知乎

lightllm代码解读——三进程异步协作

robindu

关注

LLM（大模型）技术的发展日新月异，不停有新东西涌现。LLM的推理部署技术也吸引了很多关注，LightLLM是其中受到比较多关注的一个。本篇文章介绍其中的三进程异步协作设计（Tri-process asynchronous collaboration），LightLLM将tokenization、model、detokenization三个主要计算部分拆开处理。因为整条链路的计算涉及到CPU+GPU这样的异构计算，将计算链路的各个部分拆分到不同进程处理，有可能使得各部分更加协调，提升机器利用率。

一、概览

在lightLLM github官方主页上的主架构图比较好的说明了LightLLLM中各个组件的划分。router是居中调度的模块，计算相对密集的就是tokenization、model inference、detokenization三个部分。其中tokenization在httpserver中直接完成，router使用独立的进程启动mp.Process，model使用rpyc部署相当于也是隔离开的进程，detokenization也使用独立进程启动。所以对应于官方说的三进程概念，应该是将router+model划归为一个进程所属。

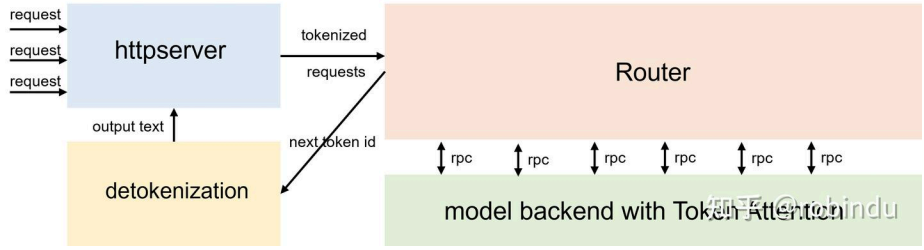

图1：LightLLM架构图

二、三进程结构

那么LightLLM是在哪里启动的三进程，不同进程间又是如何通信的呢？

这里可以先从lightllm/server/api_srever.py中开始。主要的结构如下所示，这里列出了三进程结构的启动过程。其中httpserver相当于主进程，负责连接客户端client与server内部，可以看做由fastapi app控制。而router、detokenizer则都通过典型的multiprocessing.Process来启动进程。三个进程间使用了管道Pipe通信，在httpserver主进程中保留消息接受者reader，消息生成者writer作为参数传入到router和detokenizer的进程中。在router/manager.py中可以看到这句代码pipe_writer.send('init ok')，router子进程在初始化完成后会发送信息传给主进程。reader和writer在生成时就确定好了配对关系，不需要特定指向。

```
global httpserver_manager
httpserver_manager = HttpServerManager(
  args.model_dir,
  ......
)
pipe_router_reader, pipe_router_writer = mp.Pipe(duplex=False)
pipe_detoken_reader, pipe_detoken_writer = mp.Pipe(duplex=False)
proc_router = mp.Process(
  target=start_router_process,
  args=(
    args,
    ...,
    pipe_router_writer,
  ),
)
proc_router.start()
proc_detoken = mp.Process(
```

```
        args,
        ...,
        args.trust_remote_code,
    ),
)
proc_detoken.start()

# wait load model ready
router_init_state = pipe_router_reader.recv()
detoken_init_state = pipe_detoken_reader.recv()
```

但三进程间主要的通信并不是依靠pipe管道来实现的。而是通过“zmq”库来进行。以httpserver与router之间的通信来作为例子，收发两侧的主要数据通信都基于此完成，pipe管道只负责进程初始化状态的沟通。

```
# httpserver
context = zmq.asyncio.Context(2)
self.send_to_router = context.socket(zmq.PUSH)
self.send_to_router.connect(f"tcp://127.0.0.1:{router_port}")
...
self.send_to_router.send_pyobj((prompt_ids, sampling_params, request_id))

# router
context = zmq.asyncio.Context(2)
self.recv_from_httpserver = context.socket(zmq.PULL)
self.recv_from_httpserver.bind(f"tcp://127.0.0.1:{router_port}")
...
recv_req = await self.recv_from_httpserver.recv_pyobj()
```

三、三进程协作

三进程间的通信渠道构建好之后，他们之间的协作是如何进行的呢？在主进程httpserver中，核心处理流程由客户端传入的请求触发，该部分触发实际由fastapi⁺框架托管。但在router和detokenizer中，触发处理的机制则完全由自己管理，设计思想就是使用asyncio构建异步死循环，在循环中一直尝试从上游环节await recv消息，一旦接收到上游下发的任务信息，在启动处理流程，处理完成后再通过tcp通信返回处理结果，如下图所示。

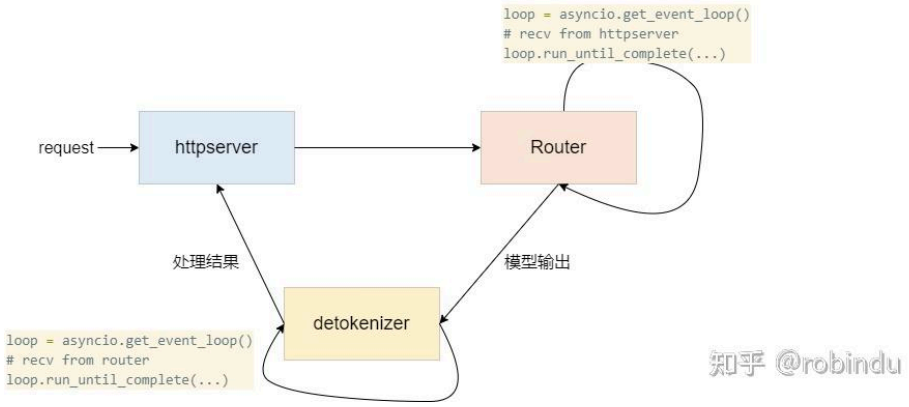

图2：三进程关系图

由于在三进程协作中使用了大量的异步调用⁺，如何保障请求与结果之间的一一对应关系呢？其实异步调用本身并不会影响到结果的混乱，而只是影响返回时间。真正需要管理的是在router环节中组batch处理部分，该部分将请求放置在队列中，在处理时从队列中不断抽取新任务，实质上真正产生了输入与输出的混乱对应。具体处理过程如下图：

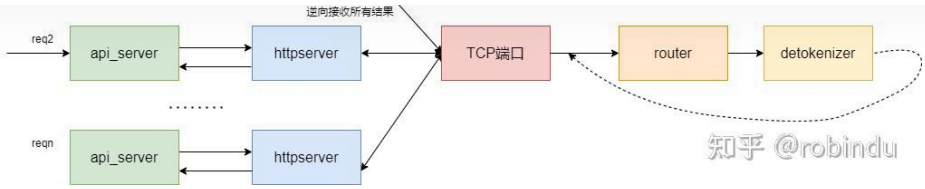

图3：三进程通信细节图

router这部分具体如何处理的呢？router中启动了两个任务，即类似生产者与消费者两个任务。生产者任务不停监听httpserver进程发送来的请求信息，并将新请求添加到req_qeue队列中。消费者任务则不停地从req_queue中抽取新请求，加入到当前正在处理的running_batch中。running_batch和req_queue其中的核心数据要素都是req请求，只是封装了其他能力。在_prefiil_batch和_decode_batch中分别将'request_id'和处理结果打包成一个dict，然后发送给detokenizer。所以本质是是通过维护的'request_id+'来完成结果与输入的对应。

```
# lightllm/server/router/manager.py
#生产者任务
async def loop_for_netio_req(self):
  while True:
    recv_req = await self.recv_from_httpserver.recv_pyobj()
    if isinstance(recv_req, tuple) and len(recv_req) == 3:
      prompt_ids, sampling_params, request_id = recv_req
      self.add_req(prompt_ids, sampling_params, request_id)
     ....

#消费者任务
async def loop_for_fwd(self,):
  counter_count = 0
  while True:
    await self._step()
    ...

async def _step(self):
  ...
  await self._decode_batch(self.running_batch)
```

但是router和detokenizer中使用'request_id'保持的输入输出对应关系，并不能使得在最外层对接client的server接口处能将正确的结果返回。这部分的逻辑要看api_server.py中的generate_stream函数，以及httpserver中的manager.py。

```
# lightllm/server/api_server.py
@app.post("/generate_stream")
async def generate_stream(request: Request) -> Response:
  ...
  request_id = uuid.uuid4().hex
  results_generator = httpserver_manager.generate(prompt, sampling_params, request_id)

 # lightllm/server/httpserver/manager.py
async def generate(self, prompt, sampling_params, request_id):
  while True:
    ...
    out_str, metadata, finished, _ = self.req_id_to_out_inf[request_id]
    # 等于将监听到的信息按照request_id分桶，然后区别返回，这样在api_server.py这个对接client的
    if len(metadata) != 0:
      self.req_id_to_out_inf[request_id] = ("", {}, finished, event)
      metadata["prompt_tokens+"] = prompt_tokens
```

四、总结

分不能很好的协调搭配，导致某一方面出现瓶颈。LightLLM中将模型前处理、模型、模型后处理分为三个进程管理，将异构运算的部分通过进程隔离开，彼此间通过网络通信。这样的做法可以解构开异构运算资源的耦合性，更高效的利用机器。

五、参考资料

[1] github.com/ModelTC/ligh...

其他系列文章链接

第一弹：lightllm代码解读——模型推理

第二弹：lightllm代码解读番外篇——triton kernel撰写

第三弹：lightllm代码解读——Router

第四弹：lightllm代码解读——三进程异步协作

第五弹：lightllm代码解读——显存管理机制

编辑于 2023-11-20 10:14 · IP 属地浙江

LLM（大型语言模型） AI技术 异步