LLM Engine

LLM engine为vLLM引擎的主类，它接收用户请求并利用LLM生成文本。LLM engine包含了一个tokenizer，一个大模型并记录了分配给KV cache的GPU内存

1. 相关配置(config的参数派生自EngineArgs类)
    - model_config：LLM模型的参数
    - cache_config：KV cache管理的相关配置
    - parallel_config：分布式执行相关配置
    - scheduler_config：请求调度器配置
    - device_config：设备相关配置
    - placement_group：模型放置的机器
    - log_stats：日志状态
2. _init_cache

    该函数评估可用的内存总量并计算出最大可分配的GPU块和CPU块数量，并基于此初始化各个worker的KV cache。当存在多个worker时，该函数取所有worker可用块的最小值，保证所有worker的块的分配

```
def _init_cache(self) -> None:
    # 各个worker调用profile_num_available_blocks函数获得所有worker可用块的数量
    num_blocks = self._run_workers(
        "profile_num_available_blocks",
        block_size=self.cache_config.block_size,
        gpu_memory_utilization=self.cache_config.gpu_memory_utilization,
        cpu_swap_space=self.cache_config.swap_space_bytes,
        cache_dtype=self.cache_config.cache_dtype,
    )

    # 可用块设置为所有worker的最小值
    num_gpu_blocks = min(b[0] for b in num_blocks)
    num_cpu_blocks = min(b[1] for b in num_blocks)
    logger.info(f"# GPU blocks: {num_gpu_blocks}, "
                f"# CPU blocks: {num_cpu_blocks}")

    if num_gpu_blocks <= 0:
        raise ValueError("No available memory for the cache blocks. "
                         "Try increasing `gpu_memory_utilization` when "
                         "initializing the engine.")

    # 计算出可支持的最大的序列长度
    max_seq_len = self.cache_config.block_size * num_gpu_blocks
    if self.model_config.max_model_len > max_seq_len:
        raise ValueError(
            f"The model's max seq len ({self.model_config.max_model_len}) "
            "is larger than the maximum number of tokens that can be "
            f"stored in KV cache ({max_seq_len}). Try increasing "
            "`gpu_memory_utilization` or decreasing `max_model_len` when "
            "initializing the engine.")

    self.cache_config.num_gpu_blocks = num_gpu_blocks
    self.cache_config.num_cpu_blocks = num_cpu_blocks

    # Initialize the cache.
    self._run_workers("init_cache_engine", cache_config=self.cache_config)
    self._run_workers("warm_up_model")
```

3. abort_request

    该函数根据给定的请求id，将请求移出。该操作主要通过调用scheduler对应的函数实现

4. add_request

    将一个请求添加到请求池中，并由调度器进行请求调度

```
def add_request(
    self,
    request_id: str,
    prompt: Optional[str],# prompt的字符串，如果已经提供了prompt_token_ids时可以为None
    sampling_params: SamplingParams,# 文本生成时的sampling参数
```

```
    prompt_token_ids: Optional[List[int]] = None,
    arrival_time: Optional[float] = None,# 请求到达时间
    lora_request: Optional[LoRARequest] = None,
) -> None:
    if lora_request is not None and not self.lora_config:
        raise ValueError(f"Got lora_request {lora_request} but LoRA is "
                         "not enabled!")

    # logprobs表示什么
    max_logprobs = self.get_model_config().max_logprobs
    if (sampling_params.logprobs
            and sampling_params.logprobs > max_logprobs) or (
                sampling_params.prompt_logprobs
                and sampling_params.prompt_logprobs > max_logprobs):
        raise ValueError(f"Cannot request more than "
                         f"{max_logprobs} logprobs.")

    if arrival_time is None:
        arrival_time = time.monotonic()

    # 将prompt进行编码，得到prompt_token_ids序列
    prompt_token_ids = self.encode_request(
        request_id=request_id,
        prompt=prompt,
        prompt_token_ids=prompt_token_ids,
        lora_request=lora_request)

    # 创建token序列
    block_size = self.cache_config.block_size
    seq_id = next(self.seq_counter)
    eos_token_id = self.tokenizer.get_lora_tokenizer(
        lora_request).eos_token_id
    seq = Sequence(seq_id, prompt, prompt_token_ids, block_size,
                   eos_token_id, lora_request)

    # 拷贝参数避免后续修改影响到sampling_params
    sampling_params = sampling_params.clone()

    # 每个请求可能会生成多个序列，用一个序列组来对应请求涉及到的所有序列
    seq_group = SequenceGroup(request_id, [seq], sampling_params,
                              arrival_time, lora_request)
    self.scheduler.add_seq_group(seq_group)
```

5. step

   执行一次iteration并返回新产生的token，其主要运行流程为

   - Step 1：调度在下一个iteration需要执行的序列(可能会进行抢占或重排)以及获得需要执行操作的tokens block(包括块的换入、换出和复制等)
   - Step 2：通知所有workers执行模型并产生输出
   - Step 3：处理模型输出，包括解码输出、更新被调度的序列组(beam_search等)、释放执行完成的序列组

```
def step(self) -> List[RequestOutput]:
    seq_group_metadata_list, scheduler_outputs = self.scheduler.schedule()

    if not scheduler_outputs.is_empty():
        # Execute the model.
        all_outputs = self._run_workers(
            "execute_model",
            driver_kwargs={
                "seq_group_metadata_list": seq_group_metadata_list,
                "blocks_to_swap_in": scheduler_outputs.blocks_to_swap_in,
                "blocks_to_swap_out": scheduler_outputs.blocks_to_swap_out,
                "blocks_to_copy": scheduler_outputs.blocks_to_copy,
            },
            use_ray_compiled_dag=USE_RAY_COMPILED_DAG)

        # Only the driver worker returns the sampling results.
        output = all_outputs[0]
    else:
        output = []

    return self._process_model_outputs(output, scheduler_outputs)
```

PagedAttention

vLLM利用自己实现的多头注意力机制kernel以兼容其paged KV caches，该kernel依赖于特别设计的内存布局和访问策略以实现高吞吐量，特别是线程将数据从全局内存读取到共享内存

1. kernel function input

```
template<
typename scalar_t,
int HEAD_SIZE,
int BLOCK_SIZE,
int NUM_THREADS,
int PARTITION_SIZE = 0>
__device__ void paged_attention_kernel(
const scalar_t* __restrict__ out,
const scalar_t* __restrict__ q,
const scalar_t* __restrict__ k_cache,
const scalar_t* __restrict__ v_cache
)
```

   - q、k_cache、v_cache：GPU全局内存中的query、key和value数据所在的位置
   - out：输出结果写到的位置
   - scalar_t：数据类型，如fp16
   - HEAD_SIZE：每个head中的元素数量
   - BLOCK_SIZE：每个block包含的tokens数量
   - NUM_THREADS：每个线程块中包含的线程数量
   - PARTITION_SIZE：执行张量并行的GPU的数量
2. 相关概念
   - Vec：同时被获取和计算的元素列表。对query和key而言，VEC_SIZE被设置使得每个thread group一次能获取和计算16bytes的数据；对value而言，V_VEC_SIZE被设置使得每个thread一次能获取并计算16bytes的数据
   - Thread group：包含THREAD_GROUP_SIZE个线程以计算一个query token和一个key token的结果
   - Warp：计算一个query token和一整个block的key token的结果
   - Thread Block：能够访问相同的shared memory，计算一个query和整个context的key token的结果
3. Query的存储与获取

   每个thread group通过以下方式获得一个head中一个token的数据，用于计算

```
const scalar_t* q_ptr = q + seq_idx * q_stride + head_idx * HEAD_SIZE;
```

   读取的数据在共享内存中以vec的形式存储，每个vec在共享内存中占据一行

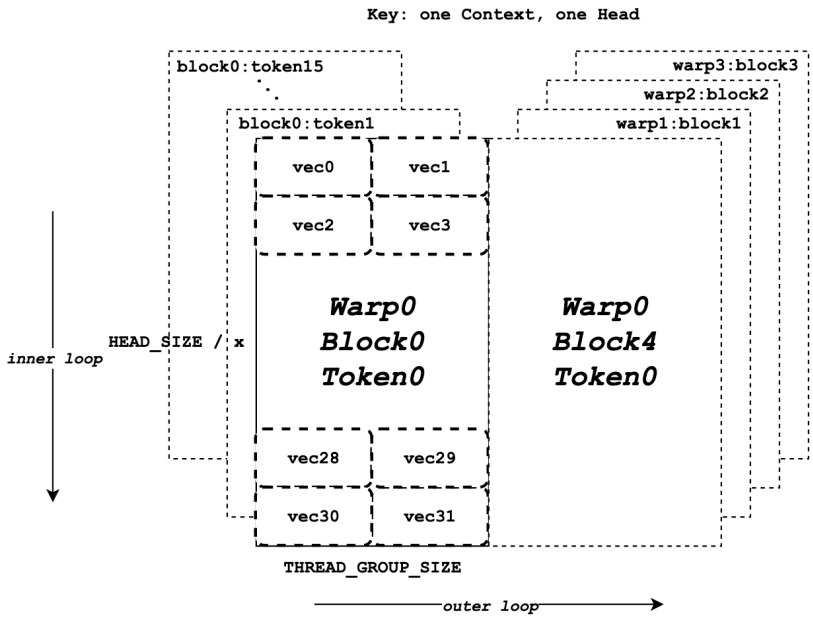
```
__shared__ Q_vec q_vecs[THREAD_GROUP_SIZE][NUM_VECS_PER_THREAD];
```

4. Keys的存储与分配

   与query不同的是，不同iteration中，thread group被分配到的key token是不同的，它根据对应的block索引、head索引和token索引来找到被分配的key token

```
const scalar_t* k_ptr = k_cache + physical_block_number * kv_block_stride + kv_head_idx * kv_head_stride +
physical_block_offset * x;
```

   当warp为4、block size为16、head size为128、x为8、thread group size为2时，其内存布局表示为



5. QK计算

整体的QK计算流程可以表示如下。程序首先完整读取整个新token的query向量，并保存在q_vecs向量中。接下来，在外循环中，程序遍历指向不同token的k_ptrs，它与query向量做内积，得到query与单个key token的计算结果(这里是计算了query与完整key token的结果，即dot操作中包含了一个reduction操作)。不同线程取出的k_vecs不同

```
q_vecs = ...
for ... {
   k_ptr = ...
   for ... {
      k_vecs[i] = ...
   }
   ...
   float qk = scale * Qk_dot<scalar_t, THREAD_GROUP_SIZE>::dot(q_vecs[thread_group_offset], k_vecs);
}
```

6. Softmax

Softmax计算的主要操作如下，其主要分为计算qk最大值m(x)和计算exp的和l(x)两个操作

$$m(x) := \max_i \quad x_i$$

$$f(x) := \begin{bmatrix} e^{x_1 - m(x)} & \dots & e^{x_B - m(x)} \end{bmatrix}$$

$$\ell(x) := \sum_i f(x)_i$$

$$\text{softmax}(x) := \frac{f(x)}{\ell(x)}$$

- qk_max和logits

  为了收集计算得到m(x)，需要按照thread group—warp—thread block的流程将每个thread group计算得到的qk值进行对比，并获得最大值。在thread group层面，由第一个线程执行如下操作，将其对应的key token与query token计算得到的qk值存储到logits相应位置中，并设置其qk_max值

  ```
  if (thread_group_offset == 0) {
     const bool mask = token_idx >= context_len;
     logits[token_idx - start_token_idx] = mask ? 0.f : qk;
     qk_max = mask ? qk_max : fmaxf(qk_max, qk);
  }
  ```

  在warp层面，为了降低通信开销，vLLM中采用了二分树形归约的方式，逐步前递qk_max，使0号thread group能够获得当前warp的qk_max最大值

```
for (int mask = WARP_SIZE / 2; mask >= THREAD_GROUP_SIZE; mask /= 2) {
    qk_max = fmaxf(qk_max, VLLM_SHFL_XOR_SYNC(qk_max, mask));
}
if (lane == 0) {
   red_smem[warp_idx] = qk_max;
}
```

最后在thread block层面，同样采用树形规约的方式，获得所有token下qk的最大值，并且可以通过广播的方式将其传递给所有线程

```
for (int mask = NUM_WARPS / 2; mask >= 1; mask /= 2) {
    qk_max = fmaxf(qk_max, VLLM_SHFL_XOR_SYNC(qk_max, mask));
}
qk_max = VLLM_SHFL_SYNC(qk_max, 0);
```

- exp_sum

  l(x)的计算同样需要聚合所有token的exp值。首先，每个thread group根据保存在logits中的qk值，计算exp(qk-qk_max)，并且聚合所有block的exp(与qk_max的归约方式类似)，获得exp_sum

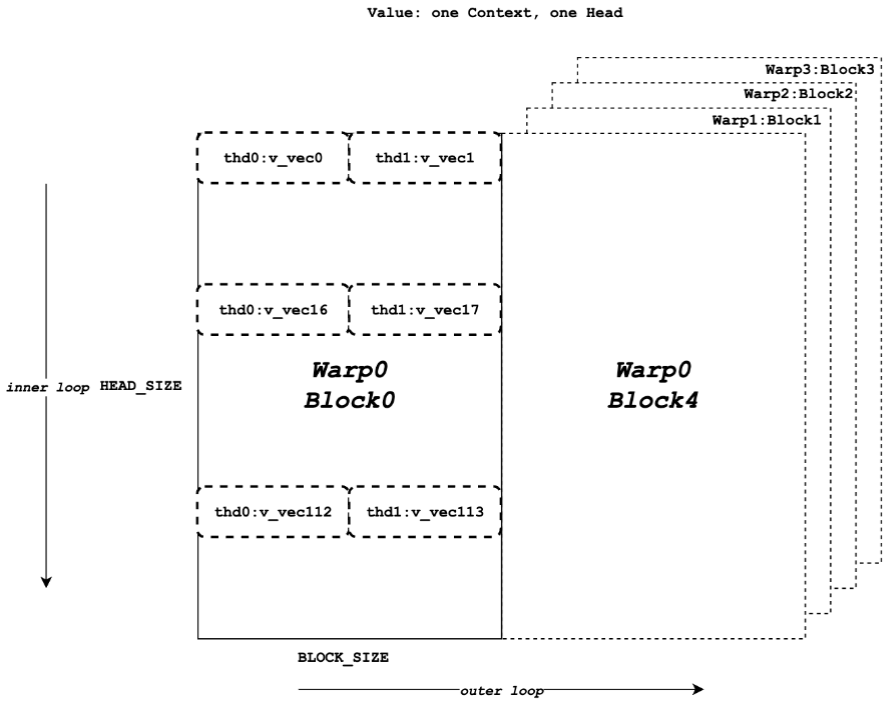
```
for (int i = thread_idx; i < num_tokens; i += NUM_THREADS) {
    float val = __expf(logits[i] - qk_max);
    logits[i] = val;
    exp_sum += val;
}
...
exp_sum = block_sum<NUM_WARPS>(&red_smem[NUM_WARPS], exp_sum);
```

接下来，通过令logits中的值除以exp_sum，得到各个qk的softmax值，此时各个token的softmax都保存在logits对应的位置中

```
const float inv_sum = __fdividef(1.f, exp_sum + 1e-6f);
for (int i = thread_idx; i < num_tokens; i += NUM_THREADS) {
   logits[i] *= inv_sum;
}
```

7. Value的存储与获取

与key token的内存布局不同，同一列的元素表示了一个token的value向量，使得对于一个value block而言，它包含了**head size行以及block size列**。这种布局的主要原因在于value和softmax结果的计算为加权平均，结果的每个元素都是权重与所有value对应位置的元素的加权和。这种布局使得不同线程仍旧可以按行进行元素获取，如下图所示



每个线程每次获取V_VEC_SIZE个元素与logits中对应的V_VEC_SIZE个元素进行点乘计算。计算由两个循环进行，内循环计算了单个块中value与logits的结果，而外循环计算了多个块的value和logits的结果(一个warp可能要计算多个block)，从而完成整个context的计算。下述运行的代码说明了一个warp的运行操作

```
float accs[NUM_ROWS_PER_THREAD];
for ... { // Compute dot product of the whole context.
    logits_vec = // V_VEC_SIZE elements from logits
    for ... { // Compute dot product of each block
        v_vec = // V_VEC_SIZE elements from value block
        ...
        accs[i] += dot(logits_vec, v_vec);
    }
}
```

8. LV计算

在每个thread得到其value的计算结果后，需要将结果执行reduction操作。首先，同个warp中不同线程的数据进行累加，其执行如下操作(树形归约)

```
for (int i = 0; i < NUM_ROWS_PER_THREAD; i++) {
   float acc = accs[i];
   for (int mask = NUM_V_VECS_PER_ROW / 2; mask >= 1; mask /= 2) {
      acc += VLLM_SHFL_XOR_SYNC(acc, mask);
   }
   accs[i] = acc;
}
```

接下来，vLLM将所有warp的结果进行聚合，使得每个thread包含了其被分配的head的完整聚合结果(所有warp的计算结果累加)。留意到虽然每个thread只包含了部分的结果(NUM_ROWS_PER_THREAD个)，但完整的输出已经被计算得到，只是存放在了不同的thread寄存器中

```
float* out_smem = reinterpret_cast<float*>(shared_mem);
for (int i = NUM_WARPS; i > 1; i /= 2) {
    // 大序号的warp将输出数据存放到共享内存中
    int mid = i / 2;
    float* dst = &out_smem[(warp_idx - mid) * HEAD_SIZE];
    for (int j = 0; j < NUM_ROWS_PER_THREAD; j++) {
                ...
        dst[row_idx] = accs[j];
    }
    // 小序号的warp将输出数据从共享内存中取出并累加到自身的accs数组上
    const float* src = &out_smem[warp_idx * HEAD_SIZE];
    for (int j = 0; j < NUM_ROWS_PER_THREAD; j++) {
                ...
        accs[j] += src[row_idx];
    }
}
```

9. 计算最终的Output

首先定义相应序列和head对应的结果的输出位置，用指针表示为

```
scalar_t* out_ptr = out + seq_idx * num_heads * max_num_partitions * HEAD_SIZE + head_idx * max_num_partitions *
HEAD_SIZE + partition_idx * HEAD_SIZE;
```

这里max_num_partition表示的是张量并行采用的GPU数量，使得每个token vector被划分成了max_num_partition份

最后，每个thread将自己的运行结果写到对应的内存位置中，代码如下

```
for (int i = 0; i < NUM_ROWS_PER_THREAD; i++) {
    const int row_idx = lane / NUM_V_VECS_PER_ROW + i * NUM_ROWS_PER_ITER;
    if (row_idx < HEAD_SIZE && lane % NUM_V_VECS_PER_ROW == 0) {
        from_float(*(out_ptr + row_idx), accs[i]);
    }
}
```