

TensorRT-LLM保姆级教程（三）-使用Triton推理服务框架部署模型

吃果冻不吐果冻皮 2024-08-01 👁 1,167 ⌚ 阅读19分钟

关注

本文为稀土掘金技术社区首发签约文章，30天内禁止转载，30天后未获授权禁止转载，侵权必究！

随着大模型的爆火，投入到生产环境的模型参数量规模也变得越来越大（从数十亿参数到千亿参数规模），从而导致大模型的推理成本急剧增加。因此，市面上也出现了很多的推理框架，用于降低模型推理延迟以及提升模型吞吐量。

本系列将针对 TensorRT-LLM 推理框架进行讲解。

- [TensorRT-LLM保姆级教程（一）-快速入门](#)
- [TensorRT-LLM保姆级教程（二）-开发实践](#)
- TensorRT-LLM保姆级教程（三）-使用Triton推理服务框架部署模型

之前讲述过模型推理服务化框架Triton。

- [模型推理服务化框架Triton保姆式教程（一）：快速入门](#)
- [模型推理服务化框架Triton保姆式教程（二）：架构解析](#)
- [模型推理服务化框架Triton保姆式教程（三）：开发实践](#)

本文将结合 TensorRT-LLM （作为后端推理引擎）与 Triton Inference Server 完成 LLM 服务部署工作。

另外，我撰写的大模型相关的博客及配套代码均整理放置在Github：[llm-action](#)，有需要的朋友自取。



TensorRT-LLM 是一个**用于定义大语言模型并构建 TensorRT 引擎**的 Python API，以高效地在 NVIDIA GPU 上执行推理。TensorRT-LLM 包含用于创建 Python 和 C++ 运行时以及执行这些 TensorRT 引擎的组件。它还包括一个用于与 [NVIDIA Triton 推理服务](#)集成的后端 (tensorrtllm_backend) ；使用 TensorRT-LLM 构建的模型可以在单个 GPU或在具有多个 GPU 的多个节点上执行（（使用张量并行或流水线并行））。

下面将结合 TensorRT-LLM 与 Triton Inference Server 完成 LLM 部署。

环境准备

上一篇文章中讲述了如何基于源码进行安装。本文为了方便起见，直接拉取[官方](#)已经构建好的镜像运行即可。

- 镜像：nvcr.io/nvidia/tritonserver:24.06-trtllm-python-py3，基于 TensorRT-LLM 的 v0.10.0 版本。
- 模型：Qwen1.5
- 服务器：8x Nvidia H20 (96GB)
- 代码：
 - TensorRT-LLM： [github.com/NVIDIA/Tens...](#)
 - TensorRT-LLM Backend： [github.com/triton-infe...](#)

创建并进入容器。

▼ bash

复制代码

```
1 docker run -dt --name triton-server-6 \  
2 --restart=always \  
3 --gpus '"device=6"' \  
4 --network=host \  
5 --shm-size=32g \  
6 -v /data/hpc/home/guodong.li/workspace:/workspace \  
7 -w /workspace \  
8 aiharbor.msxf.local/nvidia/tritonserver:24.06-trtllm-python-py3 \  
9 /bin/bash  
10  
11 docker exec -it triton-server-6 bash
```

模型技术社区及资源

接下来，进行模型权重格式转换，并将其编译成 TensorRT 引擎。

单卡推理

第一步，将 HF 模型权重格式转换为 TensorrtLLM 模型权重格式。

▼

bash

复制代码

```
1 cd /workspace/TensorRT-LLM/examples/qwen
2 python convert_checkpoint.py --model_dir /workspace/models/Qwen1.5-14B-Chat \
3                               --output_dir /workspace/models/Qwen1.5-14B-Chat-1tp-bf16-trt \
4                               --dtype bfloat16
```

常用参数说明：

- --model_dir: HF模型权重路径
- --output_dir: TensorrtLLM 模型权重输出路径
- --dtype: TensorrtLLM 模型权重数据类型
- --pp_size: 流水线并行大小
- --tp_size: 张量并行大小

运行完成后将生成新的配置文件和权重。

▼

diff

复制代码

```
1 -rw-r--r-- 1 root root      1126 Jul 18 10:49 config.json
2 -rw-r--r-- 1 root root 29676808144 Jul 18 10:49 rank0.safetensors
```

配置文件包含了模型的基础信息以及张量并行、流水线并行推理等配置，具体如下所示。

▼

json

复制代码

```
1 {
2     "architecture": "QWenForCausalLM",
3     "dtype": "bfloat16",
4     "logits_dtype": "float32",
5     "vocab_size": 152064,
6     "max_position_embeddings": 32768,
7     "hidden_size": 5120,
8     "num_hidden_layers": 40,
```

```
11     "head_size": 128,
12     "qk_layernorm": false,
13     "hidden_act": "silu",
14     "intermediate_size": 13696,
15     "norm_epsilon": 1e-06,
16     "position_embedding_type": "rope_gpt_neox",
17     "use_parallel_embedding": false,
18     "embedding_sharding_dim": 0,
19     "share_embedding_table": false,
20     "mapping": {
21         "world_size": 1,
22         "tp_size": 1,
23         "pp_size": 1,
24         "gpus_per_node": 8
25     },
26     "quantization": {
27         "quant_algo": null,
28         "kv_cache_quant_algo": null,
29         "group_size": 128,
30         "smoothquant_val": null,
31         "has_zero_point": false,
32         "pre_quant_scale": false,
33         "exclude_modules": [
34             "lm_head"
35         ]
36     },
37     "kv_dtype": "bfloat16",
38     "rotary_scaling": null,
39     "rotary_base": 1000000.0,
40     "qwen_type": "qwen2",
41     "disable_weight_only_quant_plugin": false
42 }
```

第二步，将模型编译为 TensorRT 引擎。

使用 TensorRT-LLM API 创建模型定义，将用 NVIDIA TensorRT 原语（构成神经网络的层）构建了一个运算图。这些运算映射到特定的内核（为 GPU 预先编写的程序）。

▼ CSS



复制代码

```
1 trtllm-build --checkpoint_dir /workspace/models/Qwen1.5-7B-Chat-1tp-bf16-trt \
2 --output_dir /workspace/models/Qwen1.5-14B-Chat-1tp-bf16-trt-engine-2 \
3 --gpt_attention_plugin bfloat16 \
4 --gemm_plugin bfloat16 \
5 --max_num_tokens 10000 \
```

```
8  --workers 1 \  
9  --use_paged_context_fmha enable \  
10 --multiple_profiles enable \  
11 --max_input_len 1024 \  
12 --max_output_len 1024 \  
13 --max_batch_size 256
```

常见参数说明：

- `--gpt_attention_plugin`：默认启用 GPT 注意力插件，使用高效的Kernel并支持 KV 缓存的 in-place 更新。它会减少内存消耗，并删除不需要的内存复制操作（与使用 `concat` 运算符更新 KV 缓存的实现相比）。
- `--context_fmha`：默认启用融合多头注意力，将触发使用单个Kernel执行 MHA/MQA/GQA 块的Kernel。
- `--gemm_plugin`：GEMM 插件利用 NVIDIA cuBLASLt 执行 GEMM 运算。在 FP16 和 BF16 上，建议启用它，以获得更好的性能和更小的 GPU 内存使用量。在 FP8 上，建议禁用。如果通过 `--gemm_plugin fp8` 启用。尽管可以正确推断具有较大批量大小的输入，但性能可能会随着批量大小的增加而下降。因此，目前该功能仅推荐用于在小批量场景下的降低延迟。
- `--use_custom_all_reduce`：启用自定义 AllReduce 插件。在基于 NVLink 的节点上，建议启用，在基于 PCIE 的节点上，不建议启用。自定义 AllReduce 插件为 AllReduce 运算激活延迟优化算法，而不是原生的 NCCL 算子。然而，在基于 PCIE 的系统上可能看不到性能优势。当限制为单个设备，自定义AllReduce将被禁用。因为其Kernel依赖于对对等设备的 P2P访问，当只有一个设备可见时这是不允许的。
- `--reduce_fusion enable`：当自定义 AllReduce 已启用时，此功能旨在将 AllReduce 之后的 ResidualAdd 和 LayerNorm Kernel 融合到单个Kernel中，从而提高端到端性能。注意：目前仅 llama 模型支持此功能。
- `--paged_kv_cache`：默认启用分页KV缓存。分页 KV 缓存有助于更有效地管理 KV 缓存的内存。它通常能使批量大小增加和效率提高。
- `--workers`：并行构建的worker数。
- `--use_paged_context_fmha`：启用分页上下文注意力。
- `--multiple_profiles`：在内置引擎中启用多个 TensorRT 优化配置文件，这将有利于性能，尤其是在禁用 GEMM 插件时，因为更多优化配置文件有助于 TensorRT 有更多机会选择更好的 Kernel。然而，它会增加引擎的构建时间。

构建完成之后生成了引擎文件以及配置文件。

```
1 -rw-r--r-- 1 root root          5645 Jul 18 21:36 config.json
2 -rw-r--r-- 1 root root 15485236972 Jul 18 21:36 rank0.engine
```

其中，引擎文件除了模型配置以外，还有很多引擎相关的配置。

▼ json



复制代码

```
1 {
2   "version": "0.10.0",
3   "pretrained_config": {
4     "architecture": "QWenForCausalLM",
5     "dtype": "bfloat16",
6     "logits_dtype": "float32",
7     "vocab_size": 151936,
8     "max_position_embeddings": 32768,
9     "hidden_size": 4096,
10    "num_hidden_layers": 32,
11    "num_attention_heads": 32,
12    "num_key_value_heads": 32,
13    "head_size": 128,
14    "qk_layernorm": false,
15    "hidden_act": "silu",
16    "intermediate_size": 11008,
17    "norm_epsilon": 1e-06,
18    "position_embedding_type": "rope_gpt_neox",
19    "use_parallel_embedding": false,
20    "embedding_sharding_dim": 0,
21    "share_embedding_table": false,
22    "mapping": {
23      "world_size": 1,
24      "tp_size": 1,
25      "pp_size": 1,
26      "gpus_per_node": 8
27    },
28    "quantization": {
29      "quant_algo": null,
30      "kv_cache_quant_algo": null,
31      "group_size": 128,
32      "smoothquant_val": null,
33      "has_zero_point": false,
34      "pre_quant_scale": false,
35      "exclude_modules": [
36        "lm_head"
37      ]
38    },
39  }
```

```
41     "rotary_base": 1000000.0,
42     "qwen_type": "qwen2",
43     "disable_weight_only_quant_plugin": false
44 },
45 "build_config": {
46     "max_input_len": 1024,
47     "max_output_len": 1024,
48     "opt_batch_size": null,
49     "max_batch_size": 256,
50     "max_beam_width": 1,
51     "max_num_tokens": 10000,
52     "opt_num_tokens": 256,
53     "max_prompt_embedding_table_size": 0,
54     "gather_context_logits": false,
55     "gather_generation_logits": false,
56     "strongly_typed": false,
57     "builder_opt": null,
58     "profiling_verbosity": "layer_names_only",
59     "enable_debug_output": false,
60     "max_draft_len": 0,
61     "speculative_decoding_mode": 1,
62     "use_refit": false,
63     "input_timing_cache": null,
64     "output_timing_cache": "model.cache",
65     "lora_config": {
66         "lora_dir": [],
67         "lora_ckpt_source": "hf",
68         "max_lora_rank": 64,
69         "lora_target_modules": [],
70         "trtllm_modules_to_hf_modules": {}
71     },
72     "auto_parallel_config": {
73         "world_size": 1,
74         "gpus_per_node": 8,
75         "cluster_key": "NVIDIA-H20",
76         "cluster_info": {
77             "inter_node_bw_per_device": 25,
78             "intra_node_bw_per_device": 450,
79             "inter_node_latency": 10,
80             "intra_node_latency": 10,
81             "intra_node_sharp": true,
82             "inter_node_sharp": true,
83             "memory_bw": 4022,
84             "memory_budget_per_device": 95,
85             "math_throughput": {
86                 "int4": 0,
87                 "int8": 2530,
```

```
90         "bfloat16": 1265,
91         "float32": 632
92     },
93     "memory_efficiency": 1.0,
94     "math_efficiency": 1.0,
95     "communication_efficiency": 1.0
96 },
97 "sharding_cost_model": "alpha_beta",
98 "comm_cost_model": "alpha_beta",
99 "enable_pipeline_parallelism": false,
100 "enable_shard_unbalanced_shape": false,
101 "enable_shard_dynamic_shape": false,
102 "enable_reduce_scatter": true,
103 "builder_flags": null,
104 "debug_mode": false,
105 "infer_shape": true,
106 "validation_mode": false,
107 "same_buffer_io": {
108     "past_key_value_(\\d+)": "present_key_value_\\1"
109 },
110 "same_spec_io": {},
111 "sharded_io_allowlist": [
112     "past_key_value_\\d+",
113     "present_key_value_\\d*"
114 ],
115 "fast_reduce": true,
116 "fill_weights": false,
117 "parallel_config_cache": null,
118 "profile_cache": null,
119 "dump_path": null,
120 "debug_outputs": []
121 },
122 "weight_sparsity": false,
123 "weight_streaming": false,
124 "use_strip_plan": false,
125 "max_encoder_input_len": 1024,
126 "use_fused_mlp": false,
127 "plugin_config": {
128     "bert_attention_plugin": "float16",
129     "gpt_attention_plugin": "bfloat16",
130     "gemm_plugin": "bfloat16",
131     "smooth_quant_gemm_plugin": null,
132     "identity_plugin": null,
133     "layernorm_quantization_plugin": null,
134     "rmsnorm_quantization_plugin": null,
135     "nccl_plugin": null,
136     "lookup_plugin": null,
```



```
139         "weight_only_quant_matmul_plugin": null,
140         "quantize_per_token_plugin": false,
141         "quantize_tensor_plugin": false,
142         "moe_plugin": "float16",
143         "mamba_conv1d_plugin": "float16",
144         "context_fmha": true,
145         "context_fmha_fp32_acc": false,
146         "paged_kv_cache": true,
147         "remove_input_padding": true,
148         "use_custom_all_reduce": true,
149         "multi_block_mode": false,
150         "enable_xqa": true,
151         "attention_qk_half_accumulation": false,
152         "tokens_per_block": 64,
153         "use_paged_context_fmha": true,
154         "use_fp8_context_fmha": false,
155         "use_context_fmha_for_generation": false,
156         "multiple_profiles": true,
157         "paged_state": true,
158         "streamingllm": false
159     }
160 }
161 }
```

多卡张量并行推理

如果需要多卡并行推理设置 `--pp_size` 和 `--tp_size` 即可，具体视情况而定。参考示例如下所示。

▼ CSS

复制代码

```
1 python convert_checkpoint.py --model_dir /workspace/models/Qwen1.5-14B-Chat \
2                               --output_dir /workspace/models/Qwen1.5-14B-Chat-2tp-bf16-trt \
3                               --dtype bfloat16 \
4                               --tp_size 2
```

模型权重格式转换之后进行编译，与上面单卡推理一致。

▼ CSS

复制代码

```
1 trtllm-build --checkpoint_dir /workspace/models/Qwen1.5-14B-Chat-2tp-bf16-trt \
2 --output_dir /workspace/models/Qwen1.5-14B-Chat-2tp-bf16-trt-engine-2 \
3 --gpt_attention_plugin bfloat16 \
```

```
6 --use_custom_all_reduce enable \  
7 --paged_kv_cache enable \  
8 --workers 2 \  
9 --use_paged_context_fmha enable \  
10 --multiple_profiles enable \  
11 --max_input_len 1024 \  
12 --max_output_len 1024 \  
13 --max_batch_size 256
```

FP8量化

接下来，如果希望降低模型的权重，同时提升推理的性能，可以采用FP8量化。由于使用FP8离线静态量化，因此需要校准数据集。我这里由于网络问题无法自动下载数据集，因此，加载本地数据集，对做 `/usr/local/lib/python3.10/dist-packages/tensorrt_llm/quantization/quantize_by_modelopt.py` 代码了少量改动。

▼ python



复制代码

```
1 def get_calib_dataloader(dataset_name_or_dir="cnn_dailymail",  
2     ....  
3     elif "cnn_dailymail" in dataset_name_or_dir:  
4         dataset = load_dataset(dataset_name_or_dir, name="3.0.0", split="train")  
5         dataset = dataset["article"][:calib_size]  
6     elif os.path.isdir(dataset_name_or_dir):  
7         print(  
8             f"Recognized local dataset repo {dataset_name_or_dir} for calibration; "  
9             "assuming the calibration data are in the train split and text column."  
10        )  
11        dataset = load_dataset("parquet", split="train", data_files=dataset_name_or_dir+"/alpaca  
12        #dataset = load_dataset(dataset_name_or_dir, split="train")  
13        dataset = dataset["text"][:calib_size]  
14    else:  
15        raise NotImplementedError  
16
```

然后，进行模型权重格式转换。

▼ CSS



复制代码

```
1 cd /workspace/TensorRT-LLM/examples/quantization/  
2 python quantize.py --model_dir /workspace/models/Qwen1.5-14B-Chat \
```

```
5 --kv_cache_dtype fp8 \  
6 --output_dir /workspace/models/Qwen1.5-14B-Chat-1tp-fp8-kvfp8 \  
7 --calib_size 512 \  
8 --calib_dataset /workspace/datas
```

常用参数说明：

- `--dtype`：设置模型的数据类型
- `--qformat`：指定量化的数据格式
- `--kv_cache_dtype`：指定 KV Cache 的数据类型
- `--pp_size`：流水线并行大小
- `--tp_size`：张量并行大小

量化并转换完权重格式之后，接下来进行编译即可。对于 FP8 有一些特殊的配置用于提升性能。

▼ CSS



复制代码

```
1 # Build trtllm engines from the trtllm checkpoint# Enable fp8 context fmha to get further accele  
2  
3 trtllm-build --checkpoint_dir /workspace/models/Qwen1.5-14B-Chat-1tp-fp8-kvfp8-trt \  
4 --output_dir /workspace/models/Qwen1.5-14B-Chat-1tp-fp8-kvfp8-trt-engine-2 \  
5 --gemm_plugin disable \  
6 --max_num_tokens 10000 \  
7 --use_custom_all_reduce enable \  
8 --paged_kv_cache enable \  
9 --workers 1 \  
10 --use_paged_context_fmha enable \  
11 --multiple_profiles enable \  
12 --max_input_len 1024 \  
13 --max_output_len 1024 \  
14 --max_batch_size 256 \  
15 --use_fp8_context_fmha enable
```

常用的参数说明：

- `--use_fp8_context_fmha`：开启FP8上下文融合多头注意力，建议在使用fp8量化时开启，以提高性能。
- `--gemm_plugin`：GEMM 插件利用 NVIDIA cuBLASLt 执行 GEMM 运算。在 FP16 和 BF16 上，建议启用它，以获得更好的性能和更小的 GPU 内存使用量。在 FP8 上，建议禁

性能可能会随着批量大小的增加而下降。因此，目前该功能仅推荐用于在小批量场景下的降低延迟。


编译完成之后就可以进行模型推理了，但是通常情况下我们需要部署成模型服务以暴露API接口给客户使用。接下来使用 Triton 进行服务化部署。

使用 Triton 进行服务化部署

要创建生产环境的 LLM 服务，需使用 TensorRT-LLM 的 [Triton 推理服务后端](#)(`tensorrtllm_backend`)，以利用 TensorRT-LLM C++ 运行时进行快速推理，并包括一些优化，例如：in-flight batching 和分页 KV 缓存。

前面已经讲过要使用具有 TensorRT-LLM 后端的 Triton 推理服务，可通过 NVIDIA NGC 预构建[容器](#)即可。

首先，进入`tensorrtllm_backend`项目，然后拷贝使用Triton推理服务进行模型部署的模型仓库（即配置文件模板及前后置处理代码等）。

▼ bash  复制代码

```
1 cp -r /workspace/tensorrtllm_backend/all_models/inflight_batcher_llm /workspace/models/qwen2-14b
```

◀

▶

`inflight_batcher_llm` 模型库里面包含五个模型：

- 预处理（preprocessing）：该模型用于tokenizing，即从提示（字符串）到 `input_ids`（整数列表）的转换。
- `tensorrt_llm`：该模型是 TensorRT-LLM 模型的包装器，用于推理。输入规范可以在[这里](#)找到
- 后处理（postprocessing）：该模型用于de-tokenizing，即从`output_ids`（整数列表）到输出（字符串）的转换。
- `ensemble`：该模型可用于将预处理、`tensorrt_llm` 和后处理模型连接在一起。
- `tensorrt_llm_bls`：该模型也可用于将预处理、`tensorrt_llm` 和后处理模型连接在一起。

当使用BLS（Business Logic Scripting）模型而不是ensemble模型时，应该将模型实例的数量设置为TRT引擎支持的最大批量大小，以允许更大并发请求执行。通过修改 BLS 模

BLS 模型有一个可选参数 `accumulate_tokens`，可在流式响应模式下使用该参数来调用具有所有累积Token（而不是仅一个Token）的后处理模型。这对于某些tokenizers可能是必要的。

然后，通过以下脚本批量修改配置文件。对于一些配置文件需要仔细设置，否则，可能会影响模型服务吞吐量等。

▼ bash



复制代码

```
1 # 指定Tokenizer文件路径
2 export HF_LLAMA_MODEL=/workspace/models/Qwen1.5-14B-Chat
3 # 指定模型TensorRT引擎路径
4 export ENGINE_PATH=/workspace/models/Qwen1.5-14B-Chat-1tp-bf16-trt-engine-2
5 # 指定模型部署配置文件路径
6 export TEMPLATE_PATH=/workspace/models/qwen2-14b-trt-engine-h20-1tp
7
8 cd /workspace/tensorrtllm_backend
9
10 # 修改配置文件
11 python3 tools/fill_template.py -i ${TEMPLATE_PATH}/preprocessing/config.pbtxt \
12 tokenizer_dir:${HF_LLAMA_MODEL},triton_max_batch_size:256,preprocessing_instance_count:1
13
14 python3 tools/fill_template.py -i ${TEMPLATE_PATH}/postprocessing/config.pbtxt \
15 tokenizer_dir:${HF_LLAMA_MODEL},triton_max_batch_size:256,postprocessing_instance_count:1
16
17 python3 tools/fill_template.py -i ${TEMPLATE_PATH}/tensorrt_llm_bls/config.pbtxt \
18 triton_max_batch_size:256,decoupled_mode:False,bls_instance_count:1,accumulate_tokens:False
19
20 python3 tools/fill_template.py -i ${TEMPLATE_PATH}/ensemble/config.pbtxt \
21 triton_max_batch_size:256
22
23 python3 tools/fill_template.py -i ${TEMPLATE_PATH}/tensorrt_llm/config.pbtxt \
24 triton_backend:tensorrtllm,triton_max_batch_size:256,decoupled_mode:True,\
25 max_beam_width:1,engine_dir:${ENGINE_PATH},\
26 kv_cache_free_gpu_mem_fraction:0.95,\
27 exclude_input_in_output:True,\
28 batching_strategy:inflight_fused_batching,max_queue_delay_microseconds:0,\
29 enable_chunked_context:True,batch_scheduler_policy:max_utilization
30
31 # 查看一些关键配置文件是否修改成功。
32 echo ${TEMPLATE_PATH}
33 cat ${TEMPLATE_PATH}/tensorrt_llm/config.pbtxt | grep -C 5 "kv_cache_free_gpu_mem_fraction"
34 cat ${TEMPLATE_PATH}/tensorrt_llm/config.pbtxt | grep -C 5 "inflight_fused_batching"
35 cat ${TEMPLATE_PATH}/tensorrt_llm/config.pbtxt | grep -C 5 "batch_scheduler_policy"
36 cat ${TEMPLATE_PATH}/tensorrt_llm/config.pbtxt | grep -C 5 "enable_chunked_context"
```



- `engine_dir`: 指定模型TensorRT引擎路径。
- `kv_cache_free_gpu_mem_fraction`: 指定KV 缓存可用内存量。取值为 `0.0` 和 `1.0` 之间的浮点数。默认值为 `0.90` , 意味着90%的空闲GPU内存将用于在KV缓存中保存Token。根据该值, TensorRT-LLM 可以确定 KV 缓存管理器中Token的最大数量。**如果同一 GPU 上没有执行其他程序, 建议使用 `0.95` 或者更高的值进行测试, 以实现高吞吐量。**但该参数不能设置为 `1.0` , 因为必须为输入和输出保留一定量的内存。
- `batching_strategy`: 指定模型服务的Batch策略, 可以选择 `V1` 、 `inflight_batching` 和 `inflight_fused_batching` , 建议使用 `inflight_fused_batching` 来提高吞吐量并减少延迟。
- `batch_scheduler_policy`: 指定批处理调度策略, 目前有两种批处理调度策略: `MAX_UTILIZATION` 和 `GUARANTEED_NO_EVICT` 。当启动 in-flight sequence batching时, 建议将调度策略设置为 `MAX_UTILIZATION` , 以便在前向循环的每次迭代中打包尽可能多的请求。如果在内存分配方面, 对于 KV 缓存限制更保守方法是设置为 `GUARANTEED_NO_EVICT` 以保证启动的请求永远不会被暂停。
- `enable_chunked_context`: 开启上下文分块, 会增加上下文和生成阶段之间进行批处理的机会, 从而平衡每次迭代的计算量, 提高吞吐量。

配置修改完成之后, 即可启动 Triton 推理服务。

单卡部署

对于单机部署, 使用 `tritonserver` 启动即可。

▼ bash



复制代码

```
1 docker run -it --rm \  
2 --gpus '"device=4"' \  
3 --shm-size=32g \  
4 -p 8200:8000 \  
5 -v /data/hpc/home/guodong.li/workspace:/workspace \  
6 -w /workspace \  
7 aiharbor.msxf.local/nvidia/tritonserver:24.06-trtllm-python-py3 \  
8 tritonserver --model-repository /workspace/models/qwen2-14b-trt-engine-h20-1tp \  
9 --log-info true \  
10 --log-verbose 4
```

常用参数说明:

- `--log-info`: 启用/禁用 info-level 日志记录。
- `--log-verbose`: 设置详细日志级别。0表示禁用详细日志记录，值 ≥ 1 表示启用详细日志记录。

更多参数通过 `tritonserver --help` 查看即可。

此外，`tensorrtllm_backend` 中，推荐使用

`tensorrtllm_backend/scripts/launch_triton_server.py` 进行部署，通过 `python launch_triton_server.py -h` 参考启动参数。

多卡部署

如果使用多卡进行模型推理，这里使用 `launch_triton_server.py` 进行部署。

▼ bash



复制代码

```
1 docker run -it --rm \  
2 --gpus '"device=1,2"' \  
3 --shm-size=32g \  
4 -p 8400:8000 \  
5 -v /data/hpc/home/guodong.li/workspace:/workspace \  
6 -w /workspace \  
7 aiharbor.msxf.local/nvidia/tritonserver:24.06-trtllm-python-py3 \  
8 python /workspace/tensorrtllm_backend/scripts/launch_triton_server.py \  
9 --model_repo /workspace/models/qwen2-14b-trt-engine-h20-2tp \  
10 --world_size 2 \  
11 --log --log-file /workspace/models/qwen2-14b-trt-engine-h20-2tp/sencer.log
```

常用参数说明：

- `--world_size`: 总的并行数，目前仅支持张量并行
- `--tritonserver`: 指定tritonserver脚本的路径
- `--log`: 将triton服务的状态记录到log_file中
- `--log-file`: triton日志的路径
- `--model_repo`: 指定模型仓库路径
- `--tensorrt_llm_model_name`: 指定模型仓库中 `tensorrt_llm` Triton 模型的名称，默认值: `tensorrt_llm`。如果有多个模型名称，使用逗号分隔
- `--multi-model`: 在 Triton 模型仓库中启用对多个 TRT-LLM 模型的支持

模型服务部署成功之后，即可发送HTTP请求。

▼ json



复制代码

```
1 curl -X POST 10.xxx.6.206:8400/v2/models/ensemble/generate -d \  
2 '{"text_input": "如何保持正能量?", "parameters": {"max_tokens": 100, "bad_words": [""], "stop_words":
```

常用的请求参数：

- text_input: 提示文本
- max_tokens: 生成的Token数量
- stop_words: 停用词列表
- return_log_probs: 当 true 时，在输出中包含log概率
- top_k: 采样配置参数，从模型的输出分布中选择概率最高的k个token
- top_p: 采样配置参数，从模型的输出分布中，对Token概率进行降序排列并累加,然后选择概率和首次超过 top_p 的Token集作为采样池
- temperature: 采样配置参数，使得生成的序列更加多样化或更加保守。较低的temperature值会使概率分布更加尖锐，即模型倾向于选择概率最高的几个选项；而较高的temperature值会使分布更加平坦，增加选择低概率选项的可能性

如果希望返回流式数据，则使用 /v2/models/ensemble/generate_stream 接口请求，同时设置 "stream": true 。

▼ json



复制代码

```
1 curl -X POST 10.xxx.6.206:8400/v2/models/ensemble/generate_stream -d \  
2 '{"text_input": "如何保持正能量?", "parameters": {"max_tokens": 100, "bad_words": [""], "stop_words":
```

其他常用API:

- 0.0.0.0:8000/v2/models/{name_name} : 查询模型输入输出配置信息
- 0.0.0.0:8000/v2/models/{name_name}/config : 查看模型更详细的配置信息
- 0.0.0.0:8002/metrics : 查看统计指标

此外，还可以使用提供的 [python 客户端脚本](#) 发送请求。

对于生产环境，我们通常需要部署多实例来进行负载均衡。

目前，TensorRT-LLM 后端支持两种不同的模式在多个 GPU 上运行模型：

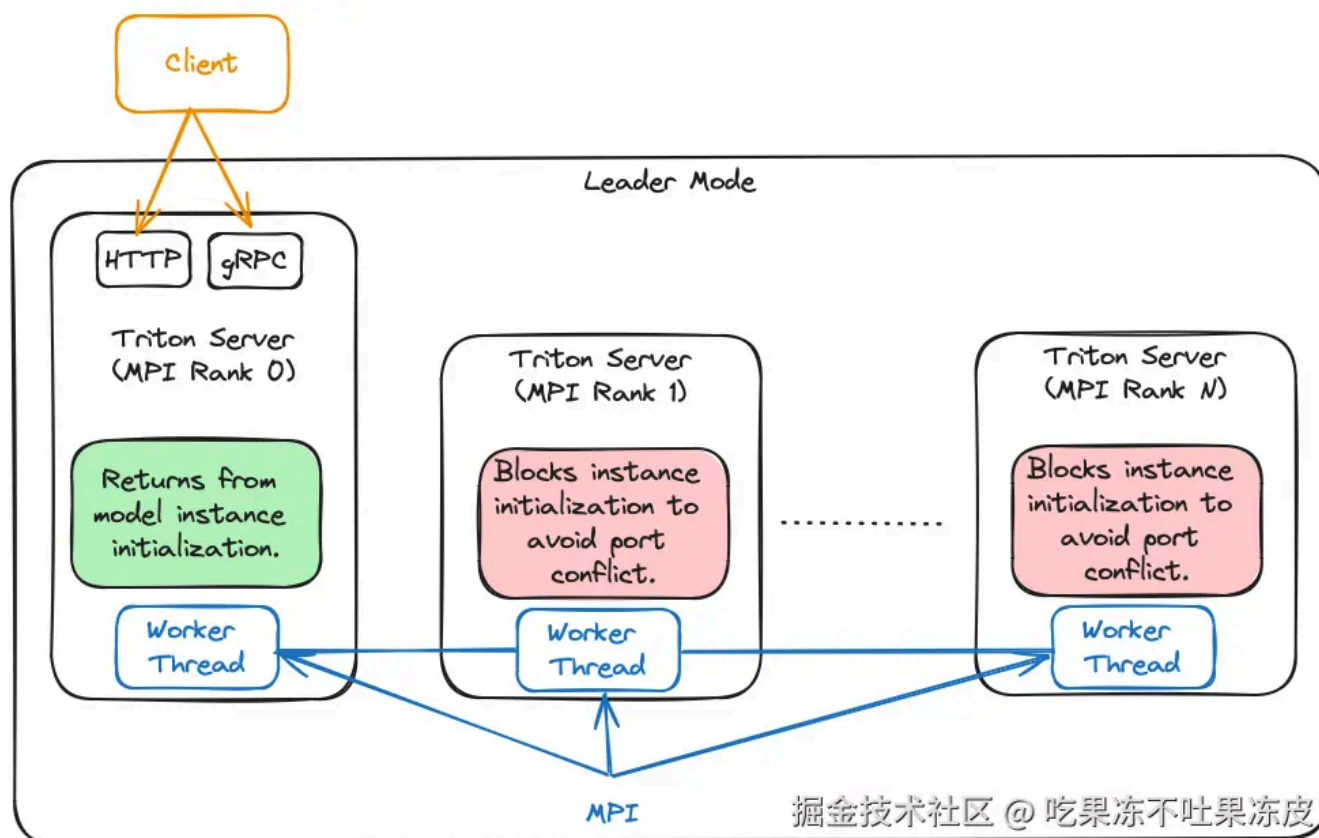
- 领导者模式
- 协调器模式

TensorRT-LLM 后端依赖 MPI 来协调跨多个 GPU 和节点的模型执行。

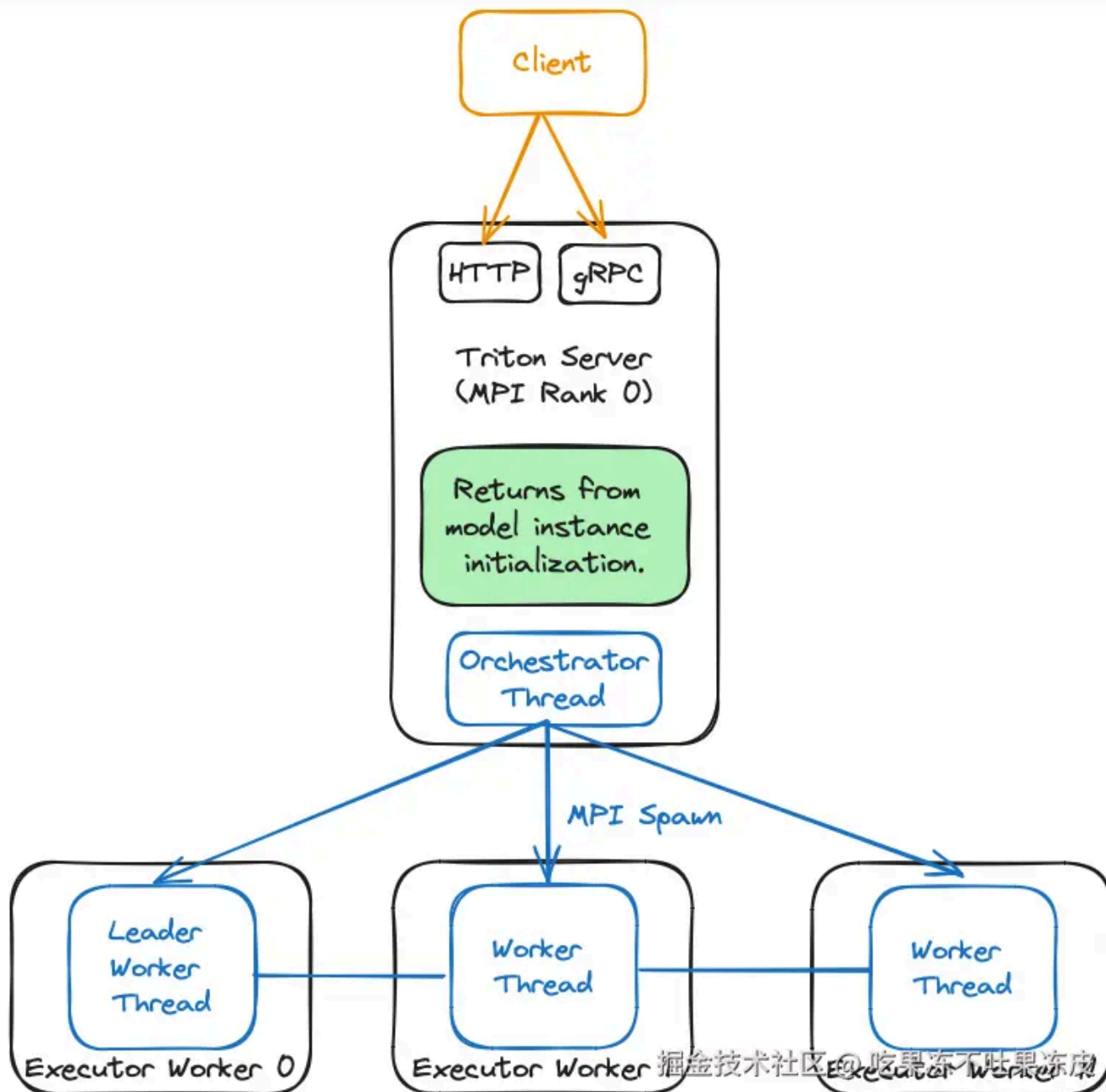
对于领导者模式，TensorRT-LLM 后端为每个 GPU 生成一个 Triton Server进程。rank 为 0 的进程是领导进程。而其他 Triton Server 进程不会

从 `TRITONBACKEND_ModelInstanceInitialize` 调用返回，以避免端口冲突并允许其他进程接收请求。

该模式不使用 `MPI_Comm_spawn`。



对于协调器模式，TensorRT-LLM 后端会生成一个充当协调器的 Triton Server 进程，并为每个模型所需的每个 GPU 生成一个 Triton Server 进程。此模式主要用于使用 TensorRT-LLM 后端服务多个模型时。在此模式下，`MPI world size` 必须为 1，因为 TRT-LLM 后端将根据需要自动创建新的工作线程。



下表总结了协调器模式和leader模式之间的区别：

	协调器模式	领导者模式
多节点支持	✗	✓
需要反向代理	✗	✓
需要客户变更，以在不同模型之间分配请求	✓	✗
需要 <code>MPI_Comm_Spawn</code> 支持	✓	✗

关于如何部署这两种模式的多实例请参考文档：[Running Multiple Instances of the LLaMa Model](#)。

结语

本文结合了 TensorRT-LLM 与 Triton Inference Server 来完成生产级 LLM 的部署工作。但相比于其他推理工具（如vLLM、LMDeploy等）来说，Triton Inference Server 集成 TensorRT-LLM 的难度相对会更高一些；同时，需要仔细设置里面的模型构建和部署参数以达到更好的推理性能。

如果觉得我的文章能够给您带来帮助，期待您的点赞收藏加关注~~

参考文档：

- [Best Practices for Tuning the Performance of TensorRT-LLM](#)**
- [TensorRT-LLM Quick Start Guide](#)*
- [TensorRT-LLM Qwen](#)*
- [End to end workflow to run llama\(Tensorrtllm Backend\)](#)
- [README.md\(Tensorrtllm Backend\)](#)**
- [Generate Extension\(Triton\)](#)
- [推理请求参数\(Tensorrtllm Backend\)](#)
- [BLS模型推理请求参数\(Tensorrtllm Backend\)](#)*
- [Ensemble Models\(Triton\)](#)
- [Business Logic Scripting\(Triton\)](#)*

标签： LLM 话题： 人工智能创作者签约季

本文收录于以下专栏



大模型实践

专栏目录

大模型实践

74 订阅 · 76 篇文章

订阅

评论 0



登录 / 注册 即可发布评论!

暂无评论数据

目录

收起 ^

- 简介
- 环境准备
- 模型格式转换及编译
 - 单卡推理
 - 多卡张量并行推理
 - FP8量化
- 使用 Triton 进行服务化部署
 - 单卡部署
 - 多卡部署
 - 发送请求
- 📄 目录

相关推荐

LangChain之数据库操作：通过链Chain和代理Agent查询数据库信息

207阅读 · 3点赞

基于本地知识库，定制一个私有GPT助手，不能再简单了

8.2k阅读 · 41点赞

OpenAI 创始成员创办「AI+教育」公司；谷歌发布 Magic Insert：让人物完美融入新背景 | RTE 开发者日报

54阅读 · 0点赞

每周AI论文速递（240722-240726）

88阅读 · 1点赞

为你推荐

TensorRT-LLM保姆级教程（一）-快速入门

吃果冻不吐果冻皮 10月前 869 1 评论 LLM

TensorRT-LLM保姆级教程（二）-离线环境搭建、模型量化及推理

吃果冻不吐果冻皮 10月前 1.8k 点赞 评论 LLM

大模型低显存推理优化-Offload技术

吃果冻不吐果冻皮 1月前 747 15 1 LLM

大模型推理服务调度优化技术-Continuous batching

吃果冻不吐果冻皮 1月前 434 8 评论 LLM

大模型推理优化技术-KV Cache

吃果冻不吐果冻皮 4月前 3.6k 10 3 LLM

大语言模型推理提速：TensorRT-LLM 高性能推理实践

阿里云云原生 8月前 1.0k 1 评论 云原生 容器

思维骨架SoT如何提升LLM的速度？ | 论文解读

Conqueror712 1年前 1.3k 4 3 LLM 人工智能

大模型国产化适配9-LLM推理框架MindIE-Service性能基准测试

吃果冻不吐果冻皮 4月前 1.7k 2 3 LLM

大模型推理框架概述

吃果冻不吐果冻皮 11月前 5.3k 2 评论 LLM

CodeDevMaster

1月前

572

5

评论

LLM

基于 Ray 的大规模离线推理

字节跳动云原生计算

1年前

2.0k

点赞

评论

分布式

云原生

大模型国产化适配3-基于昇腾910使用ChatGLM-6B进行模型训练

吃果冻不吐果冻皮

1年前

3.4k

18

3

人工智能

大模型量化技术原理：FP6

吃果冻不吐果冻皮

11天前

80

点赞

评论

LLM

利用大模型构造数据集，并微调大模型

ZackSock

4月前

413

7

评论

算法

单个4090可推理，2000亿稀疏大模型「天工MoE」开源

机器之心

3月前

521

点赞

评论

LLM

资讯