vLLM -- KV cache

Chenye Wang

Nov 5, 2024

[1] https://docs.vllm.ai/en/stable/dev/kernel/paged_attention.html

Self-attention

- input hidden state sequence

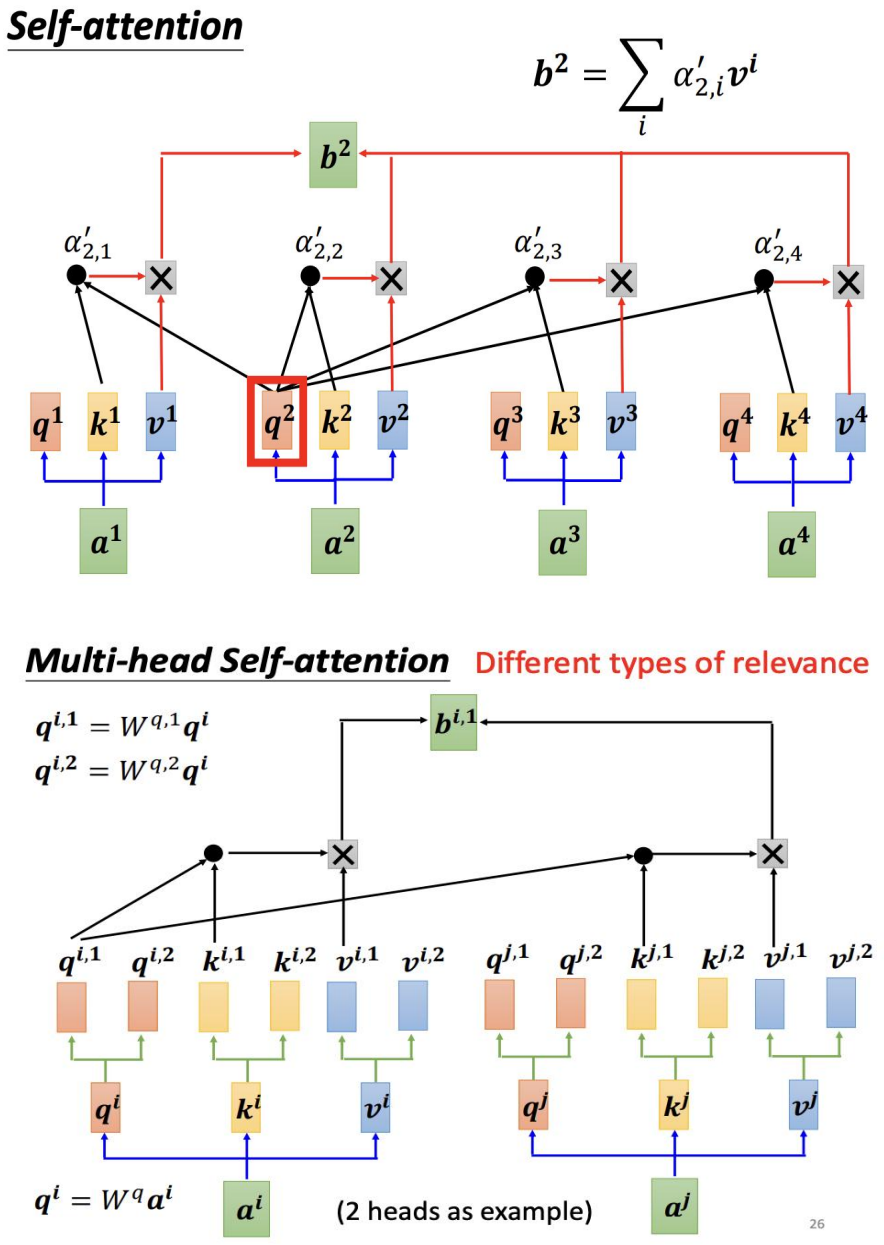
$$(x_1, \ldots, x_n) \in \mathbb{R}^{n \times d},$$

- query, key, and value vector

$$q_i = W_q x_i,\ k_i = W_k x_i,\ v_i = W_v x_i.$$

- compute attention score

$$a_{ij} = \frac{\exp(q_i^\top k_j / \sqrt{d})}{\sum_{t=1}^{i} \exp(q_i^\top k_t / \sqrt{d})},\ o_i = \sum_{j=1}^{i} a_{ij} v_j.$$



[1] Attention Is All You Need. *Advances in neural information processing systems* 30 (2017).

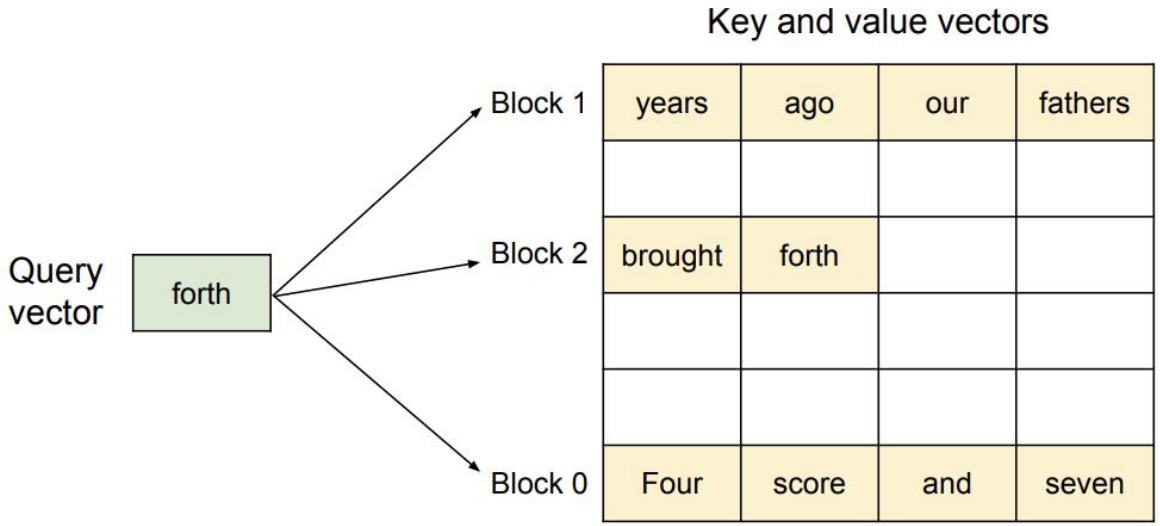
PagedAttention

- partition KV cache of each sequence into KV blocks
- KV block size(B) -- number of tokens
  - note: In vLLM, key and value vectors at different heads and layers each have a separate block

$$\text{key block } K_j = (k_{(j-1)B+1}, \ldots, k_{jB})$$

$$\text{value block } V_j = (v_{(j-1)B+1}, \ldots, v_{jB})$$

$$A_{ij} = \frac{\exp(q_i^\top K_j / \sqrt{d})}{\sum_{t=1}^{\lceil i/B \rceil} \exp(q_i^\top K_t \mathbf{1} / \sqrt{d})}, \quad o_i = \sum_{j=1}^{\lceil i/B \rceil} V_j A_{ij}^\top,$$



vLLM multi-head query attention kernel

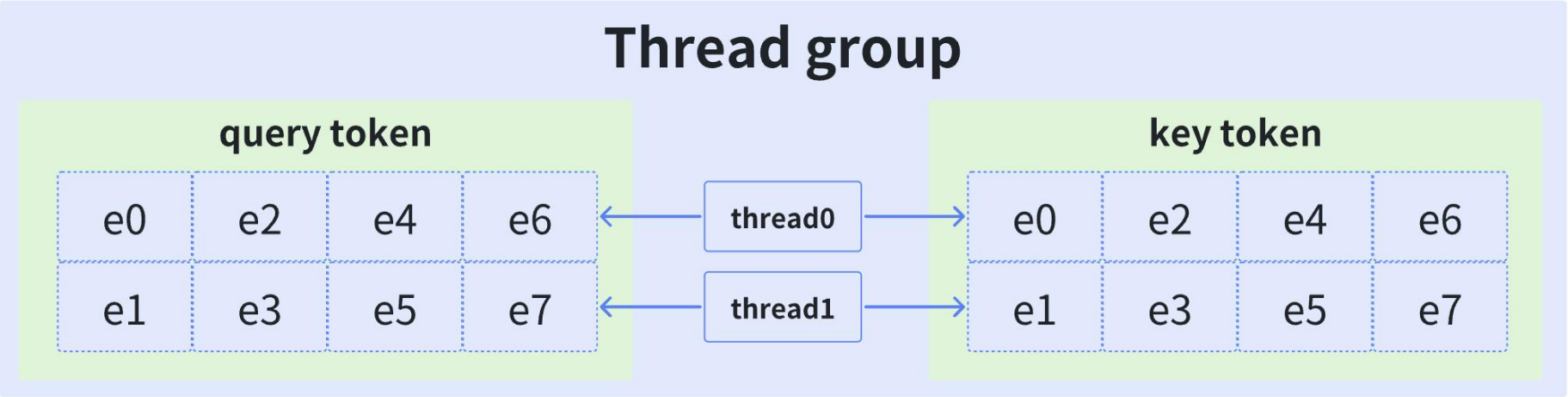
- csrc/attention/attention_kernels.cu

```
template<
typename scalar_t,          // the data type of the query, key, and value data elements, such as FP16
int HEAD_SIZE,              // the number of elements in each head
int BLOCK_SIZE,             // the number of tokens in each block
int NUM_THREADS,            // the number of threads in each thread block
int PARTITION_SIZE = 0>     // the number of tensor parallel GPUs
__device__ void paged_attention_kernel(
... // Other side args.
const 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 scalar_t* __restrict__ k_cache,   // [num_blocks, num_kv_heads, head_size/x, block_size, x]
const scalar_t* __restrict__ v_cache,   // [num_blocks, num_kv_heads, head_size, block_size]
... // Other side args.
)
```

multi-dimensional arrays

Thread group

- a small group of threads(THREAD_GROUP_SIZE)
  - fetches and calculates **one query token** and **one key token** at a time.
- Each thread handles only **a portion of** the token data.
- **x**: the total number of elements processed by one thread group



Vec

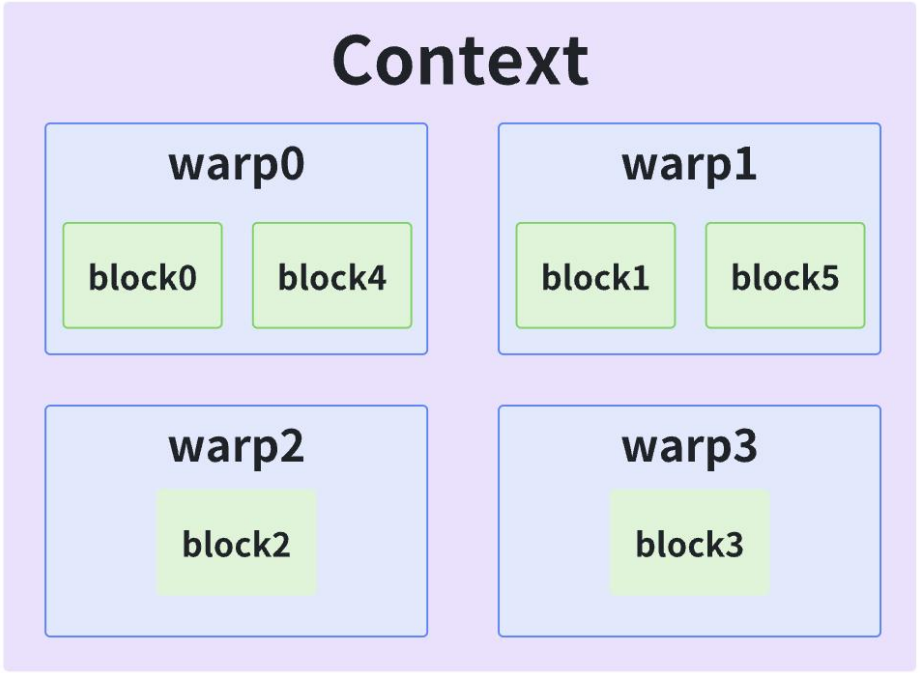
- a list of **elements** that are fetched and calculated together
- each thread group can fetch and calculate **16 bytes** of data at a time

- if scalar_t is FP16 (2 bytes) and THREAD_GROUP_SIZE = 2
  - query and key data
    - VEC_SIZE = 4
  - value data
    - V_VEC_SIZE = 8

Warp

- a group of 32 threads(WARP_SIZE)
- processes the calculation between
  - **one query token** and
  - **key tokens of one entire block** at a time
- processes multiple blocks

  in multiple iterations

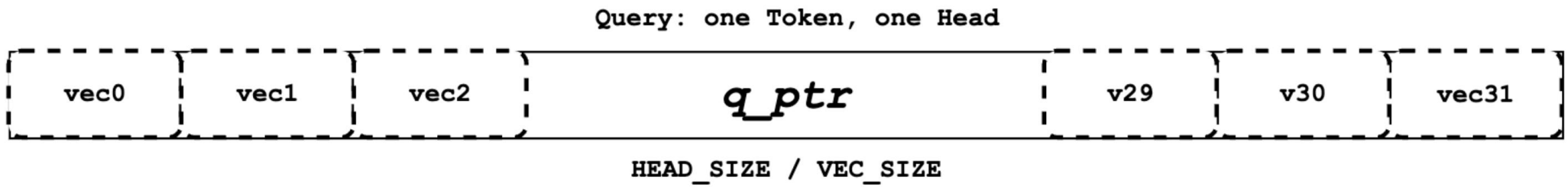


Query

- Each thread defines its own **q_ptr**
  - points to the assigned query token data on global memory.

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

- VEC_SIZE = 4, HEAD_SIZE = 128

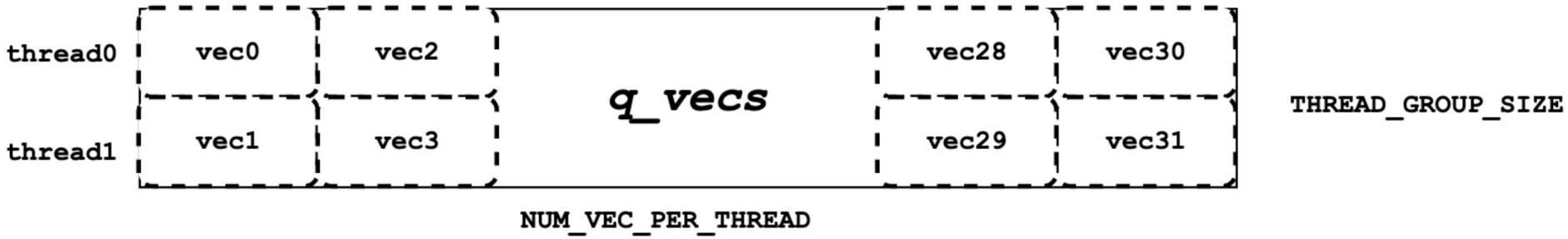


Query

- Read the global memory data pointed to by **q_ptr**
  into *shared memory* as **q_vecs**

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

- THREAD_GROUP_SIZE = 2
  - neighboring threads can read neighbor memory
    - memory coalescing



Key

- **k_ptr** in each thread will point to different key token at different iterations.

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

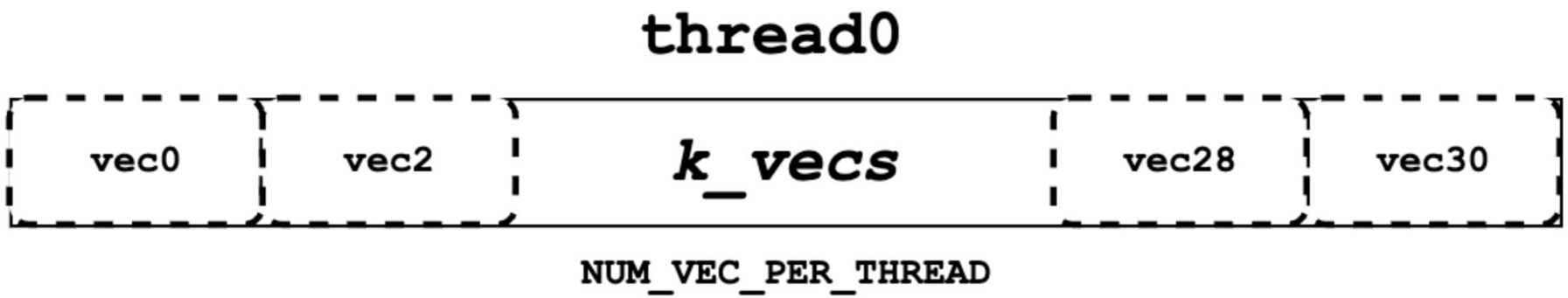
- **k_ptr** points to key token data based on k_cache at assigned block, assigned head and assigned token.

Key

- Read the key token data from **k_ptr** and

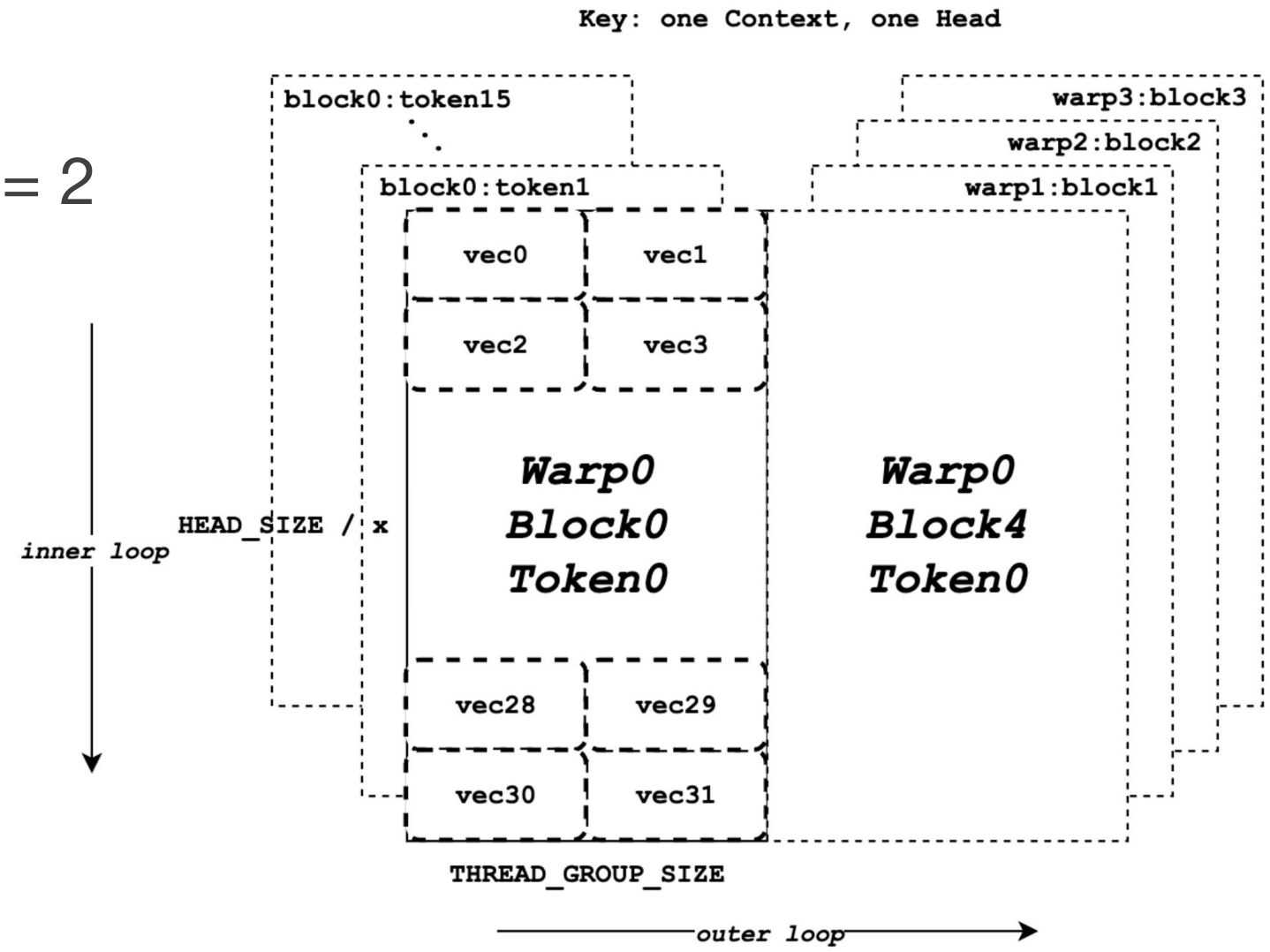
  store them on *register memory* as **k_vecs**
  - k_vecs will only be accessed by one thread once
  - q_vecs will be accessed by multiple threads multiple times

```
K_vec k_vecs[NUM_VECS_PER_THREAD]
```



Key

- THREAD_GROUP_SIZE = 2
- BLOCK_SIZE = 16
- HEAD_SIZE = 128
- x = 8
- 4 warps



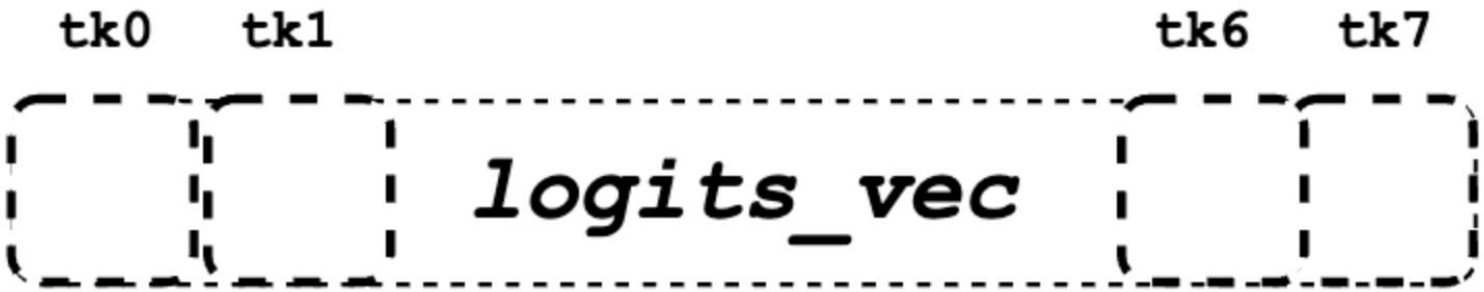
QK

- Note: there will be a cross thread group reduction happen in the **Qk_dot<>::dot**

```
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);
}
```

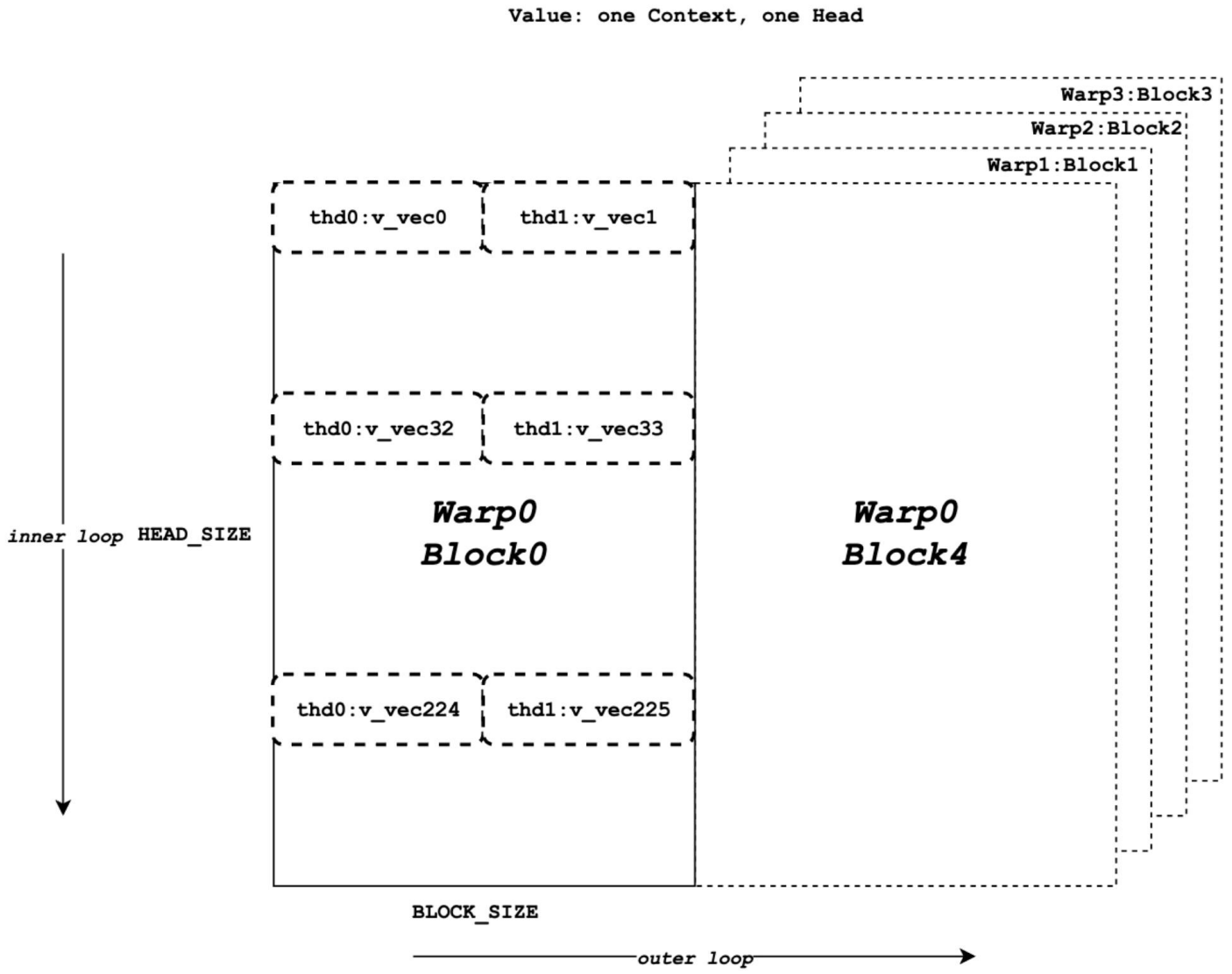
Softmax

- **logits** store the normalized softmax result of **qk** for all assigned context tokens.
  - will be used for dot multiplication with the value data



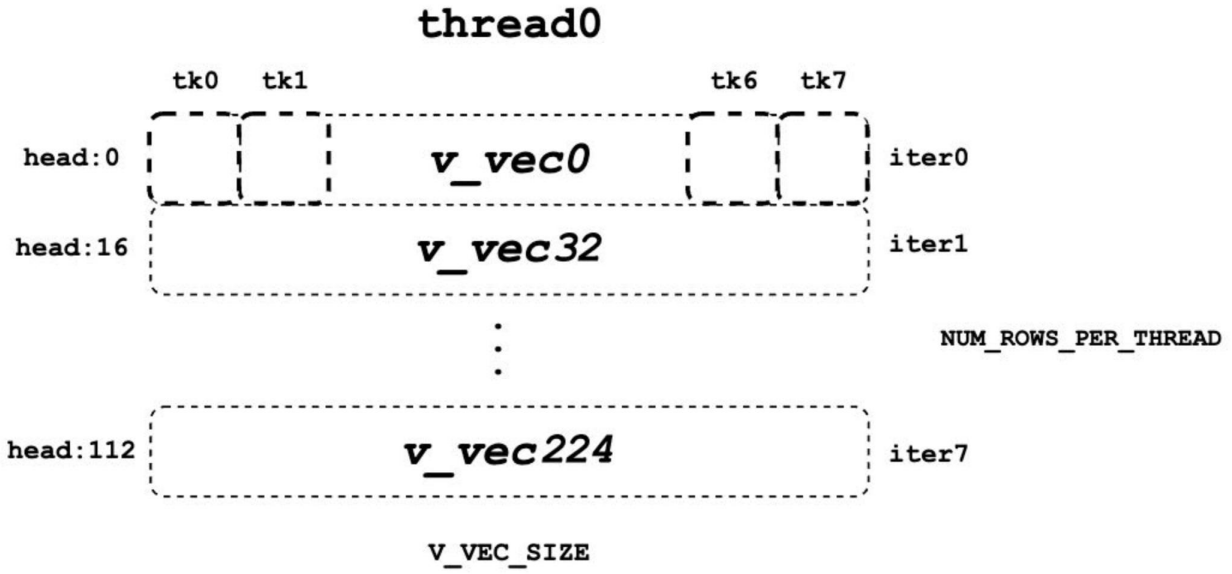
Value

- no thread group concept
- block of value data
  - HEAD_SIZE of rows
  - BLOCK_SIZE of columns
- elements
  - from the same column correspond to the same value token



Value

- Each thread fetches V_VEC_SIZE elements from the same V_VEC_SIZE of tokens at a time.




```
float accs[NUM_ROWS_PER_THREAD];
for ... { // Iteration over different blocks.
    logits_vec = ...
    for ... { // Iteration over different rows.
        v_vec = ...
        ...
        accs[i] += dot(logits_vec, v_vec);
    }
}
```