fp8推理调研报告

1. fp16->fp8推理过程中转换的方法、可行性、scale还是截断、效果

fp16转fp8方法

参考：大模型量化技术原理：FP8

FP16转FP8方法

- autoFP8: 开源FP8量化库，生成可在vLLM中运行的压缩检查点，为FP8_E4M3精度提供量化的权重、激活和KV缓存尺度。
- flux-fp8-api: 实现了使用量化的fp8矩阵乘法和更快的半精度累积的Flux扩散模型，提供API端点。
- FP8-Quantization: 实现了FP8量化。
- neural-compressor (NNCF): 优化神经网络推理的算法，提供多种压缩算法。
- tensorRT-Model-Optimizer: 库，包含了最先进的模型优化技术，用于压缩深度学习模型，支持FP8或INT8量化的扩散模型。

FP8 Quantization

FP32 / scale = Unscaled FP32

FP8 = convert(Unscaled FP32)

Neural Compressor

NNCF 是一套高级算法，用于优化英特尔® Distribution of OpenVINO ™ 工具包中的机器学习和深度学习模型以进行推理。NNCF 使用来自 PyTorch 和 TensorFlow 的模型。

高通AI研究院 FP8-quantization

来自高通AI研究院的论文 FP8 Quantization: The Power of the Exponent 通过对FP8量化格式的深入分析，包括理论基础和实验验证。提出了一种一种在 FP32 硬件中模拟 FP8 量化的新方法，，该方法可加快 FP8 量化模拟速度，同时很容易地在常见的深度学习框架中实现，有助于快速进行PTQ和QAT，并且它暴露了FP8量化器的参数（即尾数/指数位和指数偏置值），允许通过反向传播学习这些参数。

最后得出FP8格式在多种网络模型的 PTQ 中通常优于INT8，5M2E 和 4M3E FP8 格式效果最好，而对于具有更多异常值（例如：Transformer）的神经网络模型，增加指数位数效果最好。同时，还表明在进行量化感知训练时，该格式的许多优点都会消失，因为网络模型学会了在 INT8 量化中也表现良好。

上海交通大学、北京大学和微软亚洲研究院联合发布 Integer or Floating Point? New Outlooks for Low-Bit Quantization on Large Language Models

MoFQ

对INT和FP量化进行了比较分析，发现不同层的最佳量化格式因张量分布的复杂性和多样性而异，没有一种量化格式在所有情况下都始终优于其他量化格式，从而提出了一种混合格式量化方法（Mixture of Formats Quantization (MoFQ)）。该方法逐层（layer-wise）选择最优的量化格式，以最小的精度损失实现 W8A8 量化结果。无论是仅权重还是权重激活量化场景中都取得了良好的效果。

在 8 比特的权重激活量化中，MoFQ 显著优于仅 INT/FP 的方法，实现了接近全精度模型的性能。这表明 MoFQ8 有效地为每一层的分布选择最合适的格式（INT8 或 FP8），最终以最小的精度损失实现 W8A8 量化结果。值得注意的是，与仅 INT/FP 量化相比，MoFQ 不会产生硬件开销，因为位宽保持不变。

Abstract—Efficient deployment of Large Language Models (LLMs) requires low-bit quantization to reduce model size and inference cost. Besides low-bit integer formats (e.g., INT8/INT4) used in previous quantization works, emerging low-bit floating-point formats (e.g., FP8/FP4) supported by advanced hardware like NVIDIA's H100 GPU offer an alternative. Our study finds that introducing floating-point formats significantly improves LLMs quantization. We also discover that the optimal quantization format varies across layers. Therefore, we select the optimal format for each layer, which we call the Mixture of Formats Quantization (MoFQ) method. Our MoFQ method achieves better or comparable results over current methods in weight-only (W-only) and weight-activation (WA) post-training quantization scenarios across various tasks, with no additional hardware overhead.

Index Terms—Large Language Model, Low-bit Quantization, Floating Point

Table 4: WA-Quantization results On WikiText-2, LAMBADA, PIQA and HellaSwag datasets. For WikiText-2 dataset, we show perplexity metric. For the other three, we show average accuracy.

	WikiText-2 ↓				LAMBADA ↑			
	FP16	INT8	FP8	MoFQ8 (FP%)	FP16	INT8	FP8	MoFQ8 (FP%)
LLaMA-7B	5.68	368.21	6.59	**6.49(87.2%)**	0.884	0.010	0.851	**0.887(82.0%)**
LLaMA-13B	5.09	637.95	5.64	**5.41(86.1%)**	0.883	0.230	0.854	**0.881(83.0%)**
LLaMA-33B	4.10	10069.14	5.38	**5.31(92.7%)**	0.862	0.000	0.822	**0.859(90.0%)**
OPT-350M	23.27	432.86	24.46	**23.64(71.8%)**	0.674	0.290	0.658	**0.669(69.2%)**
OPT-1.3B	15.44	37.72	16.78	**16.07(78.8%)**	0.758	0.716	0.735	**0.746(80.0%)**
OPT-2.7B	13.08	27.56	14.24	**13.25(83.3%)**	0.778	0.693	0.764	**0.777(80.0%)**
OPT-6.7B	11.43	964.58	12.41	**11.68(89.1%)**	0.806	0.164	0.762	**0.800(89.4%)**
OPT-13B	10.68	11858.78	12.52	**10.79(87.2%)**	0.802	0.001	0.724	**0.801(84.1%)**
OPT-30B	10.09	13195.34	10.95	**10.17(89.4%)**	0.813	0.007	0.744	**0.812(86.0%)**

	PIQA ↑				HellaSwag ↑			
	FP16	INT8	FP8	MoFQ8 (FP%)	FP16	INT8	FP8	MoFQ8 (FP%)
LLaMA-7B	0.780	0.539	0.706	**0.779(85.8%)**	0.558	0.258	0.524	**0.560(80.6%)**
LLaMA-13B	0.783	0.532	0.768	**0.788(82.3%)**	0.587	0.264	0.576	**0.585(81.7%)**
LLaMA-33B	0.787	0.530	0.767	**0.789(81.3%)**	0.605	0.260	0.599	**0.604(89.1%)**
OPT-350M	0.619	0.554	0.615	**0.621(71.9%)**	0.292	0.265	**0.295**	0.293(73.4%)
OPT-1.3B	0.693	0.667	0.690	**0.691(75.2%)**	0.351	0.351	0.351	**0.351(80.8%)**
OPT-2.7B	0.708	0.687	0.712	**0.714(75.1%)**	0.379	0.383	0.393	**0.396(80.3%)**
OPT-6.7B	0.721	0.609	0.718	**0.720(87.0%)**	0.409	0.279	0.407	**0.411(84.4%)**
OPT-13B	0.716	0.516	0.688	**0.715(82.9%)**	0.421	0.263	0.417	**0.426(85.0%)**
OPT-30B	0.725	0.524	0.713	**0.727(80.5%)**	0.442	0.260	0.433	**0.442(85.5%)**

fp8推理

FP8推理流程

训练时为确保梯度计算准确，权重通常维持为高精度（如：BF16 或 FP32），这是由于训练时需更新参数，而在推理时，权重已固定，故可在模型加载或预处理理阶段提前将权重转换为 FP8，确保模型加载即为 FP8 格式。

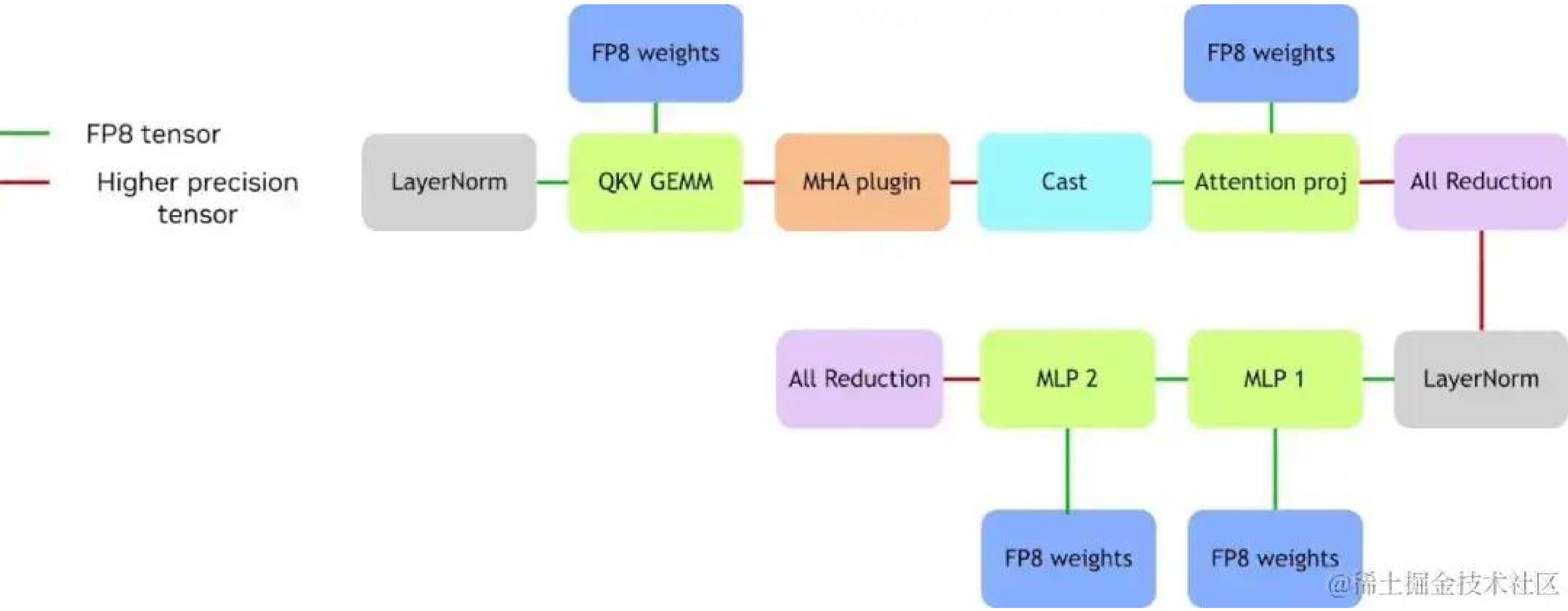
此外，推理阶段应尽量进行操作融合，如将 LayerNorm 与后续数据格式转换操作整合，确保 kernel 输入输出尽可能维持 FP8，从而能够有效提升 GPU 内存吞吐。同样，GeLU (Gaussian Error Linear Unit) 激活函数也要力求融合。

目前少量输出仍会保持为 FP16，原因是 NVIDIA NCCL 仅支持高精度reduce操作，所以现在仍然需采用 FP16 进行 reduce，完成后再转化为 FP8。

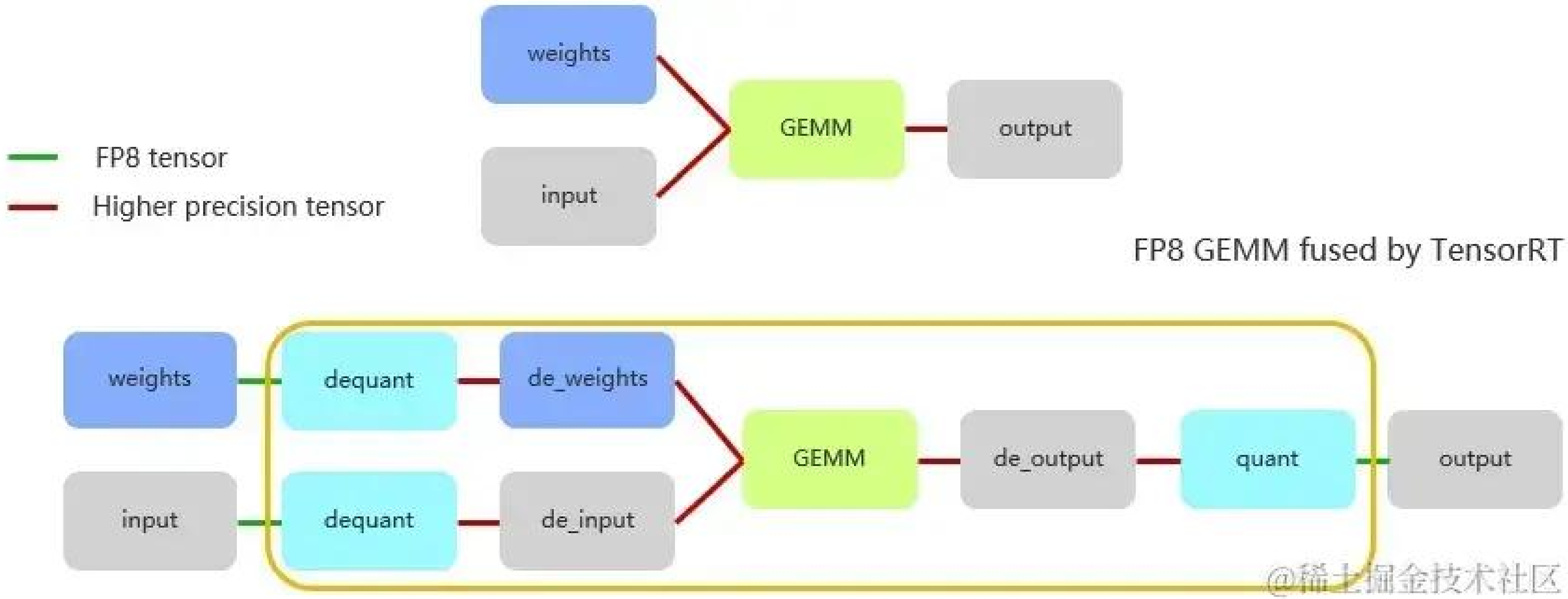
推理流程就简化为下图所示。绿线代表 FP8 的输入输出（I/O），红线表示高精度 I/O。图中可见，最前端的 LayerNorm 输出与权重均为 FP8，矩阵输出暂时保持 FP16，与前文描述一致。并且经过测试验证可得，虽然矩阵输出精度对整体性能影响较小，但与输入问题的规模相关；且因其计算密集的特性，对输出形态影响微弱。

在完成 MHA 后，需要将结果转换为 FP8 以进行后续矩阵计算，Reduce 是以 FP16 执行后再转换到 FP8 的。对于 MLP1 和 MLP2，两者逻辑相似，但不同之处在于：MLP1 的输出可保持在 FP8，因为它已经把 GeLU 加 Bias 等操作直接融合到 MLP1 的 kernel。

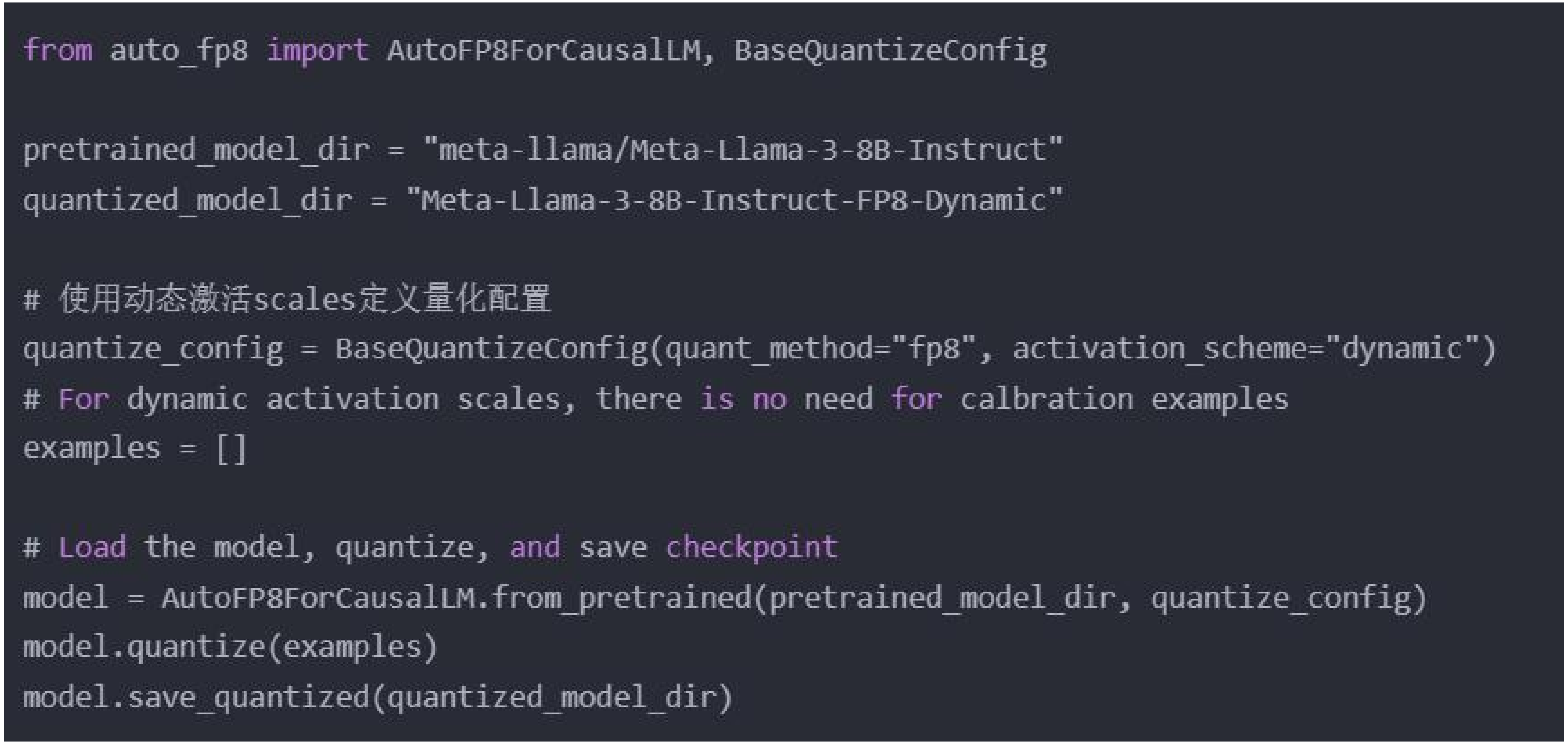
FP8推理


FP8 tensor
Higher precision tensor
LayerNorm
FP8 weights
QKV GEMM
MHA plugin
Cast
FP8 weights
Attention proj
All Reduction
LayerNorm
MLP 1
MLP 2
All Reduction
FP8 weights
FP8 weights

FP8在tensorRT_LLM的应用

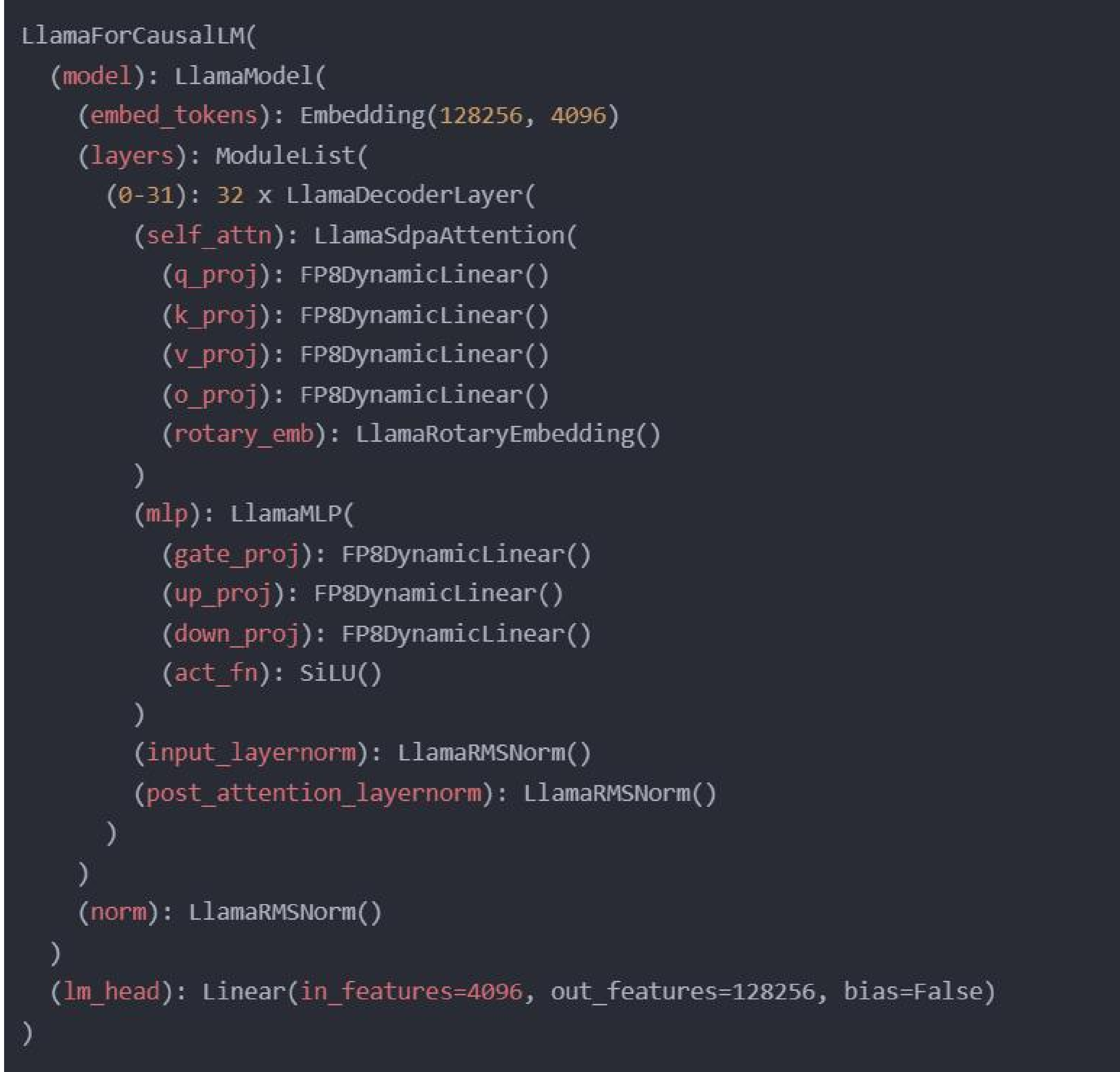


AutoFP8 离线动态量化



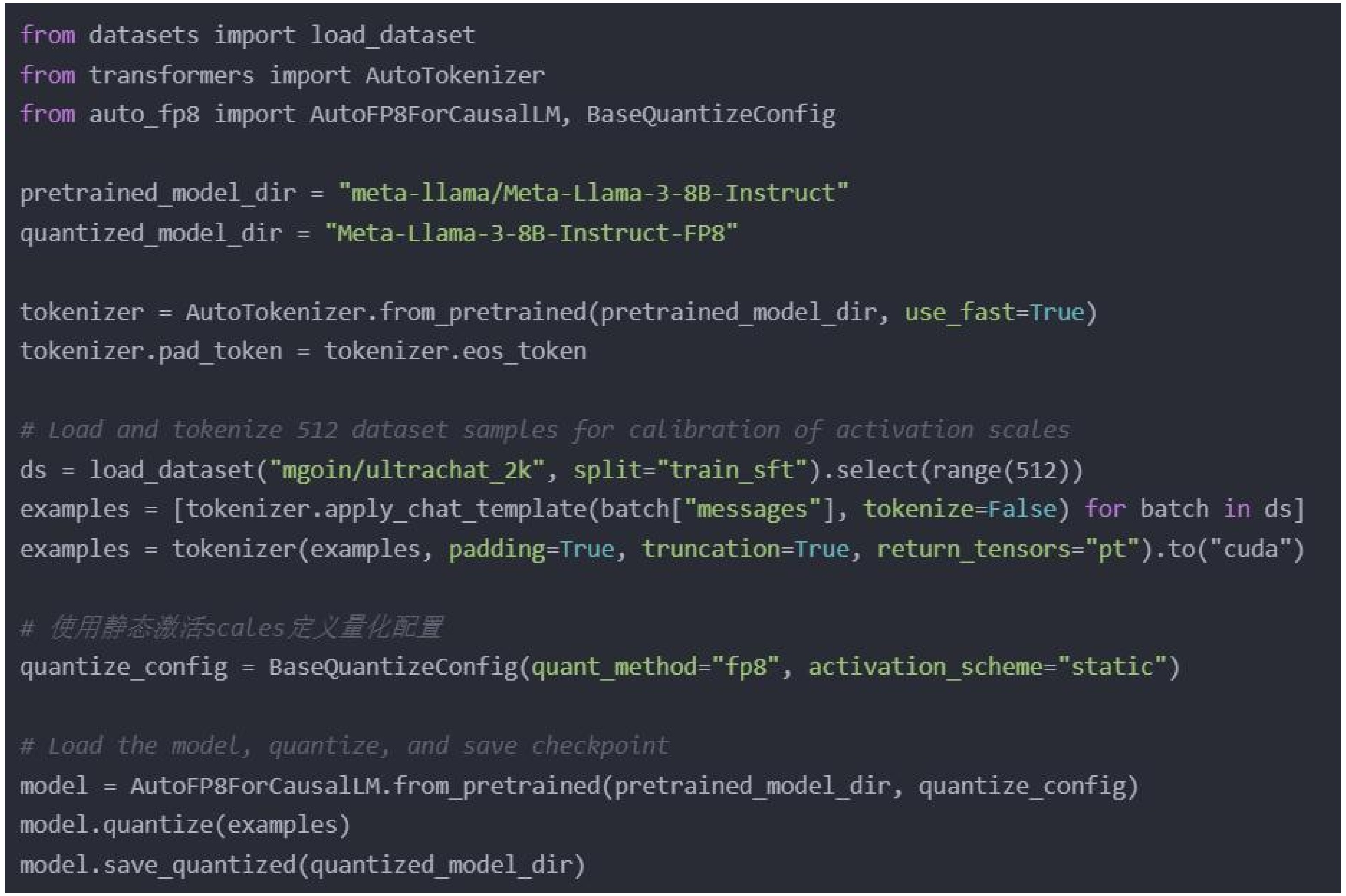
```python
from auto_fp8 import AutoFP8ForCausalLM, BaseQuantizeConfig

pretrained_model_dir = "meta-llama/Meta-Llama-3-8B-Instruct"
quantized_model_dir = "Meta-Llama-3-8B-Instruct-FP8-Dynamic"

# 使用动态激活scales定义量化配置
quantize_config = BaseQuantizeConfig(quant_method="fp8", activation_scheme="dynamic")
# For dynamic activation scales, there is no need for calbration examples
examples = []

# Load the model, quantize, and save checkpoint
model = AutoFP8ForCausalLM.from_pretrained(pretrained_model_dir, quantize_config)
model.quantize(examples)
model.save_quantized(quantized_model_dir)
```

量化网络结构



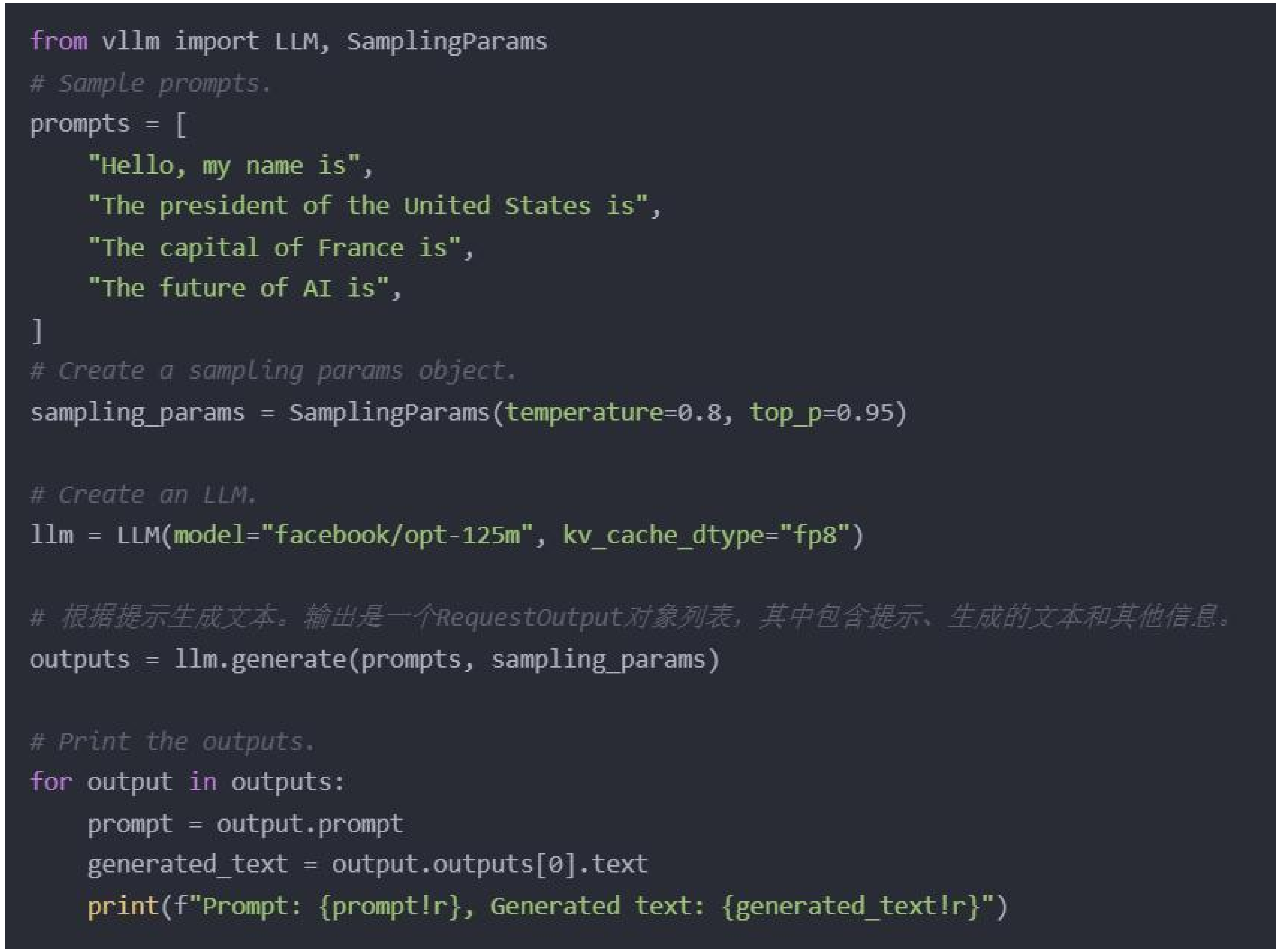
```
LlamaForCausalLM(
  (model): LlamaModel(
    (embed_tokens): Embedding(128256, 4096)
    (layers): ModuleList(
      (0-31): 32 x LlamaDecoderLayer(
        (self_attn): LlamaSdpaAttention(
          (q_proj): FP8DynamicLinear()
          (k_proj): FP8DynamicLinear()
          (v_proj): FP8DynamicLinear()
          (o_proj): FP8DynamicLinear()
          (rotary_emb): LlamaRotaryEmbedding()
        )
        (mlp): LlamaMLP(
          (gate_proj): FP8DynamicLinear()
          (up_proj): FP8DynamicLinear()
          (down_proj): FP8DynamicLinear()
          (act_fn): SiLU()
        )
        (input_layernorm): LlamaRMSNorm()
        (post_attention_layernorm): LlamaRMSNorm()
      )
    )
    (norm): LlamaRMSNorm()
  )
  (lm_head): Linear(in_features=4096, out_features=128256, bias=False)
)
```

AutoFP8 离线静态量化



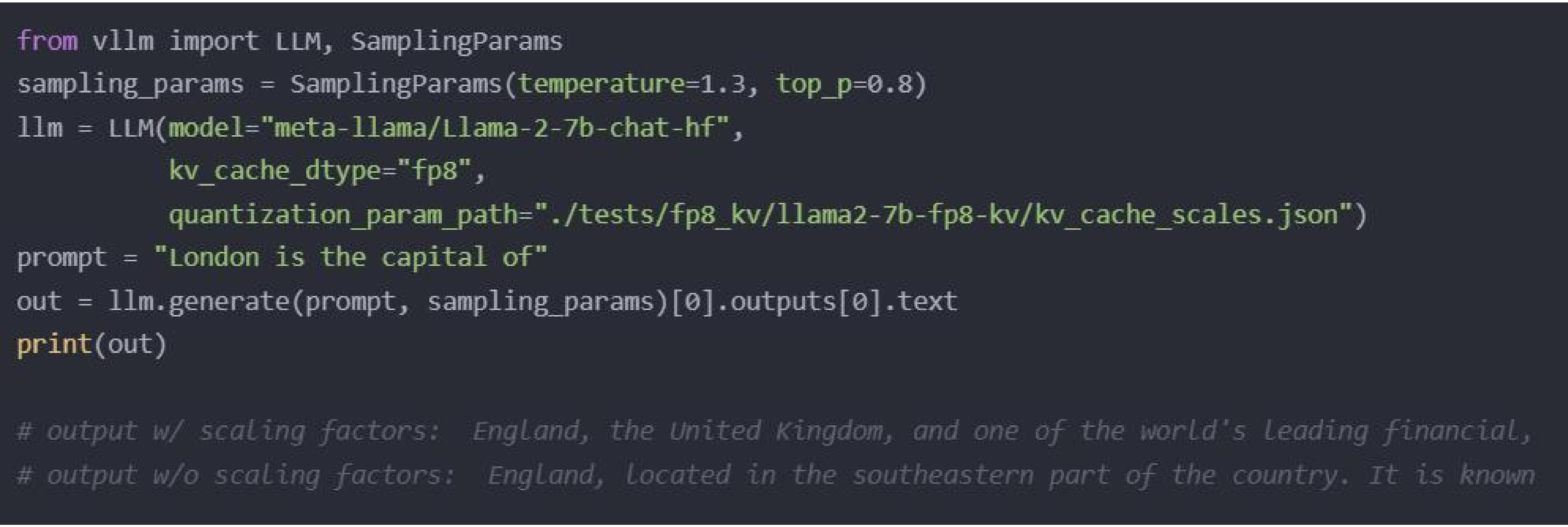
```python
from datasets import load_dataset
from transformers import AutoTokenizer
from auto_fp8 import AutoFP8ForCausalLM, BaseQuantizeConfig

pretrained_model_dir = "meta-llama/Meta-Llama-3-8B-Instruct"
quantized_model_dir = "Meta-Llama-3-8B-Instruct-FP8"

tokenizer = AutoTokenizer.from_pretrained(pretrained_model_dir, use_fast=True)
tokenizer.pad_token = tokenizer.eos_token

# Load and tokenize 512 dataset samples for calibration of activation scales
ds = load_dataset("mgoin/ultrachat_2k", split="train_sft").select(range(512))
examples = [tokenizer.apply_chat_template(batch["messages"], tokenize=False) for batch in ds]
examples = tokenizer(examples, padding=True, truncation=True, return_tensors="pt").to("cuda")

# 使用静态激活scales定义量化配置
quantize_config = BaseQuantizeConfig(quant_method="fp8", activation_scheme="static")

# Load the model, quantize, and save checkpoint
model = AutoFP8ForCausalLM.from_pretrained(pretrained_model_dir, quantize_config)
model.quantize(examples)
model.save_quantized(quantized_model_dir)
```

KV-Cache FP8 (E5M2)

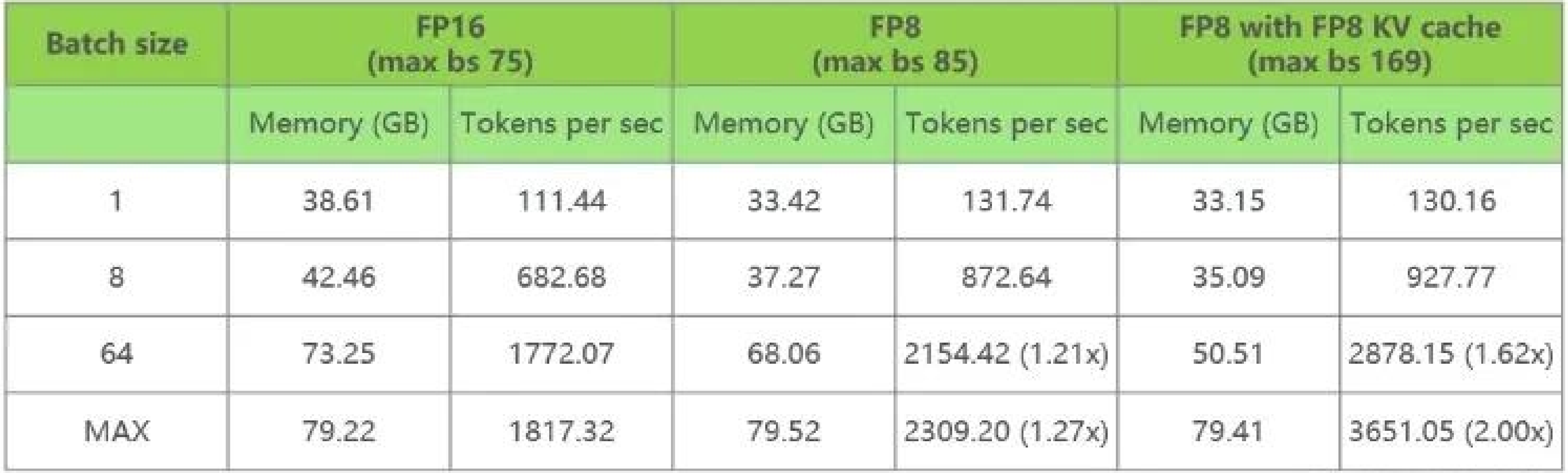


```python
from vllm import LLM, SamplingParams
# Sample prompts.
prompts = [
    "Hello, my name is",
    "The president of the United States is",
    "The capital of France is",
    "The future of AI is",
]
# Create a sampling params object.
sampling_params = SamplingParams(temperature=0.8, top_p=0.95)

# Create an LLM.
llm = LLM(model="facebook/opt-125m", kv_cache_dtype="fp8")

# 根据提示生成文本。输出是一个RequestOutput对象列表，其中包含提示、生成的文本和其他信息。
outputs = llm.generate(prompts, sampling_params)

# Print the outputs.
for output in outputs:
    prompt = output.prompt
    generated_text = output.outputs[0].text
    print(f"Prompt: {prompt!r}, Generated text: {generated_text!r}")
```

KV-Cache FP8 (E4M3)



```python
from vllm import LLM, SamplingParams
sampling_params = SamplingParams(temperature=1.3, top_p=0.8)
llm = LLM(model="meta-llama/Llama-2-7b-chat-hf",
          kv_cache_dtype="fp8",
          quantization_param_path="./tests/fp8_kv/llama2-7b-fp8-kv/kv_cache_scales.json")
prompt = "London is the capital of"
out = llm.generate(prompt, sampling_params)[0].outputs[0].text
print(out)

# output w/ scaling factors:  England, the United Kingdom, and one of the world's leading financial,
# output w/o scaling factors:  England, located in the southeastern part of the country. It is known
```



Batch size	FP16 (max bs 75)		FP8 (max bs 85)		FP8 with FP8 KV cache (max bs 169)	
	Memory (GB)	Tokens per sec	Memory (GB)	Tokens per sec	Memory (GB)	Tokens per sec
1	38.61	111.44	33.42	131.74	33.15	130.16
8	42.46	682.68	37.27	872.64	35.09	927.77
64	73.25	1772.07	68.06	2154.42 (1.21x)	50.51	2878.15 (1.62x)
MAX	79.22	1817.32	79.52	2309.20 (1.27x)	79.41	3651.05 (2.00x)

GPT-J 6b, input length 1024, output length 256, CUDA 12.2, Driver 535.104.05, TensorRT 9.1 on *H100

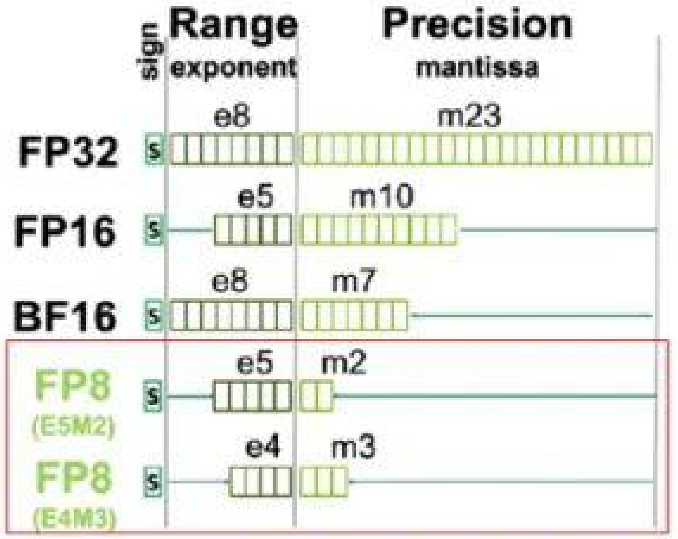
2. FP8推理的运用的技术、操作融合、过程、效果

NVIDIA量化

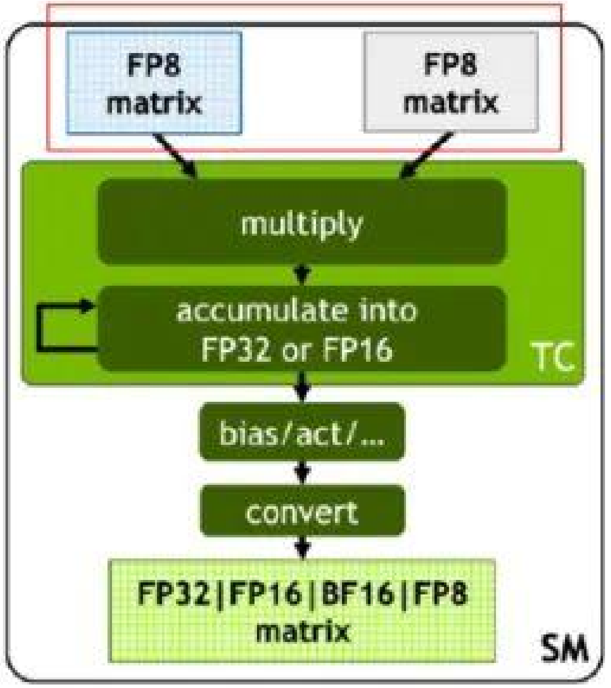
参考：TensorRT-LLM低精度推理优化
https://www.elecfans.com/d/6356149.html

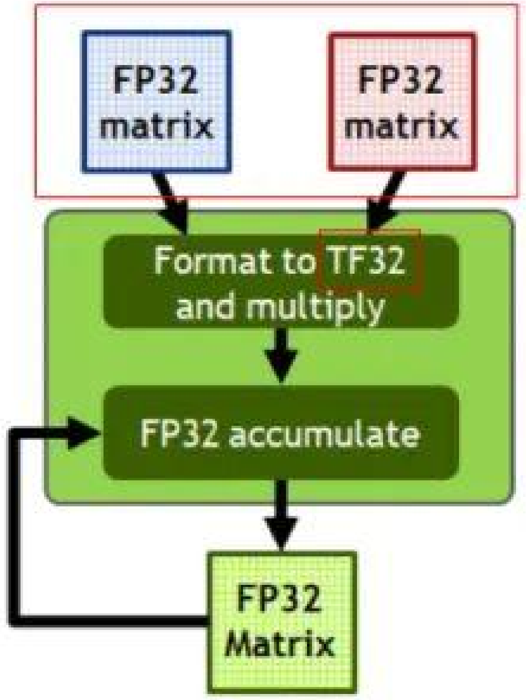
What is FP8

FP8 precisions



Two FP8 formats



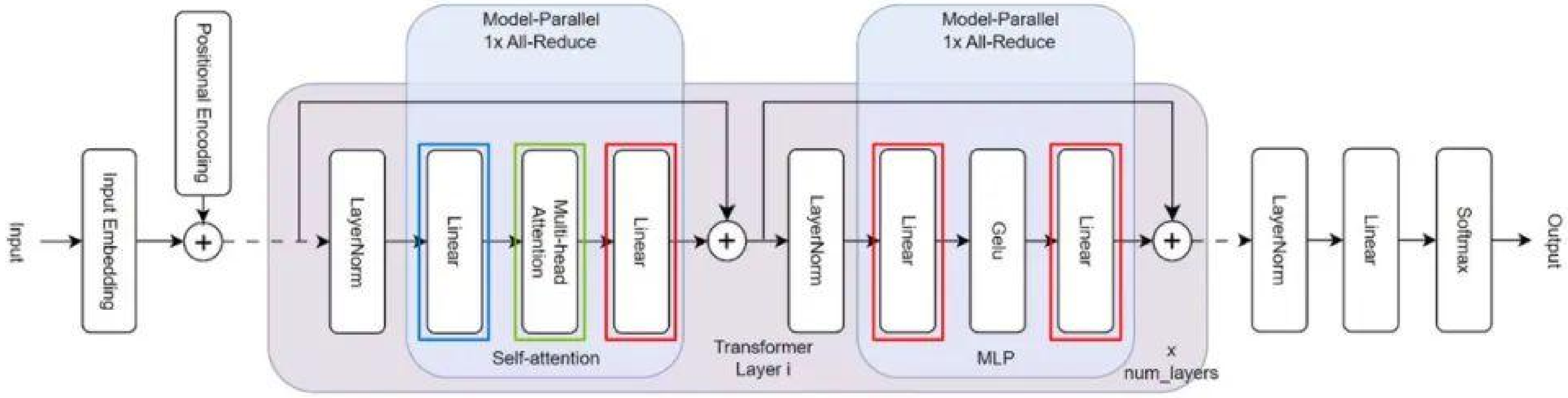


Unlike TF32 which is only adopted in computing process.
FP8 is adopted both in computing and input.

Compared with int8: better resolution for non-outlier, per-tensor quantization could keep the accuracy

What ops do we quant

PTQ (Post Training Quantization)



1. QKV + MMHA : activation and weight + output of QKV GEMM for kvcache
2. FMHA : scaling factor=1
3. OTHER GEMM : activation and weight

Workflow in TensorRT-LLM

How to use

What can you use?

- TRTLLM 0.9.0 and AMMO 0.7.3 PKG
- TRTLLM main and AMMO 0.9.3 PKG

Step 1 : gen scaling factor

- python ../quantization/quantize.py --model_dir ./tmp/llama/70B --dtype float16 --qformat fp8 --kv_cache_dtype fp8 --output_dir ./tllm_checkpoint_2gpu_fp8 --calib_size 512 --tp_size 2

Step 2 : gen the fp8 engine

- trtllm-build --checkpoint_dir ./tllm_checkpoint_2gpu_fp8 --output_dir ./engine_outputs --gemm_plugin float16 --strongly_typed --use_fp8_context_fmha enable --workers 2

Step 3 : eval

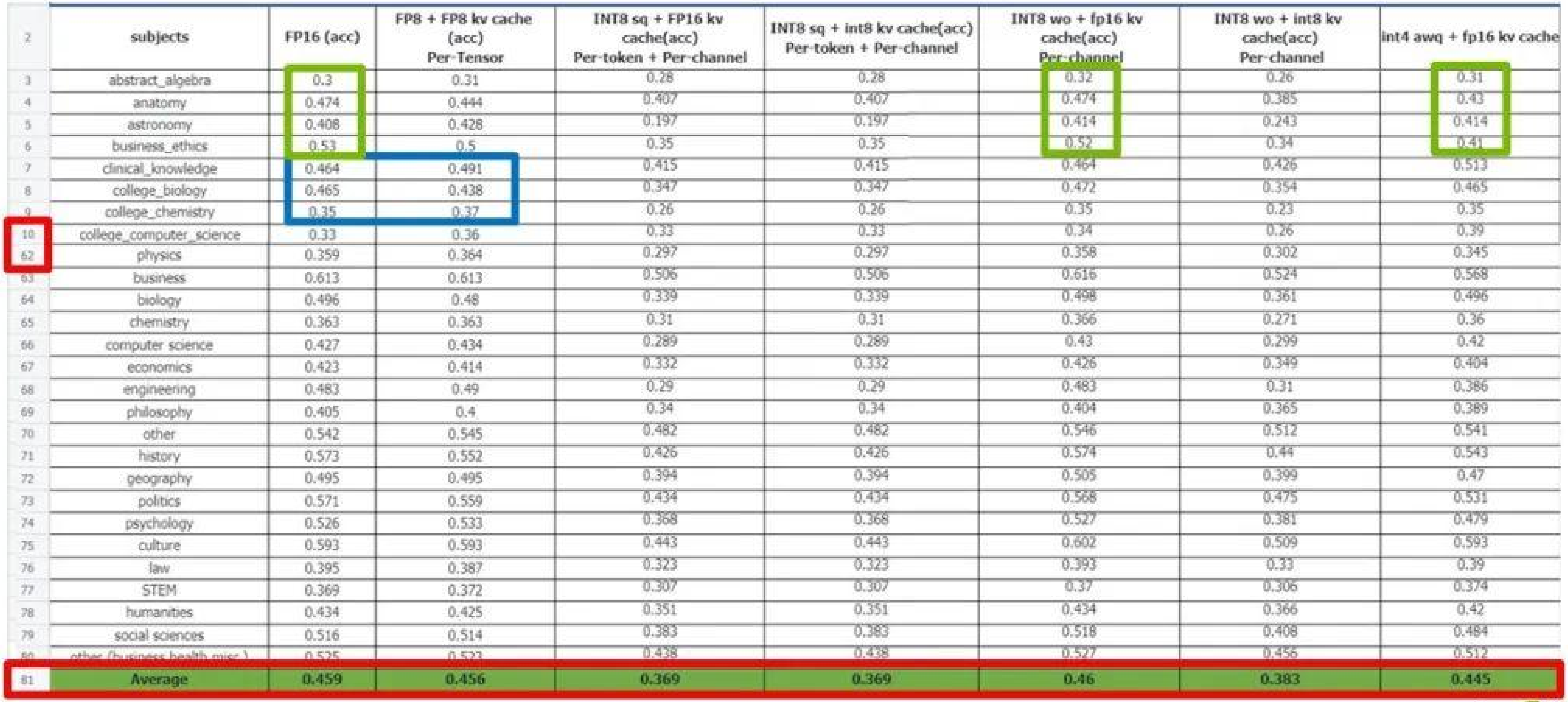
- mpirun -n 2 --allow-run-as-root python ../summarize.py --test_trt_llm --hf_model_dir ./tmp/llama/7B/ --data_type fp16 --engine_dir ./tmp/llama/7B/trt_engines/fp8/2-gpu/

If you want to support a new model, just realize step 1 and step 2.

Perf

accuracy

FP8: GEMM+KVCACHE

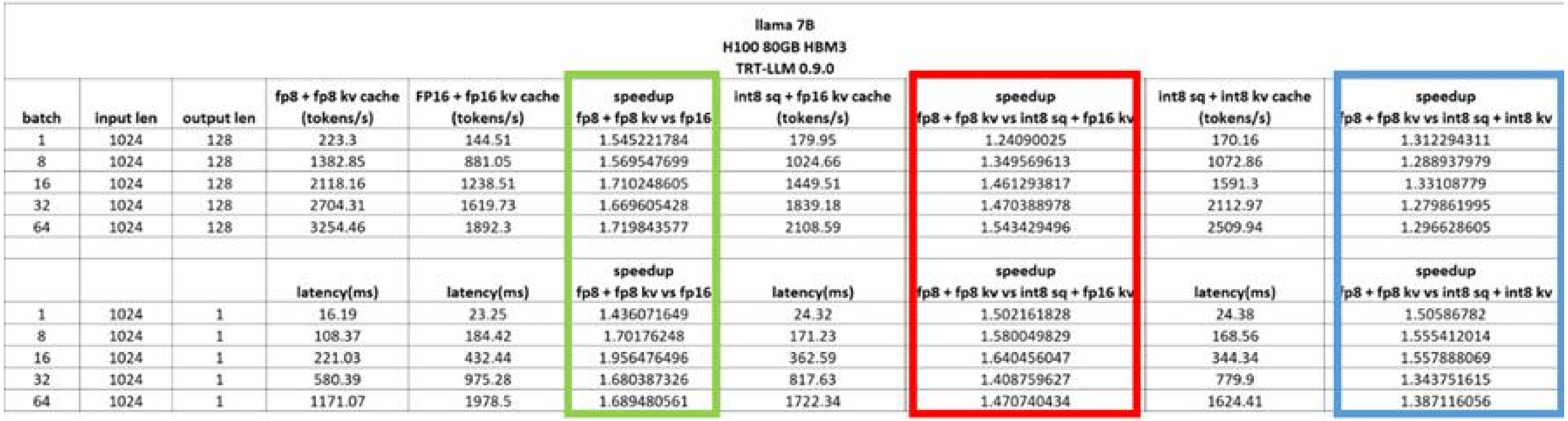


2	subjects	FP16 (acc)	FP8 + FP8 kv cache (acc) Per-Tensor	INT8 sq + FP16 kv cache(acc) Per-token + Per-channel	INT8 sq + int8 kv cache(acc) Per-token + Per-channel	INT8 wo + fp16 kv cache(acc) Per-channel	INT8 wo + int8 kv cache(acc) Per-channel	int4 awq + fp16 kv cache
3	abstract_algebra	0.3	0.31	0.28	0.28	0.32	0.26	0.31
4	anatomy	0.474	0.444	0.407	0.407	0.474	0.385	0.43
5	astronomy	0.408	0.428	0.197	0.197	0.414	0.243	0.414
6	business_ethics	0.53	0.5	0.35	0.35	0.52	0.34	0.41
7	clinical_knowledge	0.464	0.491	0.415	0.415	0.464	0.426	0.513
8	college_biology	0.465	0.438	0.347	0.347	0.472	0.354	0.465
9	college_chemistry	0.35	0.37	0.26	0.26	0.35	0.23	0.35
10	college_computer_science	0.33	0.36	0.33	0.33	0.34	0.26	0.39
62	physics	0.359	0.364	0.297	0.297	0.358	0.302	0.345
63	business	0.613	0.613	0.506	0.506	0.616	0.524	0.568
64	biology	0.496	0.48	0.339	0.339	0.498	0.361	0.496
65	chemistry	0.363	0.363	0.31	0.31	0.366	0.271	0.36
66	computer science	0.427	0.434	0.289	0.289	0.43	0.299	0.42
67	economics	0.423	0.414	0.332	0.332	0.426	0.349	0.404
68	engineering	0.483	0.49	0.29	0.29	0.483	0.31	0.386
69	philosophy	0.405	0.4	0.34	0.34	0.404	0.365	0.389
70	other	0.542	0.545	0.482	0.482	0.546	0.512	0.541
71	history	0.573	0.552	0.426	0.426	0.574	0.44	0.543
72	geography	0.495	0.495	0.394	0.394	0.505	0.399	0.47
73	politics	0.571	0.559	0.434	0.434	0.568	0.475	0.531
74	psychology	0.526	0.533	0.368	0.368	0.527	0.381	0.479
75	culture	0.593	0.593	0.443	0.443	0.602	0.509	0.593
76	law	0.395	0.387	0.323	0.323	0.393	0.33	0.39
77	STEM	0.369	0.372	0.307	0.307	0.37	0.306	0.374
78	humanities	0.434	0.425	0.351	0.351	0.434	0.366	0.42
79	social sciences	0.516	0.514	0.383	0.383	0.518	0.408	0.484
80	other (business health misc.)	0.525	0.523	0.438	0.438	0.527	0.456	0.512
81	**Average**	**0.459**	**0.456**	**0.369**	**0.369**	**0.46**	**0.383**	**0.445**

Perf

FP8: GEMM+KVCACHE

Speedup between FP8 and INT8



llama 7B
H100 80GB HBM3
TRT-LLM 0.9.0

batch	input len	output len	fp8 + fp8 kv cache (tokens/s)	FP16 + fp16 kv cache (tokens/s)	speedup fp8 + fp8 kv vs fp16	int8 sq + fp16 kv cache (tokens/s)	speedup fp8 + fp8 kv vs int8 sq + fp16 kv	int8 sq + int8 kv cache (tokens/s)	speedup fp8 + fp8 kv vs int8 sq + int8 kv
1	1024	128	223.3	144.51	1.545221784	179.95	1.24090025	170.16	1.312294311
8	1024	128	1382.85	881.05	1.569547699	1024.66	1.349569613	1072.86	1.288937979
16	1024	128	2118.16	1238.51	1.710248605	1449.51	1.461293817	1591.3	1.33108779
32	1024	128	2704.31	1619.73	1.669605428	1839.18	1.470388978	2112.97	1.279861995
64	1024	128	3254.46	1892.3	1.719843577	2108.59	1.543429496	2509.94	1.296628605
			latency(ms)	latency(ms)	speedup fp8 + fp8 kv vs fp16	latency(ms)	speedup fp8 + fp8 kv vs int8 sq + fp16 kv	latency(ms)	speedup fp8 + fp8 kv vs int8 sq + int8 kv
1	1024	1	16.19	23.25	1.436071649	24.32	1.502161828	24.38	1.50586782
8	1024	1	108.37	184.42	1.70176248	171.23	1.580049829	168.56	1.555412014
16	1024	1	221.03	432.44	1.956476496	362.59	1.640456047	344.34	1.557888069
32	1024	1	580.39	975.28	1.680387326	817.63	1.408759627	779.9	1.343751615
64	1024	1	1171.07	1978.5	1.689480561	1722.34	1.470740434	1624.41	1.387116056

Perf

accuracy

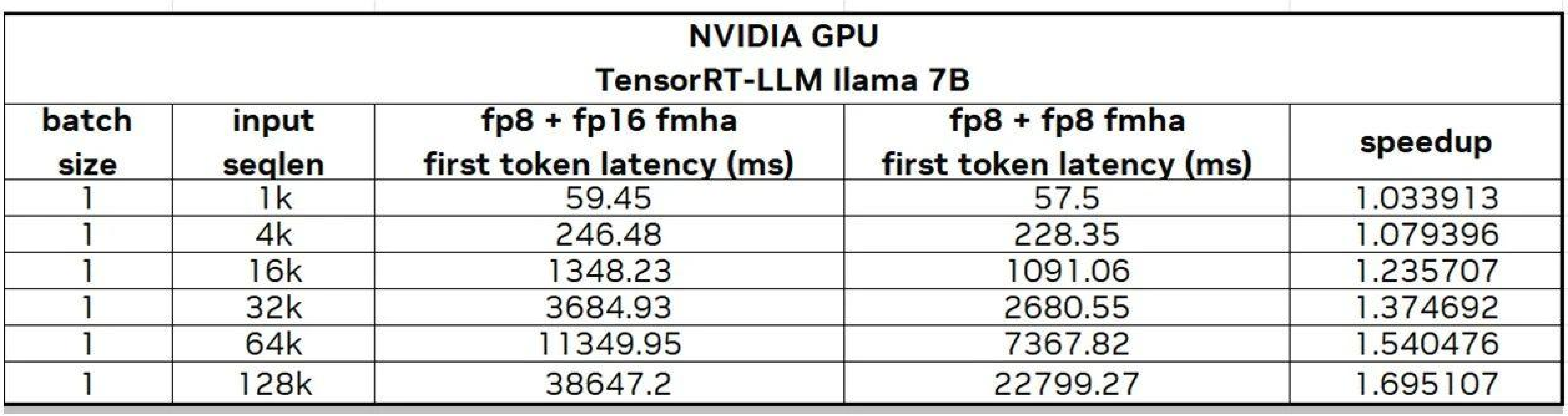
FP8: GEMM+KVCACHE+FMHA

llama v2 7b fp8 work flow				
mmlu subject	average accuracy (fp16 fmha)	average accuracy (fp8 fmha)	Accuracy change	Total number
math	0.29	0.295	1.72%	1064
health	0.482	0.485	0.62%	1640
physics	0.35	0.373	6.57%	640
business	0.618	0.611	-1.13%	437
biology	0.493	0.496	0.61%	454
chemistry	0.376	0.373	-0.80%	303
computer science	0.437	0.427	-2.29%	412
economics	0.415	0.41	-1.20%	742
engineering	0.497	0.455	-8.45%	145
philosophy	0.409	0.405	-0.98%	2012
other	0.543	0.53	-2.39%	1165
history	0.554	0.553	-0.18%	930
geography	0.5	0.52	4.00%	198
politics	0.577	0.56	-2.95%	648
psychology	0.53	0.514	-3.02%	1157
culture	0.596	0.611	2.52%	332
law	0.402	0.389	-3.23%	1763
STEM	0.372	0.375	0.81%	3018
humanities	0.435	0.428	-1.61%	4705
social sciences	0.517	0.51	-1.35%	3077
other (business, health, misc.)	0.522	0.518	-0.77%	3242
Average	**0.46**	**0.455**	**-1.09%**	14042

Perf

Speedup between FP16 and FP8

FP8: GEMM+KVCACHE+FMHA



NVIDIA GPU TensorRT-LLM llama 7B				
batch size	**input seqlen**	**fp8 + fp16 fmha first token latency (ms)**	**fp8 + fp8 fmha first token latency (ms)**	**speedup**
1	1k	59.45	57.5	1.033913
1	4k	246.48	228.35	1.079396
1	16k	1348.23	1091.06	1.235707
1	32k	3684.93	2680.55	1.374692
1	64k	11349.95	7367.82	1.540476
1	128k	38647.2	22799.27	1.695107

Perf

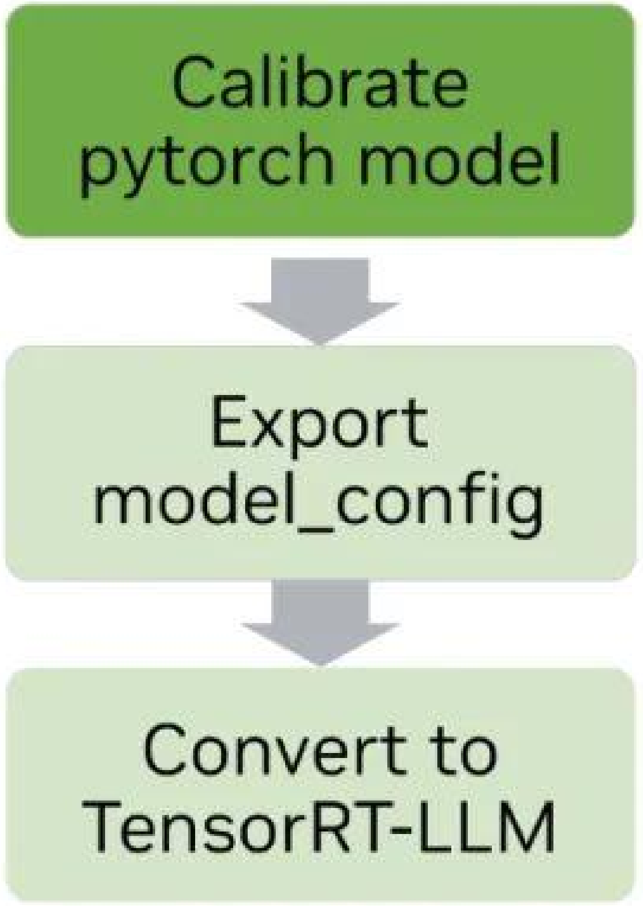
Cost time of calibration on CNNDaily

qwen-7b	total calib time(s)	total calib num	bs	avg time(s)
	81.5	1024	1	0.079589844
	40.5	512	1	0.079101563
	5.4	64	1	0.084375
H100	1.6	64	8	0.025
	211.8	1024	1	0.206835938
	103	512	1	0.201171875
	12.8	64	1	0.2
A10	8.7	64	8	0.1359375

Workflow

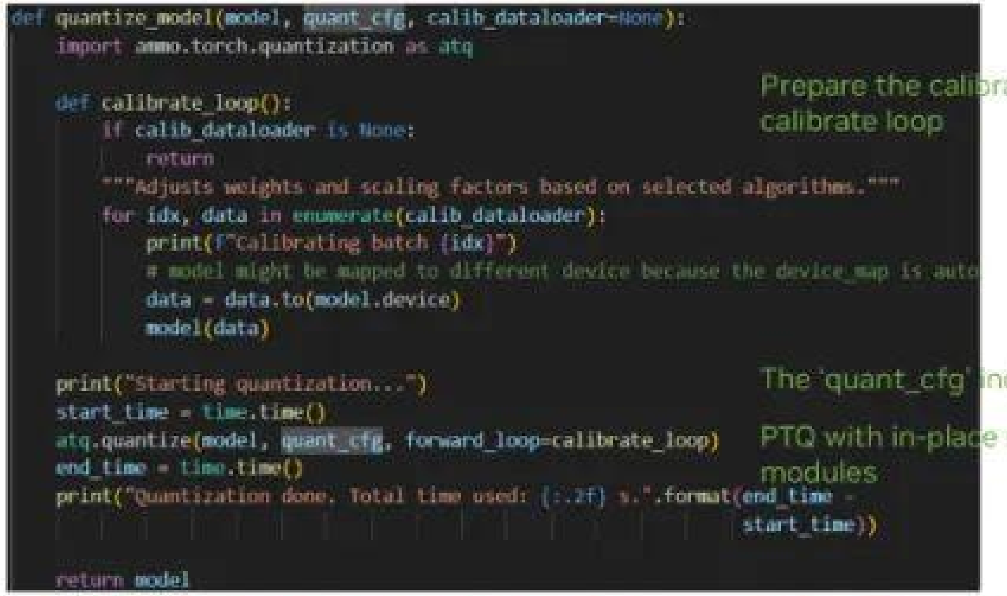
PTQ (Post Training Quantization) – Scaling Factor

step 1 : gen scaling factor



PTQ can be achieved with simple calibration on a small set of training or evaluation data (typically 128-512 samples) after converting a regular PyTorch(HF or NeMo) model to a quantized model.

- Config:FP8_CFG
- Torch model + calibrate loop -> quDEFAULT_CFG or INT8_SMOOTHQUANTantize model,



```python
def quantize_model(model, quant_cfg, calib_dataloader=None):
    import ammo.torch.quantization as atq

    def calibrate_loop():
        if calib_dataloader is None:
            return
        """Adjusts weights and scaling factors based on selected algorithms."""
        for idx, data in enumerate(calib_dataloader):
            print(f"Calibrating batch {idx}")
            # model might be mapped to different device because the device_map is auto
            data = data.to(model.device)
            model(data)

    print("Starting quantization...")
    start_time = time.time()
    atq.quantize(model, quant_cfg, forward_loop=calibrate_loop)
    end_time = time.time()
    print("Quantization done. Total time used: {:.2f} s.".format(end_time -
                                                                   start_time))

    return model
```

Prepare the calibration set and define a calibrate loop

The 'quant_cfg' includes KV_CACHE and GEMM

PTQ with in-place replacement to quantized modules

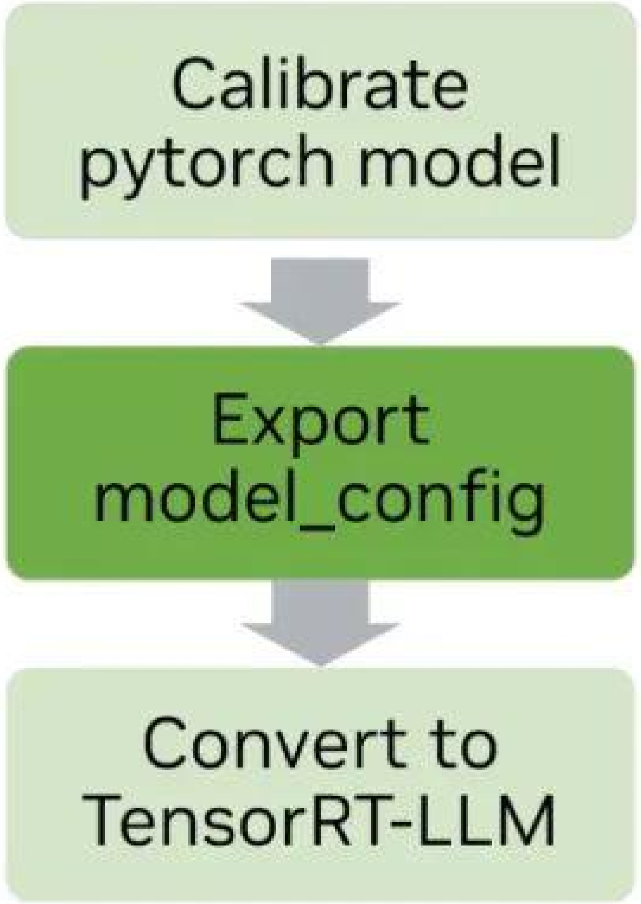
AMMO/Modelopt will hack the original pytorch graph during quantization, and the TensorQuantizer is used to compute the scaling factors

TOOL: AMMO/Modelopt

Workflow

PTQ (Post Training Quantization) – Export Quantized Model

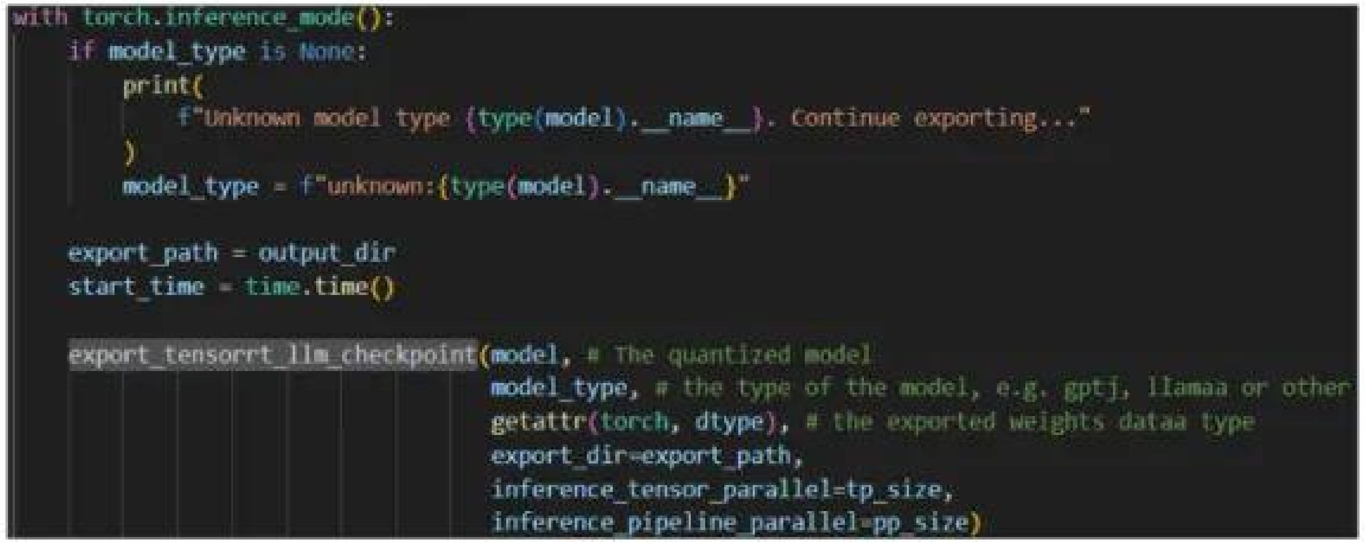
step 1 : gen scaling factor



The exported model config are stored as

•A single JSON file recording the model structure and metadata

•A group of SAFETENSORS weight files each recording the local calibrated model on a single GPU rank (model weights, scaling factors per GPU).



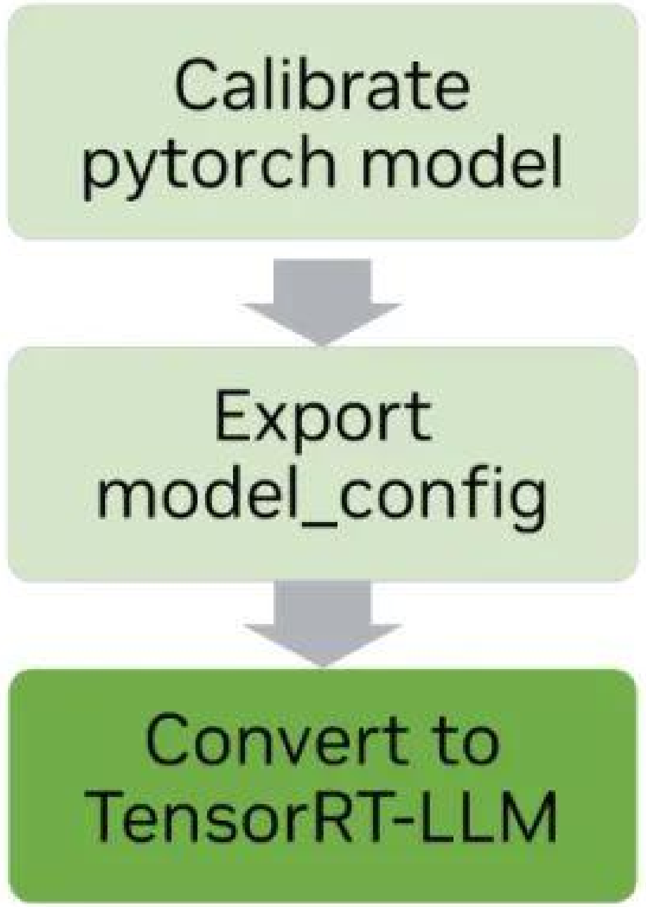
```python
with torch.inference_mode():
    if model_type is None:
        print(
            f"Unknown model type {type(model).__name__}. Continue exporting..."
        )
        model_type = f"unknown:{type(model).__name__}"

    export_path = output_dir
    start_time = time.time()

    export_tensorrt_llm_checkpoint(model, # The quantized model
                                   model_type, # the type of the model, e.g. gptj, llamaa or other
                                   getattr(torch, dtype), # the exported weights dataa type
                                   export_dir=export_path,
                                   inference_tensor_parallel=tp_size,
                                   inference_pipeline_parallel=pp_size)
```

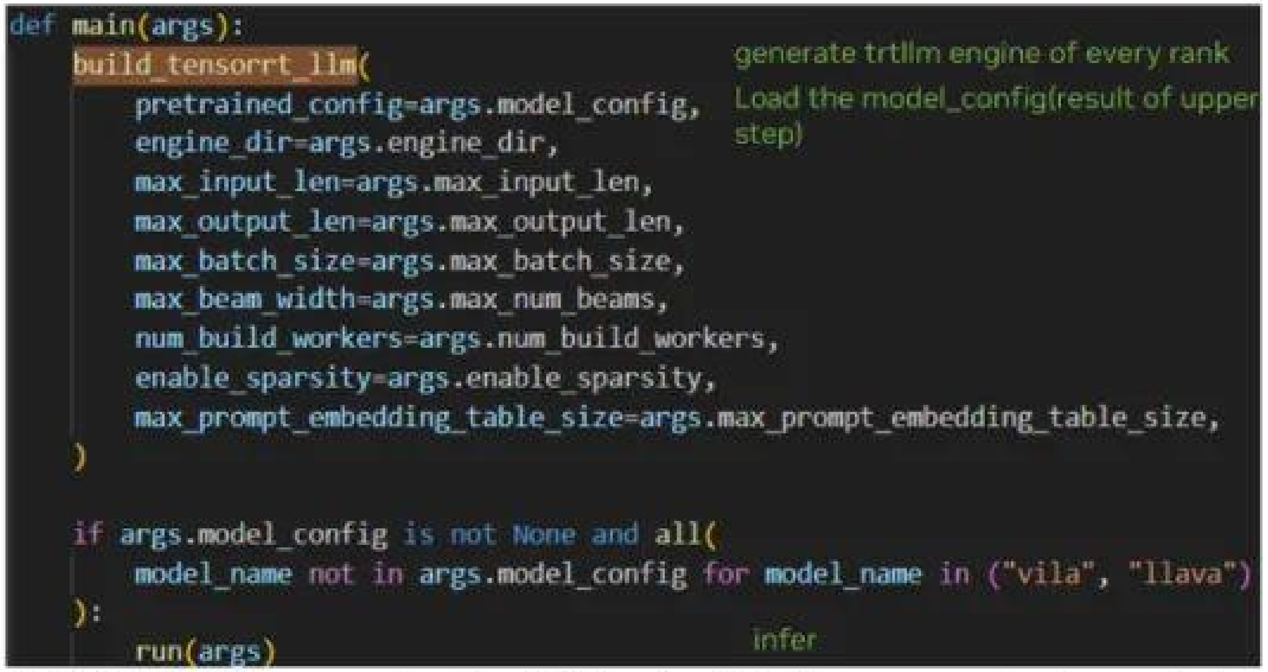
TOOL: AMMO/Modelopt

Workflow

PTQ (Post Training Quantization) – Covert to TensorRT-LLM

step 2 : gen the FP8 engine




generate trtllm engine of every rank
Load the model_config(result of upper step)
infer

```
def main(args):
    build_tensorrt_llm(
        pretrained_config=args.model_config,
        engine_dir=args.engine_dir,
        max_input_len=args.max_input_len,
        max_output_len=args.max_output_len,
        max_batch_size=args.max_batch_size,
        max_beam_width=args.max_num_beams,
        num_build_workers=args.num_build_workers,
        enable_sparsity=args.enable_sparsity,
        max_prompt_embedding_table_size=args.max_prompt_embedding_table_size,
    )

    if args.model_config is not None and all(
        model_name not in args.model_config for model_name in ("vila", "llava")
    ):
        run(args)
```

AMMO offers a single API to build the exported model from the quantization stage.

- **3.1 load model config**
    - **Model_configs keeps the graph and weights.**
- **3.2 convert to trtllm**
    - **Return TRTLLM engine**

tensorrt-llm can re-construct the quantized model with the model configs and run for inference.

How to debug

TensorRT-LLM Debug API

- Step 1: Register the tensor as output.
- Step 2: build engine.
- Step 3: Print and save the tensor.

1.In tensorrt_llm/models/gpt/model.py, we register the MLP output tensor

```
hidden_states = residual + attention_output.data

residual = hidden_states
hidden_states = self.post_layernorm(hidden_states)

hidden_states = self.mlp(hidden_states)
# register as model output
# ------------------------------------------------------
self.register_network_output('mlp_output', hidden_states)
# ------------------------------------------------------

hidden_states = residual + hidden_states
```

2. Build the TensorRT engine of the model

```
# Build TensorRT-LLM engines with --enable_debug_output
trtllm-build --checkpoint_dir gpt2/trt_ckpt/fp16/1-gpu \
        --gpt_attention_plugin float16 \
        --remove_input_padding enable \
        --enable_debug_output \
        --output_dir gpt2/trt_engines/fp16/1-gpu
```

3.In tensorrt_llm/runtime/generation.py, we print the debug info

```
if self.debug_mode:
    torch.cuda.synchronize()
    # ---------------------------------------------
    if step == 0:
        print(self.debug_buffer.keys())
    print(f"Step: {step}")
    print(self.debug_buffer['transformer.layers.6.mlp_output'])
    # ---------------------------------------------
```

4. Then, run ../run.py with --debug_mode and --use_py_session

```
python3 ../run.py --engine_dir gpt2/trt_engines/fp16/1-gpu \
        --tokenizer_dir gpt2 \
        --max_output_len 8 \
        --debug_mode \
        --use_py_session
```

How to debug

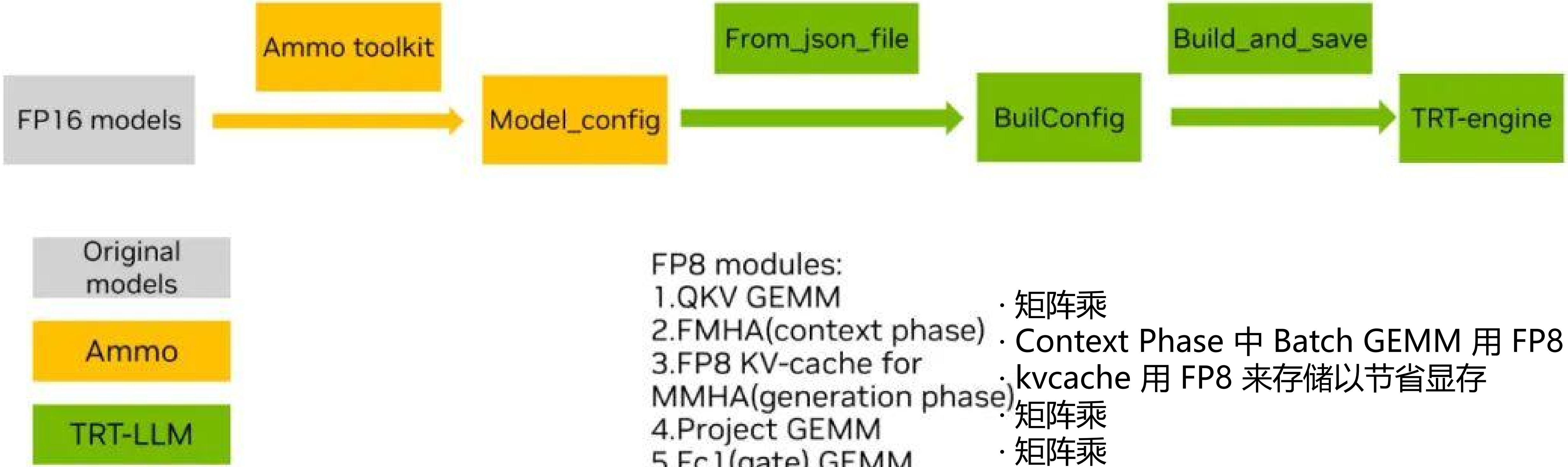
Possible Questions

- compare the intermediate output of fp16 and fp8
  - if the outputs of GEMM are different, check the channel order of the weights
  - if the attention's output are not consistency, check the parameters of the attention

Deep dive

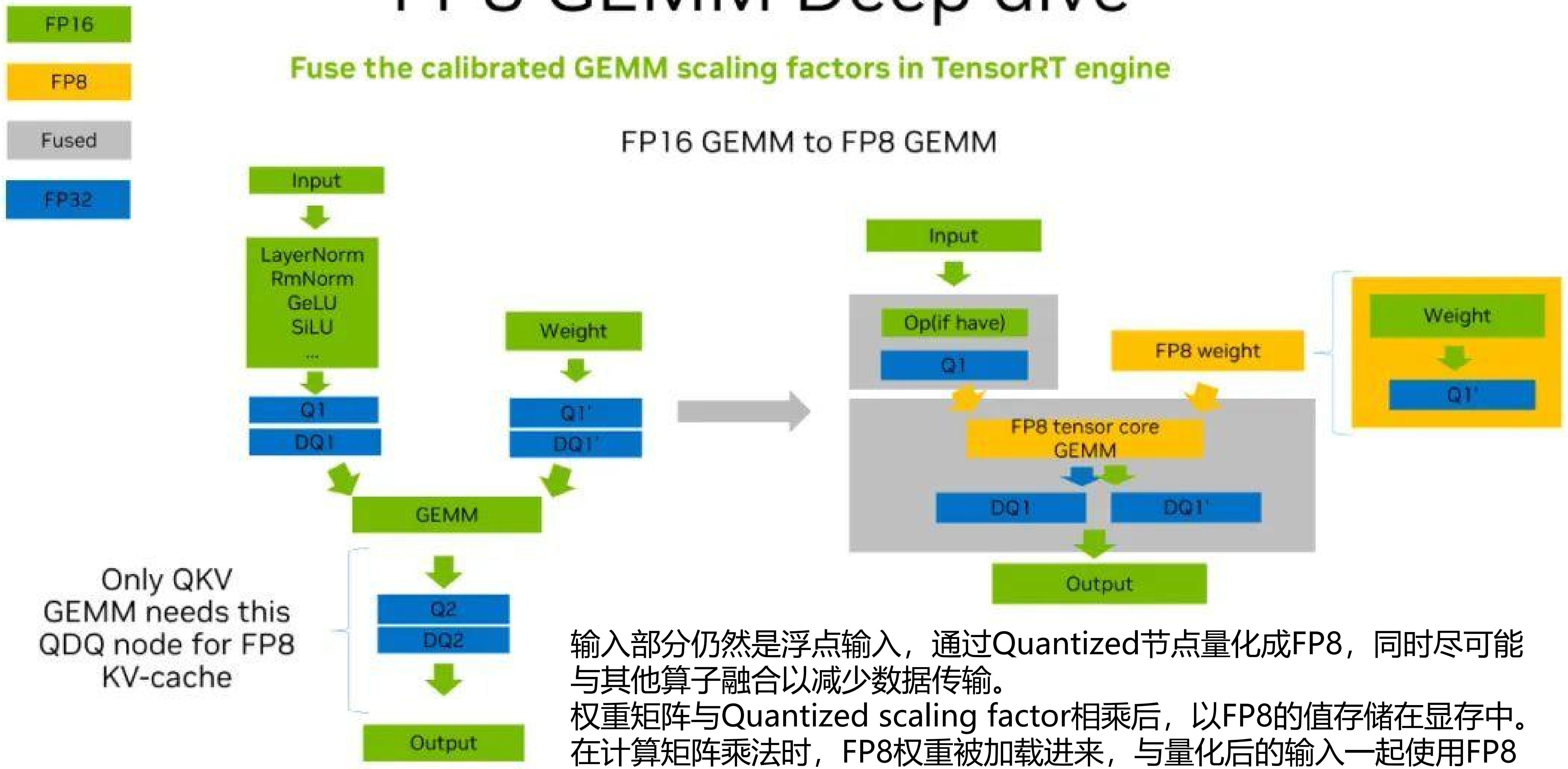
Read the model-config and build TRT-LLM engine

From FP16 models to TRT-engines



FP8 modules:
1.QKV GEMM · 矩阵乘
2.FMHA(context phase) · Context Phase 中 Batch GEMM 用 FP8
3.FP8 KV-cache for MMHA(generation phase) · kvcache 用 FP8 来存储以节省显存
4.Project GEMM · 矩阵乘
5.Fc1(gate) GEMM · 矩阵乘
6.Fc2 GEMM · 矩阵乘

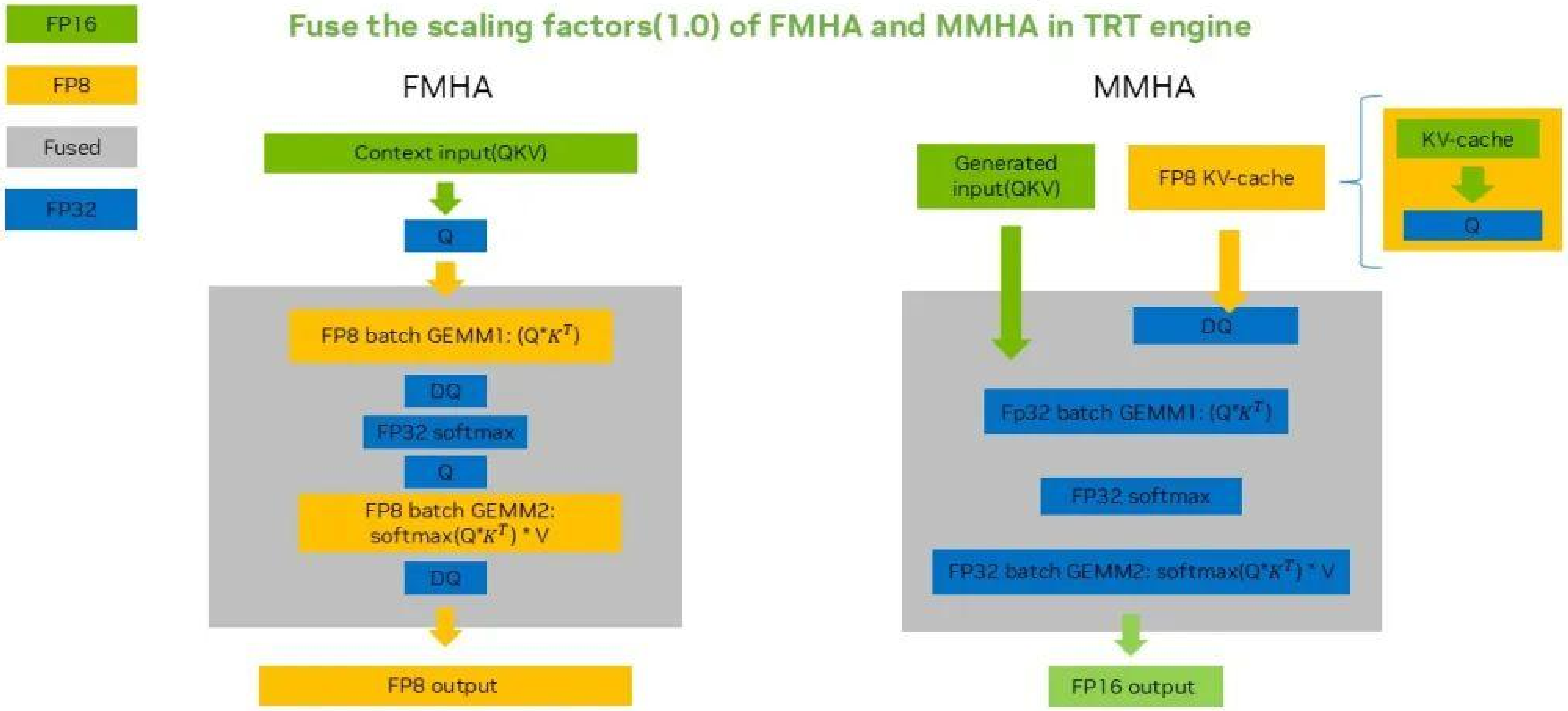
FP8 GEMM Deep dive



输入部分仍然是浮点输入，通过Quantized节点量化成FP8，同时尽可能与其他算子融合以减少数据传输。
权重矩阵与Quantized scaling factor相乘后，以FP8的值存储在显存中。
在计算矩阵乘法时，FP8权重被加载进来，与量化后的输入一起使用FP8 tensor core进行计算。

FP8 Attention deep dive

Fuse the scaling factors(1.0) of FMHA and MMHA in TRT engine



Compared with INT8, FP8 could be used to quantize the FMHA kernel because of the robust accuracy

Since the datatype convert instructions for fp8 are more efficiency, MMHA with FP8 KV-cache has better performance than INT8 KV-cache

除了矩阵乘，Attention 部分也可以借助 FP8 做运算。主要有两个：

（1）Fused Multi-Head Attention：做 Context phase 时，Attention 计算中的 batch GEMM 可以用 FP8 计算。因为 FMHA 是一个融合的 kernel，由两个 batch GEMM 和中间的 softmax 组成。由于 softmax 是累加的过程，所以必须用高精度 FP32 处理。但对于 batch GEMM，可以直接借助 FP8 的 Tensor Core 计算，最终输出是一个 FP8 的输出。这样输出的原因是 FMHA kernel 后，紧跟着一个 FP8 的矩阵乘 project GEMM，可以直接接收 FP8 的输出，所以直接输出一个 FP8 即可，减少了一次量化。

对于 FMHA，为什么不用 INT8？这里我们做过相应的实验，INT8 的 FMHA 在精度上比 FP8 有很大的下降。所以，INT8 由于精度问题用不了，而 FP8 的精度更鲁棒。同时，也因为 FP8 在绝对值相对较小的情况下，打点比 INT8 的数据分布更密集。但当绝对值很大时，对于离群点部分，INT8 不区分离群点和非离群点的打点密集程度，而 FP8 在离群点的地方打点很疏，在非离群点打点很密集，所以 FP8 的精度更鲁棒。

FP8 中的 Quantized 和 Dequantized，有一个 per tensor 量化参数就可以搞定。不需像 INT8 per token + per channel 这样复杂，FP8 就可以保持精度，这也是用 FP8 显而易见的好处。

（2）Masked Multi-Head Attention：Generation phase 计算 Attention 模块时，需要用融合的算子。因为 MMHA 的计算量比 FMHA 小很多，虽然也需要做 batched GEMM，batched GEMM 的 batch 维度都是 BS * HEAD_NUM，区别在于，context phase 的 GEMM 是 [length, head_size] * [head_size, length]，而 generation 的 GEMM 是 [1, head_size] * [head_size, length]，因此 batch GEMM 并不是计算密集型的计算过程，所以换 FP8 的收益不大，直接用浮点即可。但是加载 KV-cache 的模块可以通过 FP8 量化来节省显存。KV-cache 有 INT8 KV-cache，也有 FP8 KV-cache。相比 INT8，FP8 的精度更鲁棒，在 Hopper 硬件架构下，FP8 KV-cache 转出浮点的速度比 INT8 快。所以，FP8 KV-cache 的 MMHA 速度比 INT8 KV-cache 的 MMHA 要快。

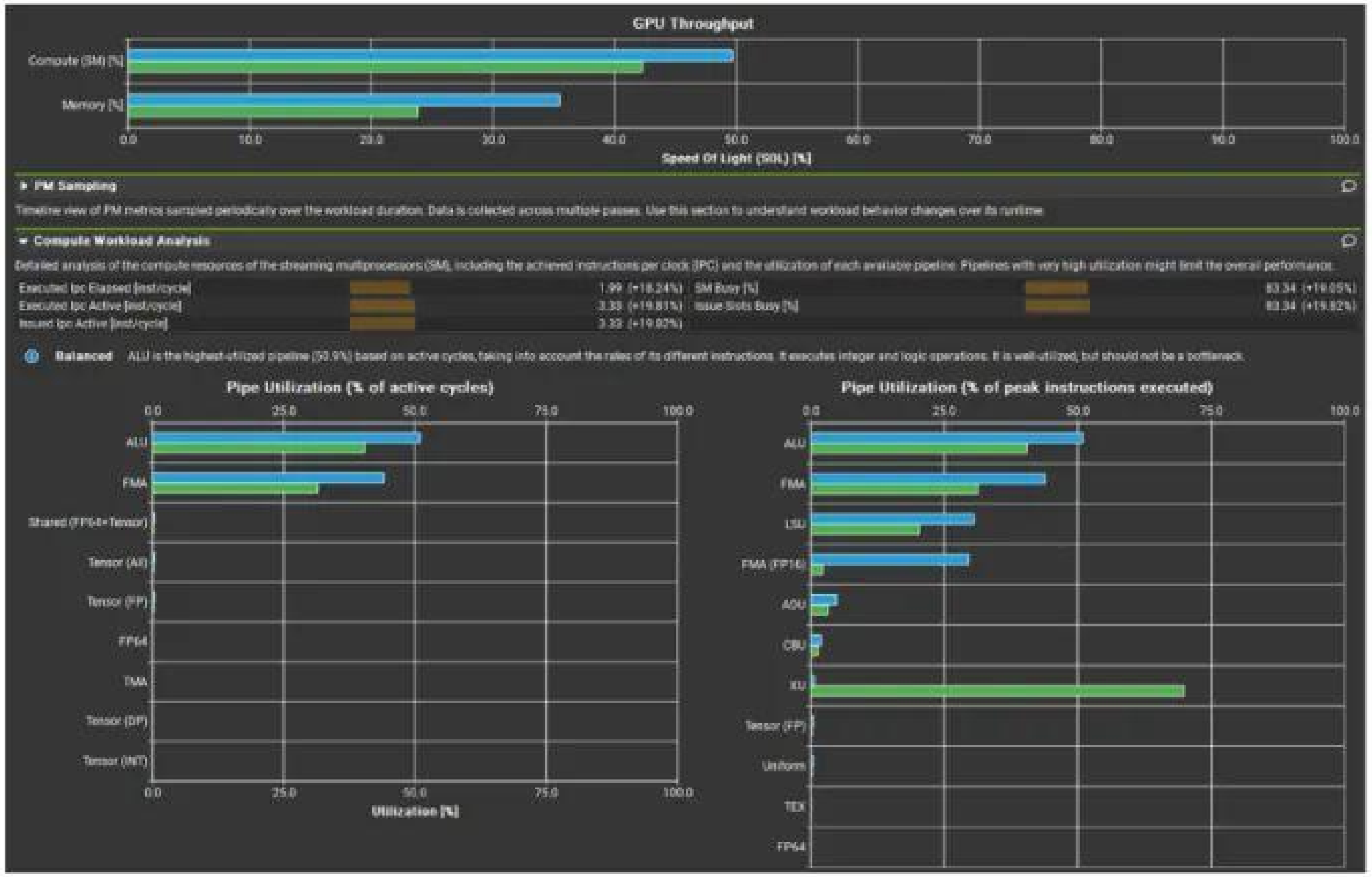
Deep dive

FP8 KV-cache vs INT8 KV-cache for MMHA

1. Compute and memory throughputs are lower than 50%, MMHA is a latency bound kernel.

2.XU Pipe (int-to-float) has very high utilization which leads to Stall MIO Throttle.

3. FP8 convert has better efficiency, (ALU and FMA have lower utilization compared with XU pipe.



▸ PM Sampling

Timeline view of PM metrics sampled periodically over the workload duration. Data is collected across multiple passes. Use this section to understand workload behavior changes over its runtime.

▾ Compute Workload Analysis

Detailed analysis of the compute resources of the streaming multiprocessors (SM), including the achieved instructions per clock (IPC) and the utilization of each available pipeline. Pipelines with very high utilization might limit the overall performance.

Executed Ipc Elapsed [inst/cycle]	1.99 (+18.24%)	SM Busy [%]	83.34 (+19.05%)
Executed Ipc Active [inst/cycle]	3.33 (+19.81%)	Issue Slots Busy [%]	83.34 (+19.82%)
Issued Ipc Active [inst/cycle]	3.33 (+19.92%)		

Balanced ALU is the highest-utilized pipeline (50.9%) based on active cycles, taking into account the rates of its different instructions. It executes integer and logic operations. It is well-utilized, but should not be a bottleneck.

NCU report for MMHA.(Blue: fp8 KV-cache MMHA, Green: int8 KV-cache MMHA)

NVIDIA TensorRT-LLM Now Accelerates Encoder-Decoder Models with In-Flight Batching

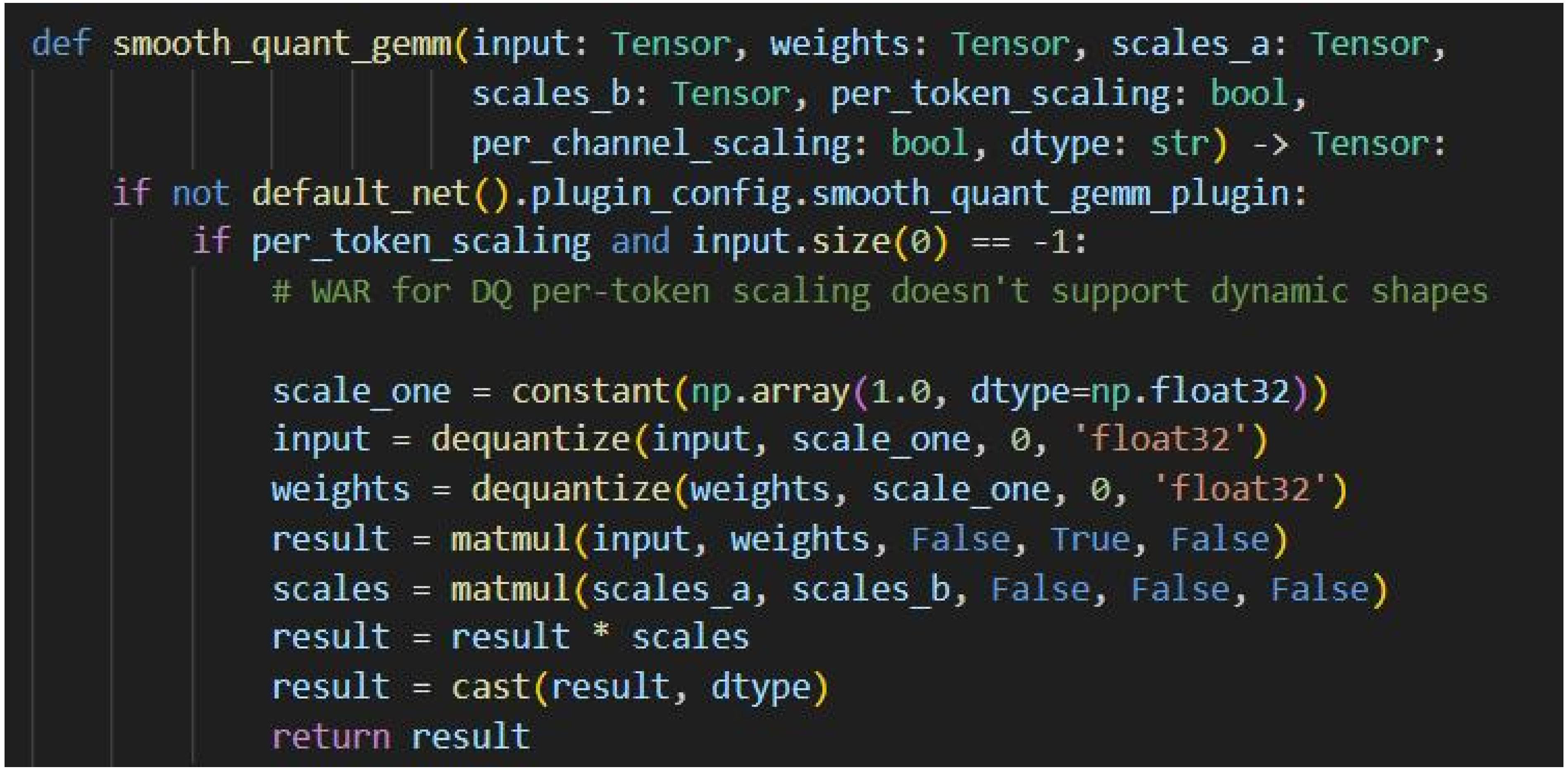
多GPU/多节点推理：支持通过全张量并行TP、管道并行PP和两者混合方式。

推理架构优化：使用“飞行批（In-flight Batching）处理”和KV缓存管理等提高吞吐量和降低延迟，特别适合编码器-解码器架构。

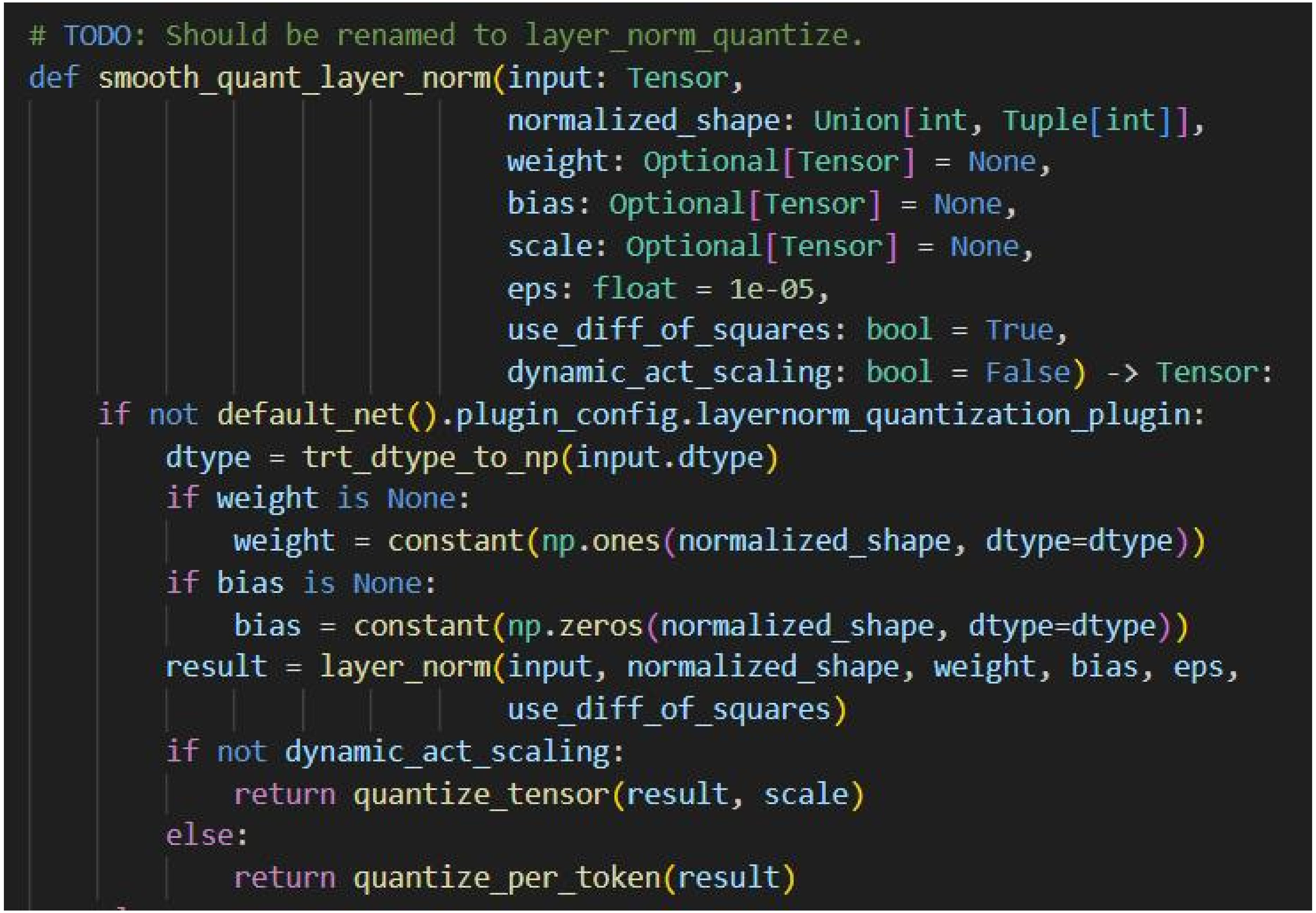
TensorRT 10.7.0 Release Notes

FP8推理：权重首先被转换为FP8，并融合操作以提高内存吞吐量。部分输出仍然需要FP16进行Reduction操作。

NVIDIA - TensorRT_LLM (E4M3)



```
def smooth_quant_gemm(input: Tensor, weights: Tensor, scales_a: Tensor,
                      scales_b: Tensor, per_token_scaling: bool,
                      per_channel_scaling: bool, dtype: str) -> Tensor:
    if not default_net().plugin_config.smooth_quant_gemm_plugin:
        if per_token_scaling and input.size(0) == -1:
            # WAR for DQ per-token scaling doesn't support dynamic shapes

            scale_one = constant(np.array(1.0, dtype=np.float32))
            input = dequantize(input, scale_one, 0, 'float32')
            weights = dequantize(weights, scale_one, 0, 'float32')
            result = matmul(input, weights, False, True, False)
            scales = matmul(scales_a, scales_b, False, False, False)
            result = result * scales
            result = cast(result, dtype)
            return result
```



```python
# TODO: Should be renamed to layer_norm_quantize.
def smooth_quant_layer_norm(input: Tensor,
                            normalized_shape: Union[int, Tuple[int]],
                            weight: Optional[Tensor] = None,
                            bias: Optional[Tensor] = None,
                            scale: Optional[Tensor] = None,
                            eps: float = 1e-05,
                            use_diff_of_squares: bool = True,
                            dynamic_act_scaling: bool = False) -> Tensor:
    if not default_net().plugin_config.layernorm_quantization_plugin:
        dtype = trt_dtype_to_np(input.dtype)
        if weight is None:
            weight = constant(np.ones(normalized_shape, dtype=dtype))
        if bias is None:
            bias = constant(np.zeros(normalized_shape, dtype=dtype))
        result = layer_norm(input, normalized_shape, weight, bias, eps,
                            use_diff_of_squares)
        if not dynamic_act_scaling:
            return quantize_tensor(result, scale)
        else:
            return quantize_per_token(result)
```

GPU架构下的FP8推理

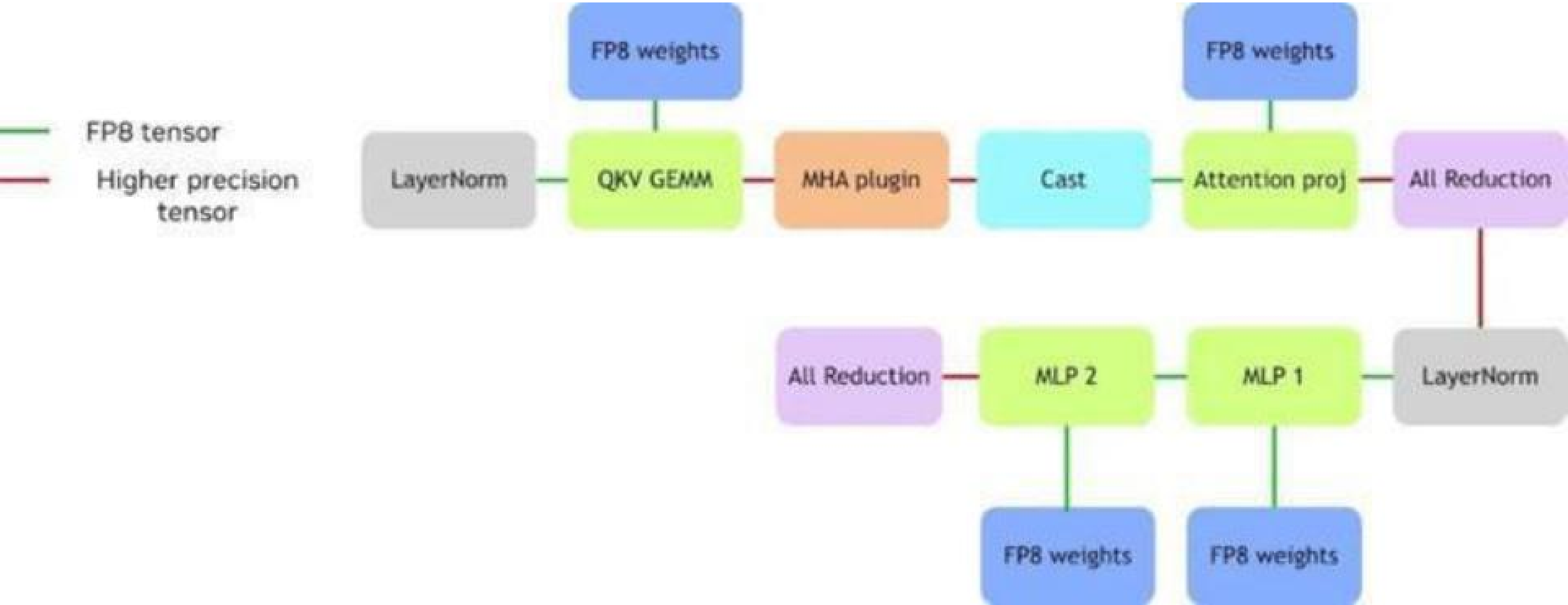
FP8推理通过NVIDIA TensorRT-LLM实现。权重输入先转换为FP8，并融合操作以提高内存吞吐，但部分输出仍需FP16进行Reduction。

NVIDIA Transformer Engine预置多种FP8 Kernel，可直接调用；

推理阶段应尽量进行操作融合，如将 LayerNorm 与后续数据格式转换操作整合，确保 kernel 输入输出尽可能维持 FP8，从而能够有效提升 GPU 内存吞吐。同样，GeLU (Gaussian Error Linear Unit) 激活函数也要力求融合。

目前少量输出仍会保持为 FP16，原因是 NVIDIA NCCL 仅支持高精度规约操作 （reduction），所以现在仍然需采用 FP16 进行 reduction，完成后再转化为 FP8。

经过上述融合后，推理流程就简化为图 12 所示。绿线代表 FP8 的输入输出（I/O），红线表示高精度 I/O。图中可见，最前端的 LayerNorm 输出与权重均为 FP8，矩阵输出暂时保持 FP16，与前文描述一致。并且经过测试验证可得，虽然矩阵输出精度对整体性能影响较小，但与输入问题的规模相关；且因其计算密集特性，对输出形态影响微弱。



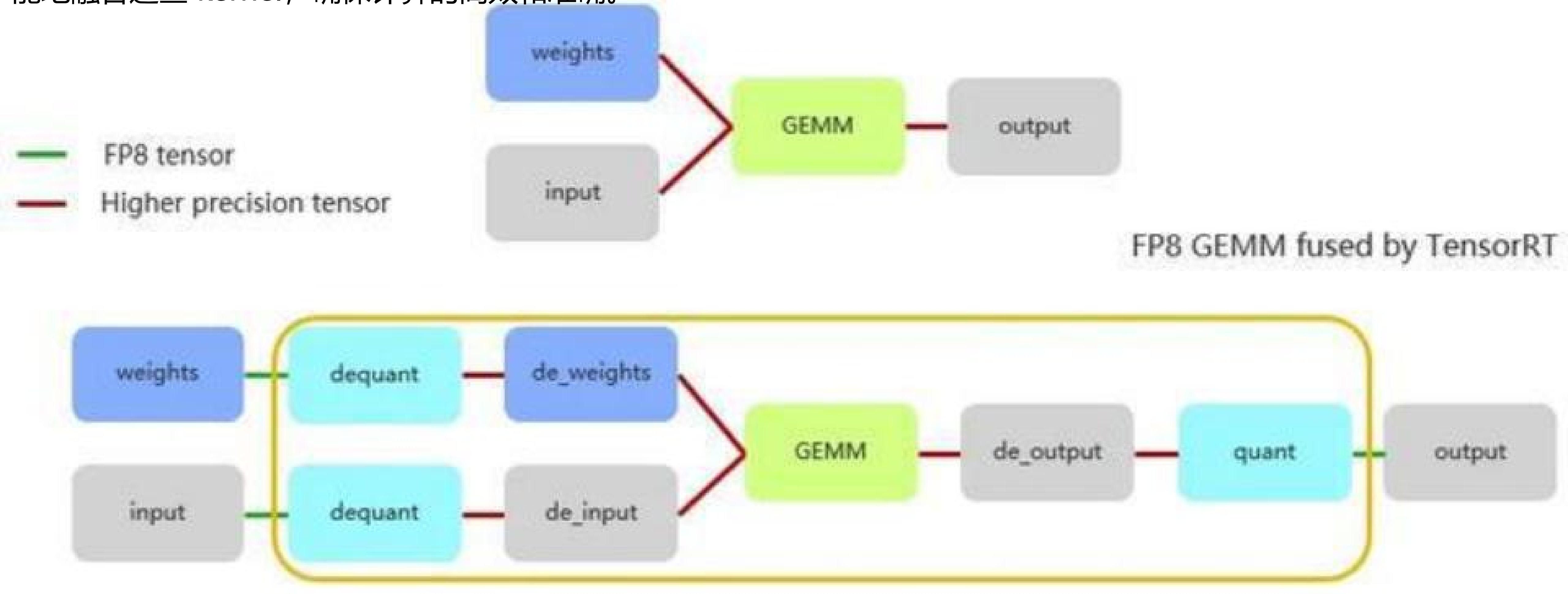
在完成 MHA（Multi-Head Attention）后，需要将结果转换为 FP8 以进行后续矩阵计算，Reduction 是以 FP16 执行后再转换到 FP8 的。对于 MLP1 和 MLP2，两者逻辑相似，但不同之处在于：MLP1 的输出可保持在 FP8，因为它已经把 GeLU 加 Bias 等操作直接融合到 MLP1 的 kernel。

由此引发的关键问题是，能否将剩余红线（高精度 I/O）全部转为绿线（FP8 I/O），实现进一步的加速优化？这正是 NVIDIA 持续进行的方向。以 reduction 为例，NVIDIA 正研究直接实现 FP8 reduction，尽管中间累加仍需高精度，但在数据传输阶段可采用 FP8。与现有 reduction 不同的是，FP8 reduction 内部需引入反量化（de-quantization）与量化 （quantization）操作，故需定制开发 reduction kernel。

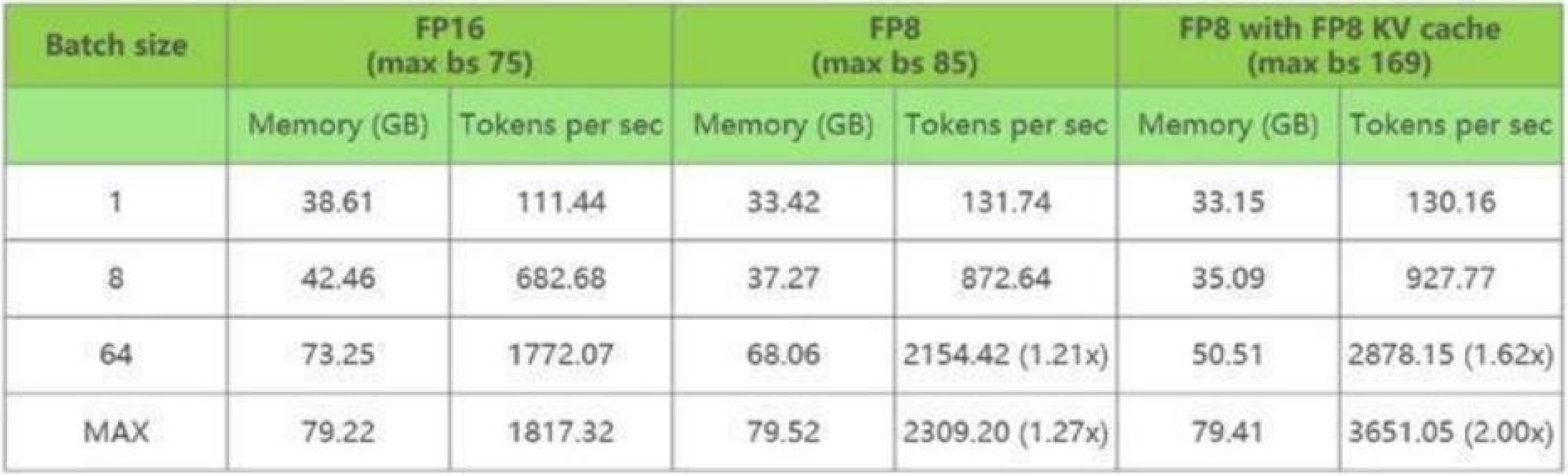
使用 TensorRT-LLM 实现 FP8 推理

- TensorRT-LLM 是基于 NVIDIA TensorRT 构建，其 FP8 能力也主要是通过 TensorRT 提供。自 TensorRT 9.0 版本起，官方就已经开始支持 FP8 推理。要在 TensorRT 中启用 FP8 推理，需完成以下几步：
- 设置 FP8 标志：通过调用 config.set_flag (trt.BuilderFlag.FP8) 在 TensorRT 配置中启用 FP8 支持。类似 INT8、BF16、FP16，FP8 也是类似的启用方式。
- 添加 GEMM 缩放因子（scale）：主要针对输入和权重，需在 weight.py （TensorRT-LLM 中的文件）中额外加载这些缩放因子。这是 FP8 推理中不可或缺的步骤。
- 编写 FP8模型：现阶段我们需要明确编写需要 FP8 支持的模型。具体做法如下：将原始 FP16 输入量化至 FP8，随后进行反量化；权重同样进行量化与反量化操作。如此编写的模型，TensorRT 会自动将量化与反量化操作尽可能与前一个 kernel 融合，以及将反量化操作与 matmul kernel 融合。最终生成的计算图表现为量化后的 X 与 W 直接进行 FP8 计算，输出也为 FP8 结果。
- 为了简化 FP8 在 TensorRT-LLM 中的应用，TensorRT-LLM 已对其进行封装，提供了 FP8 linear 函数和 FP8 row linear 函数来实现。对于使用直接线性层（linear layer），则无需重新编写代码，直接调用函数即可。

首先权重以 FP8 精度存储的，在进行计算前，权重先经历一次反量化。注意，在此之前，权重的量化已在输入前完成了，此处仅需进行反量化操作。这意味着，在进行矩阵内部计算时，实际上是使用反量化后的数据，通常是 FP16 或甚至 FP32 来进行运算的。
矩阵层尽管以 FP8 表示，但累加是采用 FP32 完成，累加后再乘以 scale 的相关参数，形成如图所示的计算流程。最终得到的结果具备较高精度。由于累加器（accumulator）需要采用高精度的数值，因此，要获得最终 FP8 的输出结果，模型还需经过一个量化节点 （quantitation node）。
回顾整个流程，输入经历了量化与反量化操作。其中，量化 kernel 发生在反量化 kernel 之前，而 TensorRT 则会智能地融合这些 kernel，确保计算的高效和准确。



FP8 GEMM fused by TensorRT



Batch size	FP16 (max bs 75)		FP8 (max bs 85)		FP8 with FP8 KV cache (max bs 169)	
	Memory (GB)	Tokens per sec	Memory (GB)	Tokens per sec	Memory (GB)	Tokens per sec
1	38.61	111.44	33.42	131.74	33.15	130.16
8	42.46	682.68	37.27	872.64	35.09	927.77
64	73.25	1772.07	68.06	2154.42 (1.21x)	50.51	2878.15 (1.62x)
MAX	79.22	1817.32	79.52	2309.20 (1.27x)	79.41	3651.05 (2.00x)

GPT-J 6b, input length 1024, output length 256, CUDA 12.2, Driver 535.104.05, TensorRT 9.1 on *H100