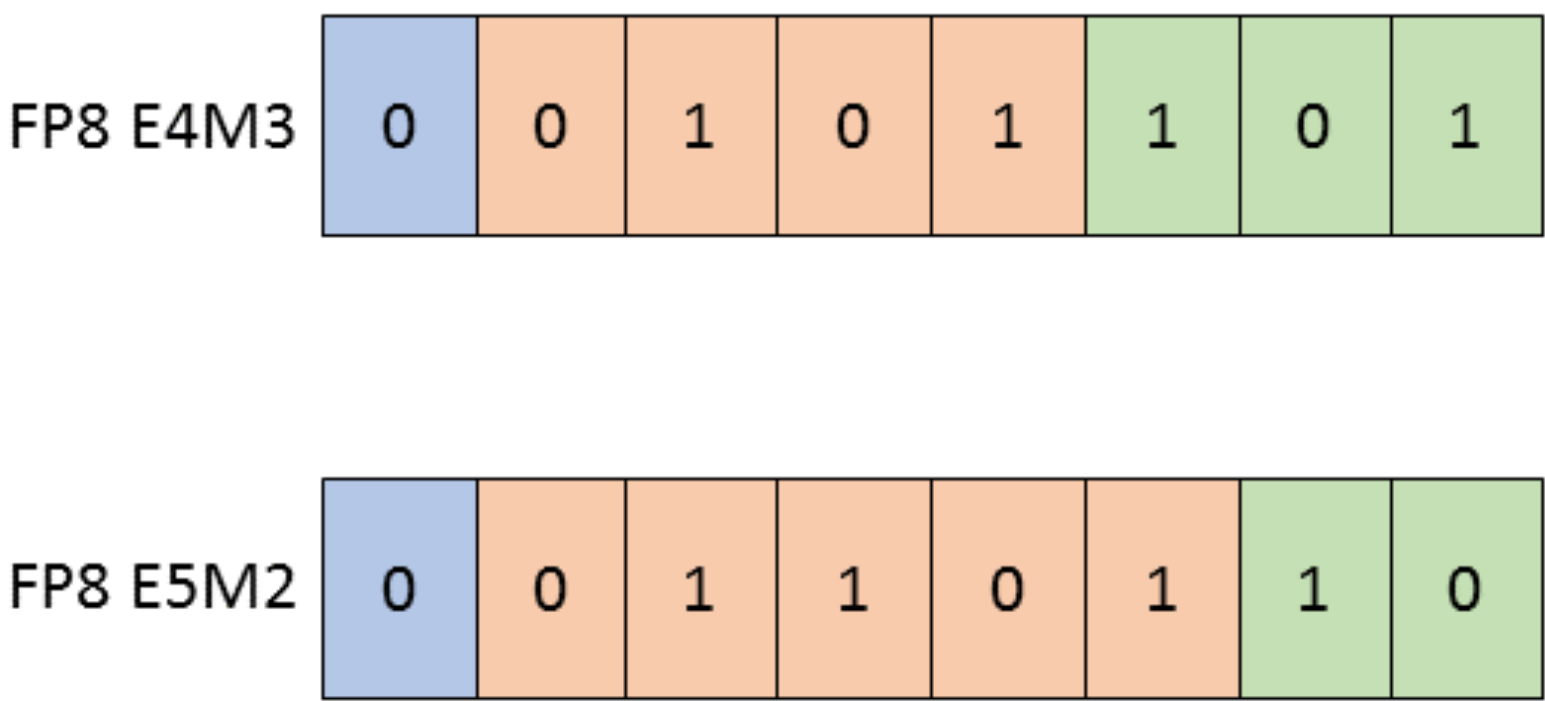


SmoothQuant的思路很巧妙，通过一个数学变换，将activation的分布“拍扁”一些，而将weight的分布“拉大”一些，在确保最后乘积不变的前提下，将量化的难度“匀一匀”，让weight分担一部分activation的压力，使得两者都处于容易量化的区间。

低精度量化：降本增效

FP8量化的原理

Activation量化的难点在于分布尺度大，很容易溢出到整数型的表达范围之外，因为整数型在数轴上的取值是均匀分布的。那如果使用指数分布的浮点数呢？FP8分为E4M3和E5M2两种格式，区别就是指数位与尾数位的权衡。E4M3的尾数位较多，因此数值更加精确，但是表达范围较小。E5M2则反之。但不过哪一种，其表达范围都比INT8大很多。



		E4M3	E5M2
表达范围	最大	448	57,344
	最小 (0除外)	1.56×10^{-2}	6×10^{-5}
精度	eps*	0.125	0.25

将INT8、FP8 E4M3和FP8 E5M2画在数轴上更加直观。在正半轴上，INT8最远只能到达127，而E4M3可以到448，E5M2更是能到57344。尽管在[8,127]这段区间FP8的分布密度不如INT8，但是FP8表达范围宽广，不容易溢出。

损失精度，最多导致12.5%的误差，但数值溢出，却会导致100%的错误。FP8尽管丢了芝麻（精度），但是留住西瓜（数值范围）。因此，FP8的量化并不需要像SmoothQuant那样有巧妙的设计和复杂的校准过程，量化过程又自然又快速，模型的准确度还普遍较好，是一种很有前景的量化方法。

低精度量化：降本增效

模型量化示例

量化发生在生成checkpoint的步骤中。除了convert_checkpoint.py之外，另一个与量化相关的脚本是quantize.py，不同的量化方法所使用的脚本会有所不同。以LLaMA 7B为例，依次演示各种量化方式的使用方法。

- Int8 weight-only:

```
python convert_checkpoint.py \  
  --model_dir /llm-models/llama-models/llama-7b-hf/ \  
  --output_dir ./checkpoint_1gpu_int8_wo \  
  --dtype float16 \  
  --use_weight_only --weight_only_precision int8
```

- Int4 weight-only:

```
python convert_checkpoint.py \  
  --model_dir /llm-models/llama-models/llama-7b-hf/ \  
  --output_dir ./checkpoint_1gpu_int4_wo \  
  --dtype float16 \  
  --use_weight_only --weight_only_precision int4
```

- Int4 AWQ:

```
python ../quantization/quantize.py \  
  --model_dir /llm-models/llama-models/llama-7b-hf/ \  
  --output_dir ./checkpoint_1gpu_int4_awq \  
  --dtype float16 \  
  --qformat int4_awq --awq_block_size 128 --calib_size 32
```

- Int4 GPTQ:

```
python convert_checkpoint.py \  
  --model_dir /llm-models/llama-models/llama-7b-hf/ \  
  --output_dir ./checkpoint_1gpu_int4_gptq \  
  --dtype float16 \  
  --use_weight_only --weight_only_precision int4_gptq --per_group \  
  --ammo_quant_ckpt_path /llm-models/int4-quantized-gptq-awq/llama-7b-4bit-gs128.safetensors
```

Note:
GPTQ的权重生成依赖外部库GPTQ-for-LLaMa，详见
examples/llama/README.md，这里直接使用已经预生成的权重。

低精度量化：降本增效

模型量化示例

● Int8 SmoothQuant:

```
python convert_checkpoint.py \  
    --model_dir /llm-models/llama-models/llama-7b-hf/ \  
    --output_dir ./checkpoint_1gpu_int8_sq \  
    --dtype float16 \  
    --smoothquant 0.5 --per_token --per_channel
```

● FP8:

```
python ../quantization/quantize.py \  
    --model_dir /llm-models/llama-models/llama-7b-hf/ \  
    --output_dir ./checkpoint_1gpu_fp8 \  
    --dtype float16 \  
    --qformat fp8 --calib_size 512
```

上述6种量化方式外加原始的FP16的checkpoint都使用如下相同的命令生成engine（注意将xxx替换为对应数据类型）：

```
trtllm-build \  
    --checkpoint_dir ./checkpoint_1gpu_xxx \  
    --output_dir ./engine_1gpu_xxx \  
    --gemm_plugin float16 \  
    --max_batch_size 16 \  
    --max_input_len 2048 \  
    --max_output_len 2048
```

可以通过查看各个engine的大小确定量化确实起效了。

```
root@s4124-0105:/app/tensorrt_llm/examples/llama# ls -lh engine_1gpu_*  
engine_1gpu_fp16:  
total 13G  
-rw-r--r-- 1 root root 3.7K Apr 10 05:09 config.json  
-rw-r--r-- 1 root root 13G Apr 10 05:09 rank0.engine  
  
engine_1gpu_fp8:  
total 6.6G  
-rw-r--r-- 1 root root 3.1K Apr 10 05:15 config.json  
-rw-r--r-- 1 root root 6.6G Apr 10 05:15 rank0.engine  
  
engine_1gpu_int4_awq:  
total 3.7G  
-rw-r--r-- 1 root root 3.2K Apr 10 05:12 config.json  
-rw-r--r-- 1 root root 3.7G Apr 10 05:12 rank0.engine  
  
engine_1gpu_int4_gptq:  
total 3.7G  
-rw-r--r-- 1 root root 3.7K Apr 10 05:12 config.json  
-rw-r--r-- 1 root root 3.7G Apr 10 05:13 rank0.engine  
  
engine_1gpu_int4_wo:  
total 3.6G  
-rw-r--r-- 1 root root 3.7K Apr 10 05:10 config.json  
-rw-r--r-- 1 root root 3.6G Apr 10 05:10 rank0.engine  
  
engine_1gpu_int8_sq:  
total 6.6G  
-rw-r--r-- 1 root root 3.8K Apr 10 05:14 config.json  
-rw-r--r-- 1 root root 6.6G Apr 10 05:14 rank0.engine  
  
engine_1gpu_int8_wo:  
total 6.6G  
-rw-r--r-- 1 root root 3.7K Apr 10 05:11 config.json  
-rw-r--r-- 1 root root 6.6G Apr 10 05:11 rank0.engine
```


低精度量化：降本增效

KV Cache量化示例

Attention机制中的K和V可以通过缓存来避免重复计算，从而提升性能。除了weight和activation占用显存外，KV Cache也可以通过量化来减少显存占用，从而容纳更大的模型，或者支持更大的batch size等。KV Cache默认采用fp16类型，目前支持使用int8或fp8两种量化方法。

如果模型的量化使用了convert_checkpoint.py脚本，那么可以使用--int8_kv_cache或者--fp8_kv_cache选项开启KV Cache的量化。如果模型的量化使用了quantize.py脚本，那么可以使用参数--kv_cache_type {int8, fp8}开启KV Cache的量化。比如：

- Int4 weight-only + Int8 KV Cache:

```
python convert_checkpoint.py \  
    --model_dir /llm-models/llama-models/llama-7b-hf/ \  
    --output_dir ./checkpoint_1gpu_int4_wo_int8_kvc \  
    --dtype float16 \  
    --use_weight_only --weight_only_precision int4 \  
    --int8_kv_cache
```

- Int8 SmoothQuant + Int8 KV Cache:

```
python convert_checkpoint.py \  
    --model_dir /llm-models/llama-models/llama-7b-hf/ \  
    --output_dir ./checkpoint_1gpu_int8_sq_int8_kvc \  
    --smoothquant 0.5 --per_token --per_channel \  
    --int8_kv_cache
```

- FP8 + FP8 KV Cache:

```
python ../quantization/quantize.py \  
    --model_dir /llm-models/llama-models/llama-7b-hf/ \  
    --output_dir ./checkpoint_1gpu_fp8_fp8_kvc \  
    --qformat fp8 --calib_size 512 \  
    --kv_cache_dtype fp8
```


低精度量化：降本增效

量化后性能对比

以FP16为基准，在H100上对 6种LLaMA 7B量化后的模型做性能测试，得到加速比如下表所示：

Input Length	Output Length	Batch Size	Int8 WO	Int4 WO	Int4 AWQ	Int4 GPTQ	Int8 SQ	FP8
128	1024	1	1.587	1.971	1.890	2.036	1.063	1.456
		2	1.556	1.796	1.709	1.875	1.070	1.492
		4	1.503	1.924	1.732	1.972	1.340	1.793
		8	1.443	1.668	1.513	1.534	1.464	1.803
		16	1.364	1.445	1.424	1.401	1.366	1.712
		32	1.405	1.503	1.464	1.309	1.442	1.784
256	2048	1	1.656	2.023	1.942	2.096	1.099	1.559
		2	1.663	2.186	1.988	2.082	1.302	1.712
		4	1.486	1.831	1.632	1.712	1.403	1.782
		8	1.474	1.665	1.544	1.479	1.449	1.843
		16	1.417	1.559	1.453	1.314	1.408	1.742
		32	1.488	1.574	1.532	1.235	1.519	1.784

结论：当batch size较大（计算压力较大）时，FP8量化的性能是最佳的！

低精度量化：降本增效

量化后MMLU准确率对比

规模小的模型的MMLU得分自然较低，在低基数下比较准确度损失不够明显，因此这里直接引用了对各个较大规模模型的准确度测试数据：

模型	量化方法	基准MMLU(FP16)	量化后MMLU	MMLU损失
Falcon-180B	FP8	70.4	70.3	0.14%
	INT8-SQ	70.4	68.6	2.56%
	INT4-AWQ	70.4	69.8	0.85%
Falcon-40B	FP8	56.1	55.6	0.89%
	INT8-SQ	56.1	54.7	2.50%
	INT4-AWQ	56.1	55.5	1.07%
LLaMA-v2-70B	FP8	69.1	68.5	0.87%
	INT8-SQ	69.1	67.2	2.75%
	INT4-AWQ	69.1	68.4	1.01%
MPT-30B	FP8	47.5	47.4	0.21%
	INT8-SQ	47.5	46.8	1.47%
	INT4-AWQ	47.5	46.5	2.11%

结论：当模型规模较大时，FP8量化的准确度都是最佳的！

低精度量化：降本增效

量化方法选择的最佳实践

用途不同，用户能接受的准确度影响和校准所需时间会有所不同。下表总结了选择量化方法时要考虑的权衡因素。



量化方法	性能提升 (batch size <= 4)	性能提升 (batch size >= 16)	准确度影响	校准时间
FP8	中	高	极低	几分钟
Int8 SQ	中	中	低	几分钟
Int8 weight-only	中	低	低	无
Int4 weight-only	高	低	高	无
Int4 AWQ	高	低	低	数十分钟
Int4 GPTQ	高	低	低	数十分钟

在batch size ≤ 4的推理场景下，主要的考量是内存带宽，因此推荐使用weight-only量化方法。而对于大batch的推理场景，例如线上服务场景（batch size ≥ 16），内存带宽和计算密度都成了关键因素，因此建议选择weight + activation量化方法。

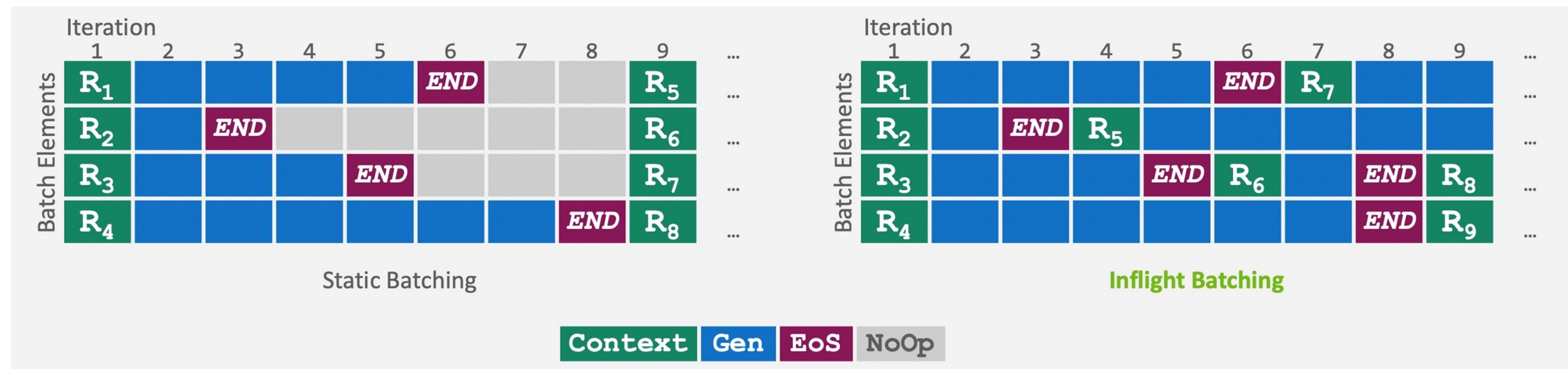
经验而言，我们建议优先使用FP8，因为它通常提供了最佳的性能与精度。如果FP8的效果不及预期，则进一步依次尝试Int8 SmoothQuant、Int4 AWQ或Int4 GPTQ。

性能调优选项

In-flight Batching

将多个请求组合为batch是提高吞吐率的一个常用手段。传统的Static Batching模式中，多个请求一起提交、一起返回。但是大语言模型的一个特殊性是，不同请求的输出长度是不定的（遇到EOS即停止），导致先完成的请求会等待后完成的请求，从而导致GPU闲置。

而In-flight Batching则会将已完成（输出了EOS）的请求踢出，并替换为一条新的请求，从而避免了“空泡”，增加GPU利用率，增加吞吐，降低first token的时延。



在使用trtllm-build命令生成engine时，当以下三个选项同时开启时，In-flight Batching会自动开启：

- `--gpt_attention_plugin float16`
- `--remove_input_padding enable`
- `--paged_kv_cache enable`

详见[文档](#)。

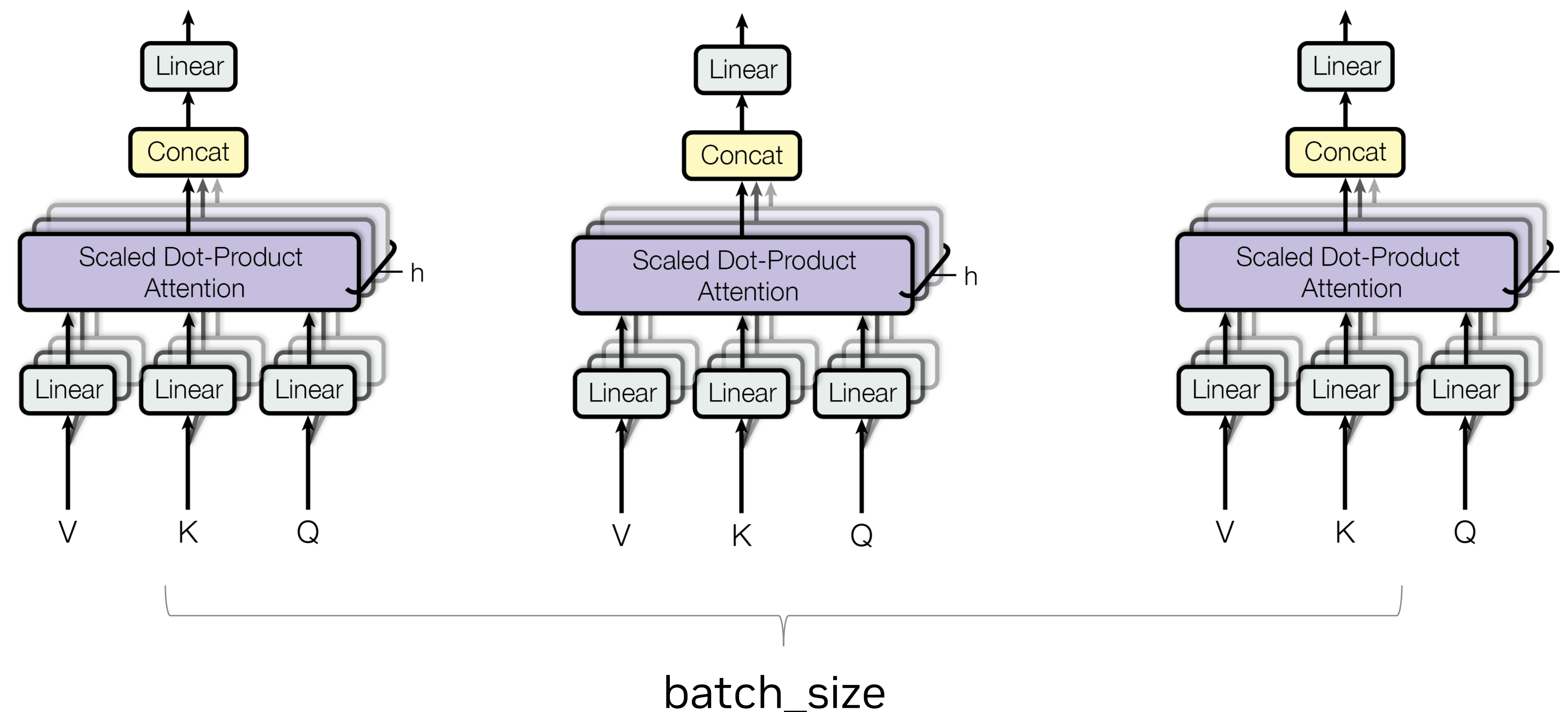
性能调优选项

Multi-block Mode

当满足以下条件时，建议使用trtllm-build时尝试--multi_block_mode参数，并评估对性能的影响：

1. $\text{Sequence_count} * \text{num_head} < \text{multiprocessor_count} / 2$
2. $\text{Input_seq_len} > 1024$ （一个经验值，表示上下文长度足够长）

若一个Batch中有batch_size个input sequences，而Multi Head Attention中的head数为num_heads，那么计算过程就天然地拥有了batch_size * num_heads的并行度。



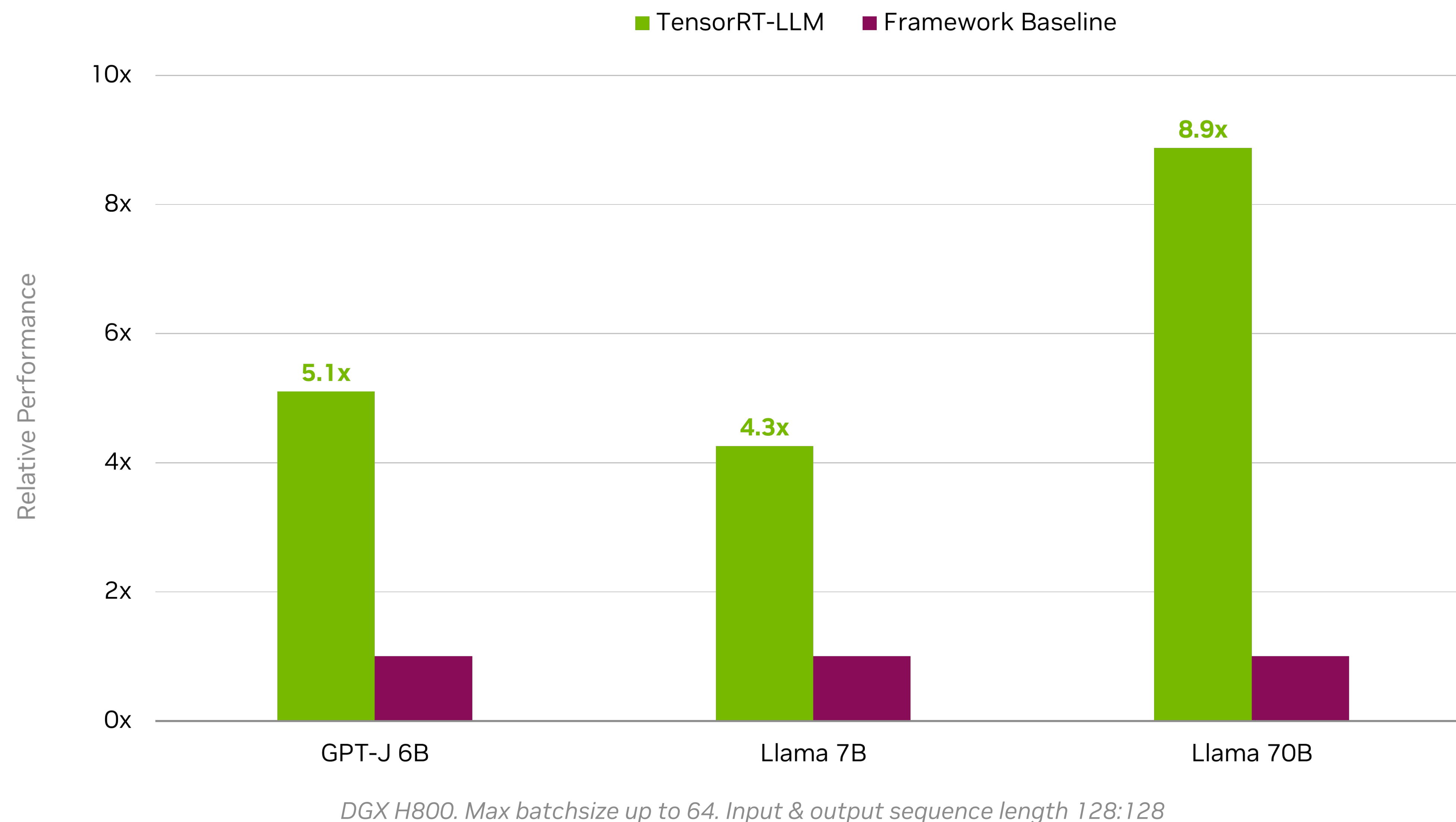
当batch_size * num_heads不够大（与流式多处理器的数量相比，CUDA线程块的数量较少）时，无法充分利用GPU，此时Multi-Block Mode可能是有益的。该模式预计将减少Multi Head Attention Kernel在生成阶段的延迟。需要注意的是，它要求上下文长度足够长，以便每个CUDA线程块执行的工作保持足够的效率。

详见[文档](#)。

TensorRT-LLM优化效果

与其他开源框架相比

我们选择在TensorRT-LLM中使用FP8量化后的模型，来对比HuggingFace的原始FP16模型，在测试模型上取得了最高8.9倍的性能加速，如下图所示：



以Llama 70B的测试为例，TensorRt-LLM采用tp_size=2，而HuggingFace采用pp_size=4。由于采用了FP8量化，所需显存减半，使用的GPU数量得以减半。

综上，采用TensorRT-LLM的包括FP8量化在内的一系列优化后，做到了降本增效的目标。



Thanks!