Tutorial on vLLM Framework:

https://github.com/vllm-project/vllm/tree/9587b050fba00c3c35da05d3512bf7e351914a50

2024 年 10 月

Xu Zhang

Abstract:

大语言模型部署的常见情景是部署在服务器上, 处理多个用户发送的问题. 比如 ChatGPT 在 OpenAI 的服务器上回答世界各地的用户发给它的问题. 但是大模型的高算力,大储存空间的要求使得诸多企业难以承担大模型落地部署的成本. 以 vLLM 为代表的新兴大模型优化框架可以提升大模型的使用表现, 使得大模型部署更简单可行. 为了部署 vLLM 使大模型方案有更好的表现, 需要研究搭载了 vLLM 的大模型在处理并发请求时的性能. 同时, 将 vLLM 的测试结果和不使用 vLLM 的同一模型的运行结果进行对比并分析结果异同. vLLM 把模型包装成一个 vLLM 自己的类的实例来运行. 当多个请求希望同时调用同一个实例时(即并发的情况), vLLM 的 AsyncLLMEngine 类生成的实例(也就是搭载了 vLLM 的大模型)可以把同时过来的请求放到一个队列里, 然后一个一个处理请求. 为此, 我们要深入研究 vLLM 具体是怎么处理这些并发的请求的. 为了接收并发的请求, 我们需要搭建一个服务器并将大模型放到这个服务器上, 通过服务器接受请求. 不使用 vLLM 的 Hugging Face 模型(以下简称 HF 模型). 为此, 我们需要给服务器加 Lock, 实例不具有处理多个请求同时调用的能力使得请求一个一个送给 HF 模型. 另外, 我们也需要考虑多个请求并发给服务器的情况.

壹. Prerequisite:

vLLM 是大语言模型的一种架构, 可以提升模型的性能. vLLM 主要通过他们特有的 Paged Attention 算法提升模型性能.

这里可能涉及到的几个念自然语言处理(NLP)的概: seq-to-seq(也被称作 encoder-decoder)模型, attention, transformer, KV-Cache. 如果还不清楚相关概念, 可以看一下绍介的视频:

- Seq-to-seq 模型和 Attention 机制一起讲:Stanford CS224N NLP with Deep Learning | Winter 2021 | Lecture 7 - Translation, Seq2Seq, Attention
-
- Attention 机制: Attention for Neural Networks, Clearly Explained!!!
- 【機器學習 2021】自注意力機制 (Self-attention) (上)
- 【機器學習 2021】自注意力機制 (Self-attention) (下)
-
- Transformer:【機器學習 2021】Transformer (上)
- 【機器學習 2021】Transformer (下)
-
- KV Cache:The KV Cache: Memory Usage in Transformers

贰. vLLM 入门:

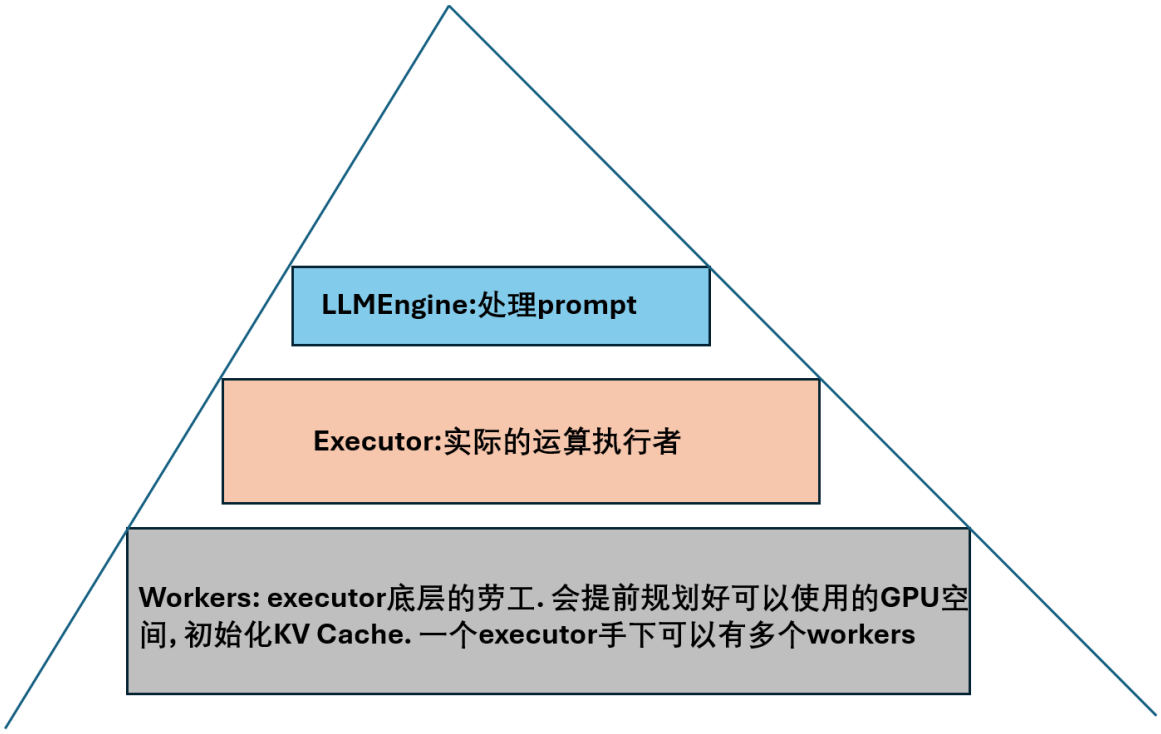
这里是 vLLM 官方的介绍视频:Fast LLM Serving with vLLM and PagedAttention

vLLM 官方网站:
https://docs.vllm.ai/en/latest/

如何安装 vllm:
https://docs.vllm.ai/en/latest/getting_started/installation.html

vLLM 的代码到底在干什么?
vLLM 常用的类有两个, LLM 和 AsyncLLMEngine. 这两个类给大模型(Hugging Face 的 transformer 模型)加入了 vLLM 的框架,然后我们用户就可以把使用了 vLLM 框架的大模型作为一个 LLM 类(或者 AsyncLLMEngine 类)的实例来处理. 这两个类的基础用法思路和 HF 的模型思路是类似的.

vLLM 框架的核心是 vLLM 代码里面的'LLM engine'.之前提到的两个常用的类(LLM 和 AsyncLLMEngine)都是 LLM engine 的 wrapper. 在具体执行 attention 机制等运算时, 实际是 LLM engine 里面的 executor 在跑. 而 executor 的底层基础则是一个个的 workers. 于是就有了这么一个等级图:



也就是说, 每次我们创建一个 LLM 或者 AsyncLLMEngine 的实例, 我们实际上是激活了一个 LLMEngine. 这个 LLMEngine 通过 call 其内置的 step()函数来进行推理.

当我们创建了一个 Engine 的实例, 这个实例名下的 executor, 以及 executor 名下的 workers 都会被创建. 当 worker 被创建时, worker 会计算该设备(我们的情况就是 gpu)的剩余可用 gpu 空间. 然后把 gpu 剩余空间除以 block size(详见之前 vllm 的介绍视频. 数据会被存入 block 里面)得到可以支配的总 block 数量. 然后根据可用 block 数量初始化 KV cache. 所以这也意味着 vllm 会提前预支我们的 gpu 空间来初始化自己的框架(比如用于 KV cache), 即使这些被预留的空间并没有被使用. 默认情况下, vllm 会直接把 90%的 gpu 空间预留给自己使用. 我们可以通过 GPU_MEMORY_UTILIZATION 调节 vllm 占用空间的比例.

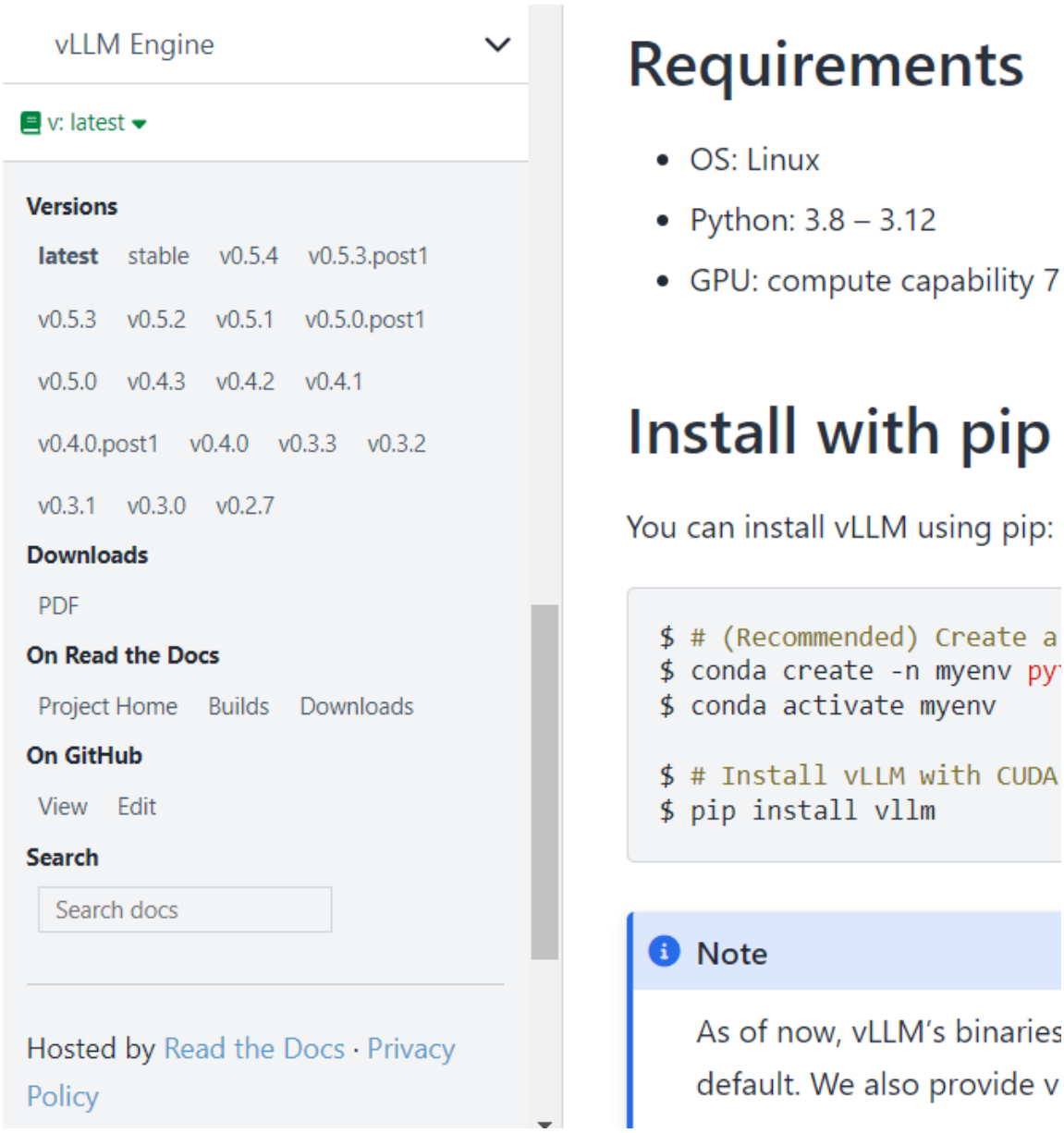
vLLM 以 block 为单位存储 token. 当一个 block(比如叫做 block A)被存满时, vLLM 的 scheduler(类似一个安排房间的管家)会把需要存储的数据(比如 KV cache 中的 key) 引导到下一个有空余地方的 block(block B)在新的地方存储. 然后我们读取数据的时候就用 scheduler 提供的一个 table(此 table 存储了 block A 和 block B 的地址, 还有每个 block 存了多少东西)找到 block A 和 block B 的地址, 然后把数据提取出来. 更具体的示例可以参见 vllm 的介绍视频.

通过 paged attention 有效的空间管理, vllm 可以为我们节省存储空间的使用, 并提高模型的运行效果. 为了保证 vllm 的高效率运行, vllm 项目组使用了自己的 attention 运算的 implementation:
https://docs.vllm.ai/en/latest/dev/kernel/paged_attention.html#concepts

主要的 idea 就是适配 paged attention, 把大段的词句 matrix 分给更小的存储单位(block), 而不是传统地占用连续的存储空间, 以此节省空间. 将切片切块的更小的数据单位分别进行 attention 的运算然后再拼回来. 理论上会得到与正常 attention 运算(比如 pytorch 的 implementation)完全一样的结果. 但是在实践上, 因为计算机不完美的数据存储能力/浮点数的 loss of precision, 我们可能会得到和正常的 attention implementation 不一样的运算结果. 也可能导致使用 vllm 和不使用 vllm 的输出有差别.

Remark:

注意: vLLM 是有版本差别的. 阅读文档的时候请选择较新的版本阅读.



一些底层的东西在不同版本有差别. 比如从 2023 年的 0.3 到 2024 的 0.5, 最大的差别之一就是加入了 executor 的概念. 具体可见 vllm 的 github 源代码 vllm/vllm/engine/llm_engine.py 中定义 GPUExecutor 的部分和 vllm/vllm/worker/worker.py .

那么, vLLM 常用两个类 LLM 和 AsyncLLMEngine 有什么区别呢? 什么时候该用 AsyncLLMEngine 呢? LLM, 正如其名, 创建了一个和 HF 模型(Hugging Face 简称 HF)性质很类似的实例. 和 HF 模型的实例一样, 当多行代码同时调用 LLM 类实例时, LLM 类实例会报错. 这是因为 LLM 类和 HF 模型一样, 没有做针对并发请求的相关处理. 所以如果用这个类部署 vLLM 大模型, 只能在服务器端对并发的请求进行一些处理, 使得请求不会一次性发给 LLM 类实例.

这里, vLLM 官网有 LLM 示例代码的示例:https://docs.vllm.ai/en/latest/getting_started/examples/offline_inference.html

与之对应的, AsyncLLMEngine 类的实例具有处理并发请求的能力. 当多个请求同时发给 AsyncLLMEngine 实例时, 实例会先把请求加入一个队列, 然后按一定顺序把请求给 vLLM 的引擎处理. 所以在使用 AsyncLLMEngine 实例的场景下, 我们不需要专门给并发请求做特殊处理.

这里是 AysncLLMEngine 的官方 API 服务器示例:
https://github.com/vllm-project/vllm/blob/9587b050fba00c3c35da05d3512bf7e351914a50/vllm/entrypoints/api_server.py#L14

AsyncLLMEngine 更具体的使用
创建 AsyncLLMEngine 实例的方法如示例所示. 一般是 AsyncLLMEngine.from_engine_args(). 在这里, from_engine_args()是 vLLM 为 AsyncLLMEngine 类量身定制的初始化方法. 定义如下:

```python
@classmethod
def from_engine_args(
    cls,
    engine_args: AsyncEngineArgs,
    start_engine_loop: bool = True,
    usage_context: UsageContext = UsageContext.ENGINE_CONTEXT,
    stat_loggers: Optional[Dict[str, StatLoggerBase]] = None,
) -> "AsyncLLMEngine":
    """Creates an async LLM engine from the engine arguments."""
    # Create the engine configs.
    engine_config = engine_args.create_engine_config()

    if engine_args.engine_use_ray:
        from vllm.executor import ray_utils
        ray_utils.assert_ray_available()

    executor_class = cls._get_executor_cls(engine_config)

    # Create the async LLM engine.
    engine = cls(
        executor_class.uses_ray,
        engine_args.engine_use_ray,
        **engine_config.to_dict(),
        executor_class=executor_class,
        log_requests=not engine_args.disable_log_requests,
        log_stats=not engine_args.disable_log_stats,
        start_engine_loop=start_engine_loop,
        usage_context=usage_context,
        stat_loggers=stat_loggers,
    )
    return engine
```

这里的 cls(不了解可以 google 一下)是指的 AsyncLLMEngine 这个 class 本身. 也就是类似 self, 但是又有重要的差别. 在这里, 我们可以简单地理解为 ”engine = cls(...) = AsyncLLMEngine(...)” 也就是说 from_engine_args()是 AsyncLLMEngine 正常初始化方法的一个 wrapper.

from_engine_args 可以接受 AsyncEngineArgs 的输入. 这里受 AsyncEngineArgs 可以简单理解为 vLLM 帮我们把正常的 engine argument 又处理了一下. 所以 AsyncEngineArgs 的输入就是我们的 engine 参数. 比如让 engine 不要使用 ray"engine_use_ray = False".

所以就形成了如下的用法示例:

```
engine = AsyncLLMEngine.from_engine_args(
  AsyncEngineArgs(
    model="你的本地模型路径",
    max_model_len=1000,
    engine_use_ray = False,  #模型引擎不用 ray
    worker_use_ray=False,  #模型 worker(前文提到的那个 worker)不用 ray
    trust_remote_code=True ) )
```

其他 vLLM 的示例则可以看这里:https://github.com/vllm-project/vllm/tree/main/examples

叁. 服务器和客户端:

我们的测试需要多线程的处理. 这就需要一个可以应答多线程的服务器. 计算机网络里面不同设备的交流在 5 个层面展开从最底层的设备层, 再到最上层的 app 层. 一个 app 的里面信息(我们运行的大模型本质上也是个 app)要发送出去就需要 app 的信息和外部信息交换的窗口. 这个窗口叫做 socket. Python 有相关的 socket 库可以实现这种底层的运行. 但是实际上 socket 过于底层, 以至于我们要处理太多的细节. 这不仅对计算机网络有较高的要求, 也要求很大

的工作量. 所以我们使用 asgi/wsgi 的框架. Asgi 是 python 官方钦定的用 python 写的 app 和外部信息交换的框架标准. 常见的使用这个标准的框架有 FastAPI 和 Flask. Flask 上手稍微简单一点, 也是更传统的选择. FastAPI 属于近些年的后起之秀. vLLM 官方的 api server 搭建使用 FastAPI. RAY 框架(大模型的另一种框架)也是和 FastAPI 更兼容.

我们最初搭建的时候使用的 Flask 上手. 后来因为 vLLM 的官方示例用的 FastAPI, 于是改用的 FastAPI.

这些网络框架的重要 feature 在于 routing. 简单的说一个 route 可以定义该服务器的一个功能并把这个功能上线在一个 url. 客户端就可以去这个对应的 url 访问并使用该 route 所对应的这个功能.

```
@app.get("/server_info")
def server_info():
    return "server_FastAPI_no_vLLM"
```

比如这里我们定义了一个叫做”server_info”的 route. 这个 route 会上线在”www.主站的网址/server_info”的 url 网页里. 访问这个 url 会返回服务器的名字”server_FastAPI_no_vLLM”.

肆. Q&A:

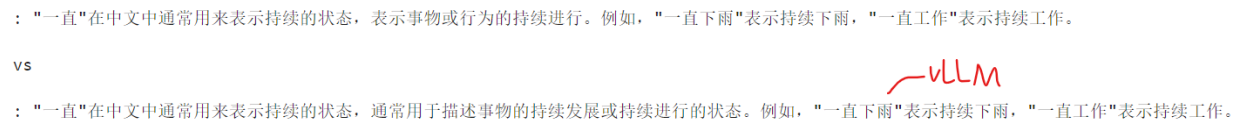
1. vLLM 不同的注意力机制会对模型输出产生影响.

考虑这些参数:

```
sampling_params = SamplingParams(use_beam_search = False,
temperature=0.01, top_p = 1, top_k = 1, max_tokens = 1000, min_tokens = 0,
repetition_penalty = 1, length_penalty = 1, seed = 42),

engine = AsyncLLMEngine.from_engine_args(
   AsyncEngineArgs(
      model="/root/qwen2_ins",
      max_model_len=1000,
      engine_use_ray = False,
      worker_use_ray=False,
      trust_remote_code=True
   )
)
```

对大模型问复杂的问题. 可以注意到大模型的输出在使用 vllm 和不使用 vllm 的时候不同.



我们可以看到, 这里 sampling_params 已经设定了 temperature=0.01, top_p = 1, top_k = 1, 并且有了固定的 seed. 我们的测试也显示输出是稳定的(同一个问题有同样的回答).

那么输出不一样的原因在哪呢?

可以参考这个 github 问题:

https://github.com/QwenLM/Qwen2/issues/695#issuecomment-2292675351

之前有讲过, vLLM 有自己的 attention 运算代码. 这会导致 attention 算出来的结果有差别.

另外 vLLM 和 HF 模型关闭 sampling 的方式也不一样. vLLM 是通过给 SamplingParams 设定 seed. HF 模型实在 model.generate 这里加一个”do_sample=False”的参数. 可以看到, 一个是特定的数(seed), 一个是没有选择的”do_sample=False”. 所以这可能会导致初始的状态有差别, 从而导致不同的结果. 当然这一点我们没有进行具体的论证, 仅是猜测以供参考.

。

2. 灵异现象! 我的小模型搭载 vLLM 运行时 GPU 空间占用达 90%!

vllm 初始化 worker 的时候会先计算手头可以使用的 gpu_block. 然后将所有可支配的 gpu_block 分配给 worker. 计算 gpu_blocker 需要使用 gpu_memory_utilization. 也就是说 vllm 把这么多的空间提前预留给了 worker 用. 这也意味着 vllm 显存空间的占用不是动态的. 我们只能通过 gpu_memory_utilization 改变其占用的显存. gpu_memory_utilization 默认占用显卡 90%的空间, 所以初始化时如果没有这条, 我们会有 90%的显存占用.

3. 每个 vLLM 引擎实例只有 1 个 worker 吗? 如果我的 engine 有多个 workers,怎样才能限制 worker 数量为 1?

一个 executor 是负责 carry out LLMengine 运算的单位。而一个 executor 可以有一个 worker 或者多个 workers。这取决于 executor 的种类。GPU_executor 只能有一个 worker(即 driver_worker). 而 ray_gpu_executor 则可以有多个 ray_worker.其中在 driver node 的那个会变成 driver node 的 resource holder. (源代码见:vllm/vllm/executor/ray_gpu_executor.py).

至于限制 worker 数量为 1 的事情, 在创建的 Async LLM engine 的时候, 确保使用的设备是 gpu, 并且 在 AsyncLLMEngine.from_engine_args(

AsyncEngineArgs())的参数设置中将 engine_use_ray = False. 如此,Engine 初始化的 executor 就会是一个 GPUExecutor. 而 GPUExecutor 是通过创建一个”worker”来实现的. vLLM 的 worker 会占用整张 gpu. 如此,在单卡情况下只会有一个 worker.

4. vLLM 处理多线程时会出现输出不稳定的情况.

考虑这些参数:

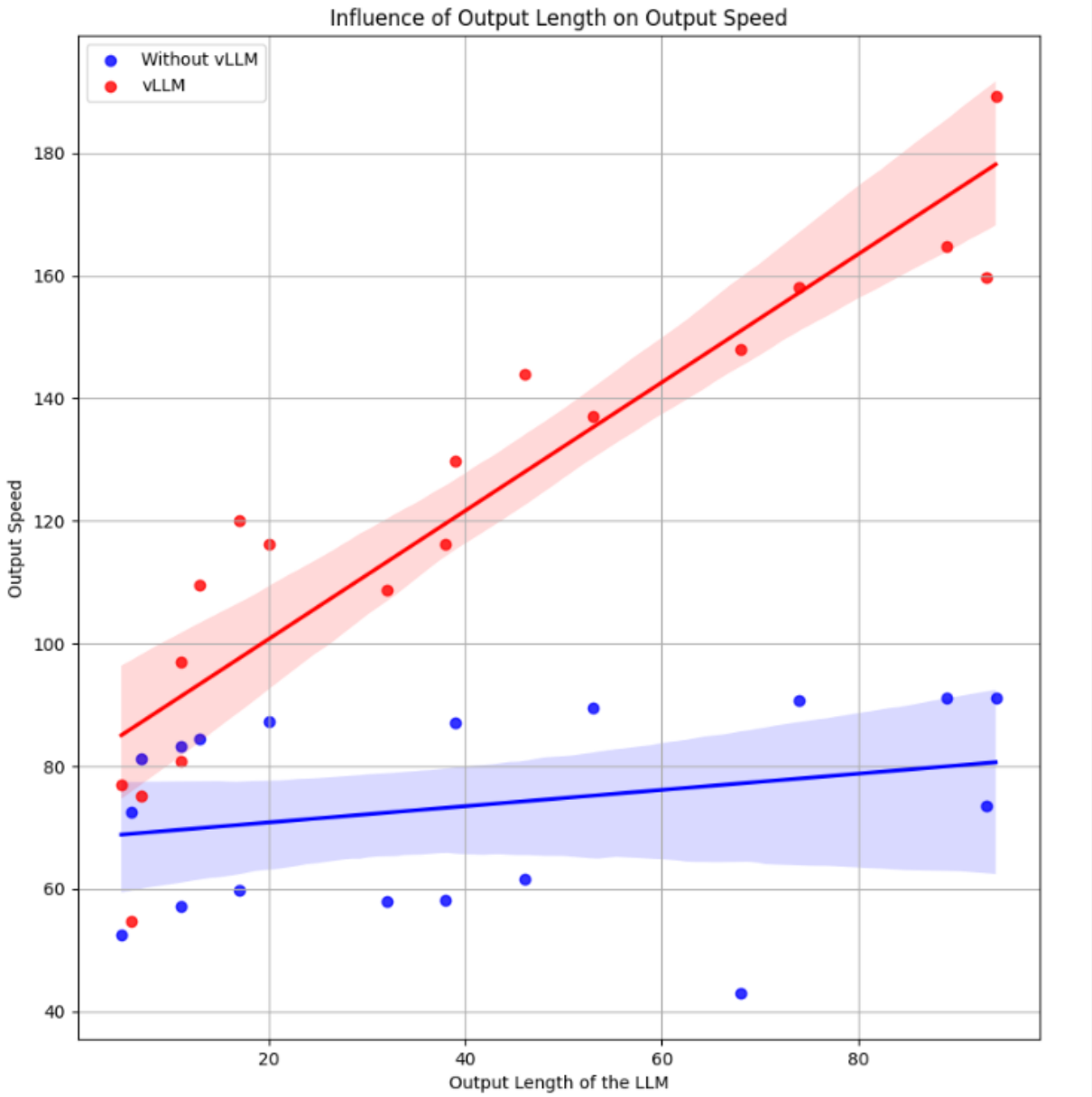
```
sampling_params = SamplingParams(use_beam_search = False,
temperature=0.01, top_p = 1, top_k = 1, max_tokens = 1000, min_tokens = 0,
repetition_penalty = 1, length_penalty = 1, seed = 42),

engine = AsyncLLMEngine.from_engine_args(
   AsyncEngineArgs(
```

```
        model="/root/qwen2_ins",
        max_model_len=1000,
        engine_use_ray = False,
        worker_use_ray=False,
        trust_remote_code=True
    )
)
```

这些参数 thread = 1 的情况下可以保证 vLLM 的模型对同一个问题输出同样的回答. 但是, 对搭载了这些参数的 vLLM 大模型服务器进行 thread 为 3 的同时访问, 多次输入同样的问题:”您如何跟踪和应用行业内的新技术和新标准？如何确保您的技能始终与时俱进?” 大模型会输出:

5. vLLM 和不用 vLLM 处理多线程的效果对比

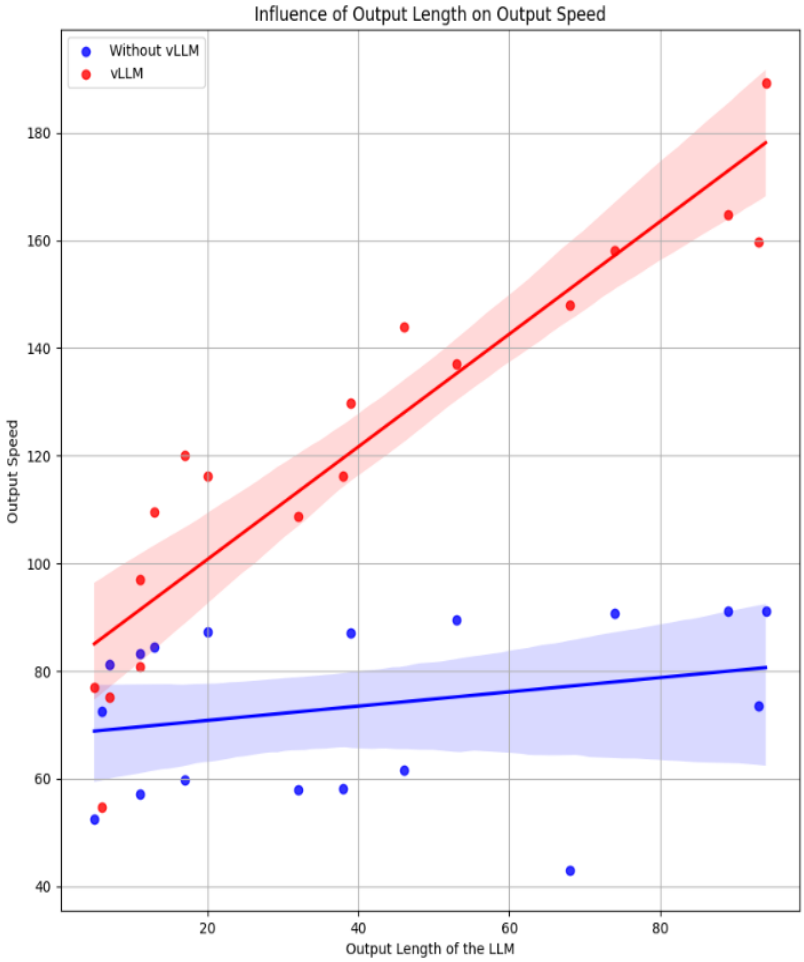


单线程时的稳定输出如此:

```
******************************************
喂给大模型的prompt是:
您如何跟踪和应用行业内的新技术和新标准？如何确保您的技能始终与时俱进？
******************************************
200
{'response': '\n作为一个人工智能，我并没有实际的行业经验或技能，因此无法提供具体的跟踪和应用新技术和新标准的建议。但是，我可以提供一些通用的建议，帮助您更好地跟踪和应用新技术和新标准：\n\n1. 保持学习：不断学习新技术和新标准是保持竞争力的关键。可以通过阅读专业书籍、参加在线课程、参加行业会议等方式来获取最新的知识和技能。\n\n2. 保持开放的心态：新技术和新标准的出现往往伴随着新的挑战和机遇。保持开放的心态，不断尝试新事物，以适应变化。\n\n3. 保持专业性：在应用新技术和新标准时，保持专业性非常重要。这包括了解行业标准、遵守法律法规、确保数据安全等。\n\n4. 保持灵活性：新技术和新标准的出现往往伴随着新的挑战和机遇。保持灵活性，能够快速适应变化，以应对新的挑战。\n\n5. 保持沟通：在应用新技术和新标准时，保持沟通是非常重要的。与行业内的专家、同事和客户保持沟通，以获取最新的信息和建议。\n\n总的来说，跟踪和应用新技术和新标准需要持续的学习、开放的心态、专业性、灵活性和沟通。', 'total_time': 0.9495296478271484, 'start_time': 1723794954.1994984, 'end_time': 1723794955.149028, 'output_speed': 258.02248572295196, 'length_of_output': 435, 'speed_result': '[0.95s/it, est. start_time: at 1723794954.20 , output: 258.02 toks/s]', 'task_id': '2JzyRzPc', 'idx': 10}
```

6. vLLM 和不用 vLLM 处理多线程的效果对比

对大模型输出长度和大模型输出速度的关系进行了测试. 如下图所示:



此图展现了输出长度和输出速度的关系. 其中 x 轴为输出长度, y 轴为输出速度. 我们可以看到在所有长度上, vLLM 的速度都明显快于不用 vLLM. 并且随着输出长度的增加, vLLM 的速度大致以斜率为 1 的回归直线呈线性增长. 与之对应的, 不用 vLLM 的增长就要慢很多. 这显示出了, 在更长输出的情况下, vLLM 的速度提升优势更加明显.

以上就是教程的全部, 希望对各位水友有帮助.