Background

For current large model inference, KV cache occupies a significant portion of GPU memory, so reducing the size of KV cache is an important direction for improvement. Recently, several papers have approached this issue from different angles, detailed comparison in the table, including:

1. FastDecode (https://arxiv.org/abs/2403.11421): This method offloads all computation of KV cache to the CPU. The computation and storage of KV cache occurs on CPU.
2. Compression methods based on quantization (GEAR, Mixed Precision): By applying various quantization techniques, the size of individual token KV caches is reduced without decreasing the number of tokens stored in the KV cache. This method may also result in corresponding residual and outlier matrices, which need to be stored in memory but not in the KV cache. It may also involve quantizing unimportant token KV caches to reduce the memory footprint of the KV cache.
3. Partial KV cache eviction (H2O, SnapKV, LESS, Adaptive Compression, Scissorhands, Dynamic Memory Compression, StreamingLLM): By removing some relatively useless KV cache entries, the memory footprint of the KV cache is reduced. Essentially, this reduces the number of tokens stored in the KV cache without reducing the size of individual token KV caches.

	Paper link	Eviction or not	Quantization or compression	Feature	vLLM integration
GEAR	https://arxiv.org/abs/2403.05527	No	Yes	Quantization on all tokens KV cache, consider outlier and residual.	Not widely applicable as it requires generating outlier and residual for each token generation, which is costly.
Mixed Precision	https://arxiv.org/pdf/2402.18096	No	Yes	Quantization to INT2 for unpivotal tokens KV cache and INT8 for pivotal tokens KV cache, consider outlier.	Hard to maintain different KV blocks for the block manager.
H2O	https://arxiv.org/abs/2306.14048	Yes	No	Select topK, probably 20% of KV cache, evict all other KV pairs.	Primarily aiming for generality and abstraction rather than being specific to one or two approaches.

					Details mentioned in the following parts.
SnapKV	https://arxiv.org/pdf/2404.14469	Yes	No	Select topK, evict all other KV pairs.	Same with the eviction method above.
LESS	https://arxiv.org/abs/2402.09398	Yes	No	Evict most KV cache, consider low rank matrix and keep constant size KV cache.	Same with the eviction method above.
Adaptive Compression	https://arxiv.org/pdf/2310.01801	Yes	No	Keep local and special tokens and general tokens, evict other token KV cache.	Same with the eviction method above.
Scissorhands	https://arxiv.org/pdf/2305.17118	Yes	Yes	Evict most unpivotal tokens, quantization to some KV cache.	Same with the eviction method above.
Dynamic Memory Compression	https://arxiv.org/pdf/2403.09636	No	Yes	Consider using appendix of accumulation on the last KV pair, which could reduce KV cache size.	Not widely applicable as it requires models to be trained using the same method.
StreamingLLM	https://arxiv.org/pdf/2309.17453	Yes	No	Keep the first tokens and the most recent tokens KV cache and evict all others.	Same with the eviction method above.

When addressing Sparse KV cache, we have previously considered supporting quantization (VLLM has already implemented this), implementing quantization + outlier + residual like GEAR (not widely applicable as it requires generating outlier and residual for each token generation, which is costly), and implementing KV cache accumulation + appendix (not widely applicable as it requires models to be trained using the same method). Finally, the idea is to **implement partial KV cache eviction**, primarily aiming for generality and abstraction rather than being specific to one or two approaches. Considering that six of the sparse KV cache methods we found are based on evicting cache entries, this method is also suitable for modification as part of a framework to be integrated into VLLM.

Sparse KV Cache Workflow

First, let's clarify the required parameters, including:

1. An optional flag "--sparse-kv-cache-type" indicating if we want to specify any sparse KV cache type. Default is ‘auto’ without using any sparse KV cache type, otherwise, there could be various methods, such as attention scores for H2O.
2. Compression ratio for evicting KV cache entries: 20% if we want to achieve 80% reduction of KV cache usage. We can calculate the value of 'n' for recreating KV cache every 'n' step based on the compression ratio.

The entire workflow includes:

1. During the first decoding pass, besides computing the KV values for all input tokens, we also need to calculate and retain information about the priority ranking of all token pairs, such as attention scores in H2O.
2. During each scheduling of VLLM, we need to check whether 'n' steps have been completed, indicating the necessity for KV cache compression. If necessary, based on the priority ranking of tokens, one or more new KV cache blocks will be allocated, modifying the position information of input positions. The block manager will then manage the transfer of corresponding KV blocks from the original sequence group to the latest KV block. Finally, the reference count of the original KV block will be decremented, and the corresponding original KV blocks may even be released.
3. The corresponding KV values are added to the KV cache until the next compression of the KV cache after 'n' steps, repeating this process until the entire process is completed.

Code Modification

Argument

1. Modify 'add_cli_args' in 'vllm/engine/arg_utils.py' to define all relevant arguments. Previously, both 'vllm/entrypoints/api_server.py' and 'vllm/entrypoints/openai/cli_args.py' had relevant entry points calling 'add_cli_args'.

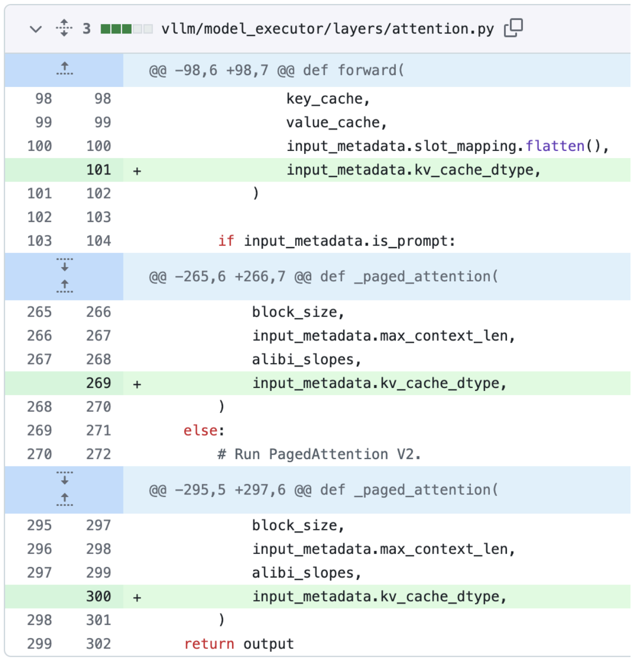
```
    @staticmethod
    def add_cli_args(
            parser: argparse.ArgumentParser) -> argparse.ArgumentParser:
        """Shared CLI arguments for vLLM engine."""

        # NOTE: If you update any of the arguments below, please also
        # make sure to update docs/source/models/engine_args.rst

        # Model arguments
        ...
        parser.add_argument(
            '--kv-cache-dtype',
            type=str,
            choices=['auto', 'fp8'],
            default=EngineArgs.kv_cache_dtype,
            help='Data type for kv cache storage. If "auto", will use model '
            'data type. FP8_E5M2 (without scaling) is only supported on cuda '
            'version greater than 11.8. On ROCm (AMD GPU), FP8_E4M3 is instead '
            'supported for common inference criteria. ')
        parser.add_argument(
            '--sparse-kv-cache-type',
            type=str,
            choices=['auto', 'h2o'],
            default=EngineArgs.sparse_kv_cache_type,
            help='Sparse kv cache storage type. If "auto", will not use any sparse
kv cache.'
        )
        parser.add_argument('--max-model-len',
                            type=int,
                            default=EngineArgs.max_model_len,
                            help='model context length. If unspecified, '
                            'will be automatically derived from the model.')
        ...
```

2. Modify vllm/config.py, need to verify sparse KV cache arguments.

```python
class CacheConfig:
    """Configuration for the KV cache.

    Args:
        block_size: Size of a cache block in number of tokens.
        gpu_memory_utilization: Fraction of GPU memory to use for the
            vLLM execution.
        swap_space: Size of the CPU swap space per GPU (in GiB).
        cache_dtype: Data type for kv cache storage.
        num_gpu_blocks_override: Number of GPU blocks to use. This overrides the
            profiled num_gpu_blocks if specified. Does nothing if None.
    """

    def __init__(
        self,
        block_size: int,
        gpu_memory_utilization: float,
        swap_space: int,
        cache_dtype: str,
        sparse_cache_type: str,
        num_gpu_blocks_override: Optional[int] = None,
        sliding_window: Optional[int] = None,
        enable_prefix_caching: bool = False,
    ) -> None:
        self.block_size = block_size
        self.gpu_memory_utilization = gpu_memory_utilization
        self.swap_space_bytes = swap_space * _GB
        self.num_gpu_blocks_override = num_gpu_blocks_override
        self.cache_dtype = cache_dtype
        self.sliding_window = sliding_window
        self.enable_prefix_caching = enable_prefix_caching
        self._verify_args()
        self._verify_cache_dtype()
        self._verify_cache_sparse_type()

        # Will be set after profiling.
        self.num_gpu_blocks = None
        self.num_cpu_blocks = None

    def _verify_cache_sparse_type(self) -> None:
        if self.sparse_cache_type == "auto":
            pass
        elif self.sparse_cache_type == "h2o":
            # Ensure supporting both CUDA and ROCM.
            ...
        else:
            raise ValueError(f"Unknown sparse kv cache type: 
{self.sparse_cache_type}")
```

3. For the correspondingvllm/engine/llm_engine.py, vllm/model_executor/layers/attention.py, vllm/model_executor/input_metadata.py, vllm/worker/worker.py, vllm/worker/model_runner.py, vllm/attention/ops/paged_attn.py and vllm/attention/backends/xformers.py related backends code, they all should have

similar arguments update like belows.



```
3 vllm/model_executor/layers/attention.py

@@ -98,6 +98,7 @@ def forward(
                 key_cache,
                 value_cache,
                 input_metadata.slot_mapping.flatten(),
+                input_metadata.kv_cache_dtype,
             )

         if input_metadata.is_prompt:
@@ -265,6 +266,7 @@ def _paged_attention(
             block_size,
             input_metadata.max_context_len,
             alibi_slopes,
+            input_metadata.kv_cache_dtype,
         )
     else:
         # Run PagedAttention V2.
@@ -295,5 +297,6 @@ def _paged_attention(
             block_size,
             input_metadata.max_context_len,
             alibi_slopes,
+            input_metadata.kv_cache_dtype,
         )
     return output
```

Scheduling and Memory management

During each scheduling of VLLM, we need to check whether 'n' steps have been completed, indicating the need for KV cache compression. If necessary, based on the priority ranking of tokens, one or more new KV cache blocks will be written, modifying the position information of input positions. The block manager will then manage the transfer of corresponding KV blocks from the original sequence group to the latest KV block. Finally, the reference count of the original KV block will be decremented, and the corresponding original KV blocks may even be released.

1. Update _schedule_running and _append_slots in vllm/core/scheduler.py.

```
def _schedule_running(
    self,
    running_queue: deque,
    budget: SchedulingBudget,
    curr_loras: Optional[Set[int]],
    policy: Policy,
    enable_chunking: bool = False,
 ) -> Tuple[deque, SchedulerRunningOutputs]:
    """Schedule sequence groups that are running.

    Running queue should include decode and chunked prefill requests.

    Args:
        running_queue: The queue that contains running requests (i.e.,
            decodes). The given arguments are NOT in-place modified.
        budget: The scheduling budget. The argument is in-place updated
            when any decodes are preempted.
        curr_loras: Currently batched lora request ids. The argument is
            in-place updated when any decodes are preempted.
        policy: The sorting policy to sort running_queue.
        enable_chunking: If True, seq group can be chunked and only a
            chunked number of tokens are scheduled  if
            `budget.num_batched_tokens` has not enough capacity to schedule
            all tokens.

    Returns:
        A tuple of remaining running queue (should be always 0) after
        scheduling and SchedulerRunningOutputs.
    """
    # Blocks that need to be swapped or copied before model execution.
    blocks_to_swap_out: Dict[int, int] = {}
    blocks_to_copy: Dict[int, List[int]] = {}

    decode_seq_groups: List[ScheduledSequenceGroup] = []
    prefill_seq_groups: List[ScheduledSequenceGroup] = []
    preempted: List[SequenceGroup] = []
    swapped_out: List[SequenceGroup] = []

    # NOTE(woosuk): Preemption happens only when there is no available slot
    # to keep all the sequence groups in the RUNNING state.
    # In this case, the policy is responsible for deciding which sequence
    # groups to preempt.
    now = time.time()
    running_queue = policy.sort_by_priority(now, running_queue)

    while running_queue:
        seq_group = running_queue[0]
        num_running_tokens = self._get_num_new_tokens(
            seq_group, SequenceStatus.RUNNING, enable_chunking, budget)

        # We can have up to 1 running prefill at any given time in running
        # queue, which means we can guarantee chunk size is at least 1.
        assert num_running_tokens != 0
        num_running_seqs = seq_group.get_max_num_running_seqs()

        running_queue.popleft()
        # Add a condition check in while loop based on the newly created KV cache to see if
    there is enough slots or not.
        # if use sparse-kv-cache flag and there is i%n==0 step.
        # We could consider reserve some KV blocks for contain these blocks.
        while not self._can_append_slots(seq_group):
            budget.subtract_num_batched_tokens(seq_group.request_id,
                                    num_running_tokens)
                budget.subtract_num_seqs(seq_group.request_id,
                                    num_running_seqs)
                if curr_loras is not None and seq_group.lora_int_id > 0:
                    curr_loras.remove(seq_group.lora_int_id)
                if running_queue:
                    # Preempt the lowest-priority sequence groups.
                    victim_seq_group = running_queue.pop()
                    preempted_mode = self._preempt(victim_seq_group,
                                                   blocks_to_swap_out)
                    if preempted_mode == PreemptionMode.RECOMPUTE:
                        preempted.append(victim_seq_group)
                    else:
                        swapped_out.append(victim_seq_group)
                else:
                    # No other sequence groups can be preempted.
                    # Preempt the current sequence group.
                    preempted_mode = self._preempt(seq_group,
                                                   blocks_to_swap_out)
                    if preempted_mode == PreemptionMode.RECOMPUTE:
                        preempted.append(seq_group)
                    else:
                        swapped_out.append(seq_group)
                    break
            else:
                logger.debug(f"append slot for {seq_group}")
                print(f"append slot for {seq_group}")
                print(f"num_free_gpu get_num_free_gpu_blocks
{self.block_manager.get_num_free_gpu_blocks()}")
                self._append_slots(seq_group, blocks_to_copy)
                print(f"num_free_gpu get_num_free_gpu_blocks
{self.block_manager.get_num_free_gpu_blocks()}")
                is_prefill = seq_group.is_prefill()
                if is_prefill:
                    prefill_seq_groups.append(
                        ScheduledSequenceGroup(
                            seq_group=seq_group,
                            token_chunk_size=num_running_tokens))
                else:
                    decode_seq_groups.append(
                        ScheduledSequenceGroup(seq_group=seq_group,
                                               token_chunk_size=1))
                budget.add_num_batched_tokens(seq_group.request_id,
                                              num_running_tokens)
                budget.add_num_seqs(seq_group.request_id, num_running_seqs)
                if curr_loras is not None and seq_group.lora_int_id > 0:
                    curr_loras.add(seq_group.lora_int_id)
```

```
        # Make sure all queues are updated.
        assert len(running_queue) == 0
        return running_queue, SchedulerRunningOutputs(
            decode_seq_groups=decode_seq_groups,
            prefill_seq_groups=prefill_seq_groups,
            preempted=preempted,
            swapped_out=swapped_out,
            blocks_to_swap_out=blocks_to_swap_out,
            blocks_to_copy=blocks_to_copy,
            num_lookahead_slots=self._get_num_lookahead_slots(
                is_prefill=False))

    def _append_slots(
        self,
        seq_group: SequenceGroup,
        blocks_to_copy: Dict[int, List[int]],
    ) -> None:
        """Appends new slots to the sequences in the given sequence group.

        Args:
            seq_group (SequenceGroup): The sequence group containing the
                sequences to append slots to.
            blocks_to_copy (Dict[int, List[int]]): A dictionary mapping source
                block indices to lists of destination block indices. This
                dictionary is updated with the new source and destination block
                indices for the appended slots.
        """
        num_lookahead_slots = self._get_num_lookahead_slots(is_prefill=False)

        for seq in seq_group.get_seqs(status=SequenceStatus.RUNNING):
            # If use sparse-kv-cache flag and there is i%n==0 step.
            # Get the original KV blocks
            # Add a new block manager method "create_new_slots" similar to append_slots but
to create new KV cache slots, rather than append_slots.

            cows = self.block_manager.append_slots(seq, num_lookahead_slots)

            for src, dests in cows.items():
                if src not in blocks_to_copy:
                    blocks_to_copy[src] = []
                # This is important.
                blocks_to_copy[src].extend(dests)

            # copy the KV blocks from the original KV cache to the new KV cache with the
attention score or other tokens priority.

                        # Add a new ops method to copy all the KV cache from original to the
                  new one.
```

2. Update vllm/core/block_manager_v1.py and vllm/core/block_manager_v2.py by adding a new method "create_new_slots".

```
    def append_slots(
        self,
        seq: Sequence,
        num_lookahead_slots: int = 0,
    ) -> Dict[int, List[int]]:
        """Allocate a physical slot for a new token."""
        logical_blocks = seq.logical_token_blocks
        block_table = self.block_tables[seq.seq_id]
        # If we need to allocate a new physical block
        if len(block_table) < len(logical_blocks):
            # Currently this code only supports adding one physical block
            assert len(block_table) == len(logical_blocks) - 1

            if (self.block_sliding_window
                    and len(block_table) >= self.block_sliding_window):
                # reuse a block
                block_table.append(block_table[len(block_table) %
                                               self.block_sliding_window])
            else:
                # The sequence has a new logical block.
                # Allocate a new physical block.
                new_block = self._allocate_last_physical_block(seq)
                block_table.append(new_block)
                return {}

        # We want to append the token to the last physical block.
        last_block = block_table[-1]
        assert last_block.device == Device.GPU
        if last_block.ref_count == 1:
            # Not shared with other sequences. Appendable.
            if self.enable_caching:
                # If the last block is now complete, we may reuse an old block
                # to save memory.
                maybe_new_block = self._maybe_promote_last_block(
                    seq, last_block)
                block_table[-1] = maybe_new_block
            return {}
        else:
            # The last block is shared with other sequences.
            # Copy on Write: Allocate a new block and copy the tokens.
            new_block = self._allocate_last_physical_block(seq)

            block_table[-1] = new_block
            self.gpu_allocator.free(last_block)
            return {last_block.block_number: [new_block.block_number]}


    # Add a new block manager method "create_new_slots" similar to append_slots
but to create new KV cache slots, rather than append_slots.
    def create_new_slots(self, seq: Sequence) -> Dict[int, List[int]]:
        block_table = self.block_tables[seq.seq_id]
        # go through the block_table to see if block.ref_count == 1
        # if 1, self.gpu_allocator.free(block) to free the block
        # Update the block table for this new KV cache slots.
        pass
```

3. Add a new copy_to_paged_cache method in vllm/attention/ops/paged_attn.py.

```
    @staticmethod
    def write_to_paged_cache(
        key: torch.Tensor,
        value: torch.Tensor,
        key_cache: torch.Tensor,
        value_cache: torch.Tensor,
        slot_mapping: torch.Tensor,
        kv_cache_dtype: str,
        kv_scale: float,
    ) -> None:
        ops.reshape_and_cache(
            key,
            value,
            key_cache,
            value_cache,
            slot_mapping.flatten(),
            kv_cache_dtype,
            kv_scale,
        )

    @staticmethod
    def copy_to_paged_cache(
        src_key_cache: torch.Tensor,
        src_value_cache: torch.Tensor,
        tgt_key_cache: torch.Tensor,
        tgt_value_cache: torch.Tensor,
        slot_mapping: torch.Tensor,
        atten_score: torch.Tensor,
    ) -> None:
        ops.copy_to_cache(
            src_key_cache,
            src_value_cache,
            tgt_key_cache,
            tgt_value_cache,
            slot_mapping.flatten(),
            atten_score
        )
```

Kernel

1. Add a new copy_to_cache_kernel to csrc/cache_kernels.cu, similar to reshape_and_cache_kernel.

```
template<typename scalar_t, typename cache_t>
__global__ void copy_to_cache_kernel(
  cache_t* __restrict__ src_key_cache,           // [num_blocks, num_heads, head_size/x, block_size, x]
  cache_t* __restrict__ src_value_cache,           // [num_blocks, num_heads, head_size, block_size]
  cache_t* __restrict__ tgt_key_cache,           // [num_blocks, num_heads, head_size/x, block_size, x]
  cache_t* __restrict__ tgt_value_cache,           // [num_blocks, num_heads, head_size, block_size]
  const int64_t* __restrict__ slot_mapping,   // [num_tokens]
  const int key_stride,
  const int value_stride,
  const int num_heads,
  const int head_size,
  const int block_size,
  const int x,
  const float kv_scale) {

}

template<typename scalar_t, typename cache_t, bool is_fp8_kv_cache>
__global__ void reshape_and_cache_kernel(
  const scalar_t* __restrict__ key,           // [num_tokens, num_heads, head_size]
  const scalar_t* __restrict__ value,         // [num_tokens, num_heads, head_size]
  cache_t* __restrict__ key_cache,            // [num_blocks, num_heads, head_size/x,
block_size, x]
  cache_t* __restrict__ value_cache,          // [num_blocks, num_heads, head_size,
block_size]
  const int64_t* __restrict__ slot_mapping,   // [num_tokens]
  const int key_stride,
  const int value_stride,
  const int num_heads,
  const int head_size,
  const int block_size,
  const int x,
  const float kv_scale) {
  const int64_t token_idx = blockIdx.x;
  const int64_t slot_idx = slot_mapping[token_idx];
  if (slot_idx < 0) {
    // Padding token that should be ignored.
    return;
  }

  const int64_t block_idx = slot_idx / block_size;
  const int64_t block_offset = slot_idx % block_size;

  const int n = num_heads * head_size;

  for (int i = threadIdx.x; i < n; i += blockDim.x) {
    const int64_t src_key_idx = token_idx * key_stride + i;
    const int64_t src_value_idx = token_idx * value_stride + i;
    const int head_idx = i / head_size;
    const int head_offset = i % head_size;
    const int x_idx = head_offset / x;
    const int x_offset = head_offset % x;

    const int64_t tgt_key_idx = block_idx * num_heads * (head_size / x) * block_size * x
```

```
                            + head_idx * (head_size / x) * block_size * x
                            + x_idx * block_size * x
                            + block_offset * x
                            + x_offset;
    const int64_t tgt_value_idx = block_idx * num_heads * head_size * block_size
                              + head_idx * head_size * block_size
                              + head_offset * block_size
                              + block_offset;
    scalar_t tgt_key = key[src_key_idx];
    scalar_t tgt_value = value[src_value_idx];

    if constexpr (is_fp8_kv_cache) {
#if defined(ENABLE_FP8_E5M2)
      key_cache[tgt_key_idx] = fp8_e5m2_unscaled::vec_conversion<uint8_t,
scalar_t>(tgt_key);
      value_cache[tgt_value_idx] = fp8_e5m2_unscaled::vec_conversion<uint8_t,
scalar_t>(tgt_value);
#elif defined(ENABLE_FP8_E4M3)
      key_cache[tgt_key_idx] = fp8_e4m3::scaled_vec_conversion<uint8_t, scalar_t>(tgt_key,
kv_scale);
      value_cache[tgt_value_idx] = fp8_e4m3::scaled_vec_conversion<uint8_t,
scalar_t>(tgt_value, kv_scale);
#else
      assert(false);
#endif
    } else {
      key_cache[tgt_key_idx] = tgt_key;
      value_cache[tgt_value_idx] = tgt_value;
    }
    // Design update: If compression, compress the KV cache result from the normal one to
the compressed ones.
    // If eviction, nothing.
    k += 1;
  }
  printf("Cache kernel result done\n");
```

1. }
2. Update paged_attention_kernel in csrc/attention/attention_kernels.cu.

```
// TODO(woosuk): Merge the last two dimensions of the grid.
// Grid: (num_heads, num_seqs, max_num_partitions).
template<
  typename scalar_t,
  typename cache_t,
  int HEAD_SIZE,
  int BLOCK_SIZE,
  int NUM_THREADS,
  bool IS_FP8_KV_CACHE,
  int PARTITION_SIZE = 0> // Zero means no partitioning.
__device__ void paged_attention_kernel(
  float* __restrict__ exp_sums,           // [num_seqs, num_heads, max_num_partitions]
  float* __restrict__ max_logits,         // [num_seqs, num_heads, max_num_partitions]
  scalar_t* __restrict__ out,             // [num_seqs, num_heads, max_num_partitions,
head_size]
  const scalar_t* __restrict__ q,         // [num_seqs, num_heads, head_size]
  const cache_t* __restrict__ k_cache,    // [num_blocks, num_kv_heads, head_size/x,
block_size, x]
  const cache_t* __restrict__ v_cache,    // [num_blocks, num_kv_heads, head_size,
block_size]
  const int num_kv_heads,                 // [num_heads]
  const float scale,
  const int* __restrict__ block_tables,   // [num_seqs, max_num_blocks_per_seq]
  const int* __restrict__ context_lens,   // [num_seqs]
  const int max_num_blocks_per_seq,
  const float* __restrict__ alibi_slopes, // [num_heads]
  const int q_stride,
  const int kv_block_stride,
  const int kv_head_stride,
  const float kv_scale) {
  const int seq_idx = blockIdx.y;
  const int partition_idx = blockIdx.z;
  const int max_num_partitions = gridDim.z;
  constexpr bool USE_PARTITIONING = PARTITION_SIZE > 0;
  const int context_len = context_lens[seq_idx];

  ....
  // This block part is to calculate q*k from line 202 to line 263.
  for (int block_idx = start_block_idx + warp_idx; block_idx < end_block_idx; block_idx +=
NUM_WARPS) {
    // NOTE(woosuk): The block number is stored in int32. However, we cast it to int64
    // because int32 can lead to overflow when this variable is multiplied by large numbers
    // (e.g., kv_block_stride).
    const int64_t physical_block_number = static_cast<int64_t>(block_table[block_idx]);

    // Load a key to registers.
    // Each thread in a thread group has a different part of the key.
    // For example, if the the thread group size is 4, then the first thread in the group
    // has 0, 4, 8, ... th vectors of the key, and the second thread has 1, 5, 9, ... th
    // vectors of the key, and so on.
    for (int i = 0; i < NUM_TOKENS_PER_THREAD_GROUP; i++) {
      const int physical_block_offset = (thread_group_idx + i * WARP_SIZE) % BLOCK_SIZE;
      const int token_idx = block_idx * BLOCK_SIZE + physical_block_offset;
      K_vec k_vecs[NUM_VECS_PER_THREAD];

#pragma unroll
      for (int j = 0; j < NUM_VECS_PER_THREAD; j++) {
        const cache_t* k_ptr = k_cache + physical_block_number * kv_block_stride
                                 + kv_head_idx * kv_head_stride
                                 + physical_block_offset * x;
        const int vec_idx = thread_group_offset + j * THREAD_GROUP_SIZE;
        const int offset1 = (vec_idx * VEC_SIZE) / x;
        const int offset2 = (vec_idx * VEC_SIZE) % x;
        if constexpr (IS_FP8_KV_CACHE) {
#if defined(ENABLE_FP8_E5M2)
          Quant_vec k_vec_quant = *reinterpret_cast<const Quant_vec*>(k_ptr + offset1 *
BLOCK_SIZE * x + offset2);
          // Vector conversion from Quant_vec to K_vec.
          k_vecs[j] = fp8_e5m2_unscaled::vec_conversion<K_vec, Quant_vec>(k_vec_quant);
#elif defined(ENABLE_FP8_E4M3)
          Quant_vec k_vec_quant = *reinterpret_cast<const Quant_vec*>(k_ptr + offset1 *
BLOCK_SIZE * x + offset2);
          // Vector conversion from Quant_vec to K_vec. Use scaled_vec_conversion to convert
FP8_E4M3 quantized k
          // cache vec to k vec in higher precision (FP16, BFloat16, etc.)
          k_vecs[j] = fp8_e4m3::scaled_vec_conversion<K_vec, Quant_vec>(k_vec_quant,
kv_scale);
#else
          assert(false);
#endif
        } else {
          k_vecs[j] = *reinterpret_cast<const K_vec*>(k_ptr + offset1 * BLOCK_SIZE * x +
offset2);
        }
        // If eviction, calculate and update the latest token priority sequences and token
positions, such as attention score for each token in each layer, which could be the q*k for
H2O.
      }

      // Compute dot product.
      // This includes a reduction across the threads in the same thread group.
      float qk = scale * Qk_dot<scalar_t,
THREAD_GROUP_SIZE>::dot(q_vecs[thread_group_offset], k_vecs);
      // Add the ALiBi bias if slopes are given.
      qk += (alibi_slope != 0) ? alibi_slope * (token_idx - context_len + 1) : 0;

      if (thread_group_offset == 0) {
        // Store the partial reductions to shared memory.
        // NOTE(woosuk): It is required to zero out the masked logits.
        const bool mask = token_idx >= context_len;
        logits[token_idx - start_token_idx] = mask ? 0.f : qk;
        // Update the max value.
        qk_max = mask ? qk_max : fmaxf(qk_max, qk);
      }
    }
```

```
  }

  ...
  // This block part is to calculate q*k*v from line 338 to line 387.
  for (int block_idx = start_block_idx + warp_idx; block_idx < end_block_idx; block_idx +=
NUM_WARPS) {
    // NOTE(woosuk): The block number is stored in int32. However, we cast it to int64
    // because int32 can lead to overflow when this variable is multiplied by large numbers
    // (e.g., kv_block_stride).
    const int64_t physical_block_number = static_cast<int64_t>(block_table[block_idx]);
    const int physical_block_offset = (lane % NUM_V_VECS_PER_ROW) * V_VEC_SIZE;
    const int token_idx = block_idx * BLOCK_SIZE + physical_block_offset;
    L_vec logits_vec;
    from_float(logits_vec, *reinterpret_cast<Float_L_vec*>(logits + token_idx -
start_token_idx));

    const cache_t* v_ptr = v_cache + physical_block_number * kv_block_stride
                               + kv_head_idx * kv_head_stride;
#pragma unroll
    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) {
        const int offset = row_idx * BLOCK_SIZE + physical_block_offset;
        V_vec v_vec;
        if constexpr (IS_FP8_KV_CACHE) {
#if defined(ENABLE_FP8_E5M2)
          V_quant_vec v_quant_vec = *reinterpret_cast<const V_quant_vec*>(v_ptr + offset);
          // Vector conversion from V_quant_vec to V_vec.
          v_vec = fp8_e5m2_unscaled::vec_conversion<V_vec, V_quant_vec>(v_quant_vec);
#elif defined(ENABLE_FP8_E4M3)
          V_quant_vec v_quant_vec = *reinterpret_cast<const V_quant_vec*>(v_ptr + offset);
          // Vector conversion from V_quant_vec to V_vec. Use scaled_vec_conversion to
convert
          // FP8_E4M3 quantized v cache vec to v vec in higher precision (FP16, BFloat16, etc.)
          v_vec = fp8_e4m3::scaled_vec_conversion<V_vec, V_quant_vec>(v_quant_vec,
kv_scale);
#else
          assert(false);
#endif
        } else {
          v_vec = *reinterpret_cast<const V_vec*>(v_ptr + offset);
        }
        // If eviction, calculate and update the latest token priority sequences and token
positions, such as attention score for each token in each layer, which could be the q*k for
H2O.
        if (block_idx == num_context_blocks - 1) {
          // NOTE(woosuk): When v_vec contains the tokens that are out of the context,
          // we should explicitly zero out the values since they may contain NaNs.
          // See https://github.com/vllm-project/vllm/issues/641#issuecomment-1682544472
          scalar_t* v_vec_ptr = reinterpret_cast<scalar_t*>(&v_vec);
#pragma unroll
          for (int j = 0; j < V_VEC_SIZE; j++) {
            v_vec_ptr[j] = token_idx + j < context_len ? v_vec_ptr[j] : zero_value;
          }
        }
        accs[i] += dot(logits_vec, v_vec);
      }
    }
}
...
```

3. Add a new csrc/sparse_cache_utils.cuh file, which includes different eviction methods.

```
// sparse_cache_utils.h

#ifndef SPARSE_CACHE_UTILS_H_
#define SPARSE_CACHE_UTILS_H_

#include <cstdint>

enum class EvictMode {
    NONE,
    H2O,
    ...
};

typedef void (*EvictFunction)(const uint8_t* input, size_t input_size, EvictMode
evict_mode);

inline void H2OEvictFunction(const uint8_t* input, size_t input_size, EvictMode
evict_mode) {
    // Evict logic based on attention score for H2O.
}

#endif // SPARSE_CACHE_UTILS_H_
```

Others

We also need to update example, benchmark and test related code to ensure the corresponding changes to run as expected.