

Ring Attention




Sequence Parallel Attention Across Devices

CUDA-MODE Lecture 13

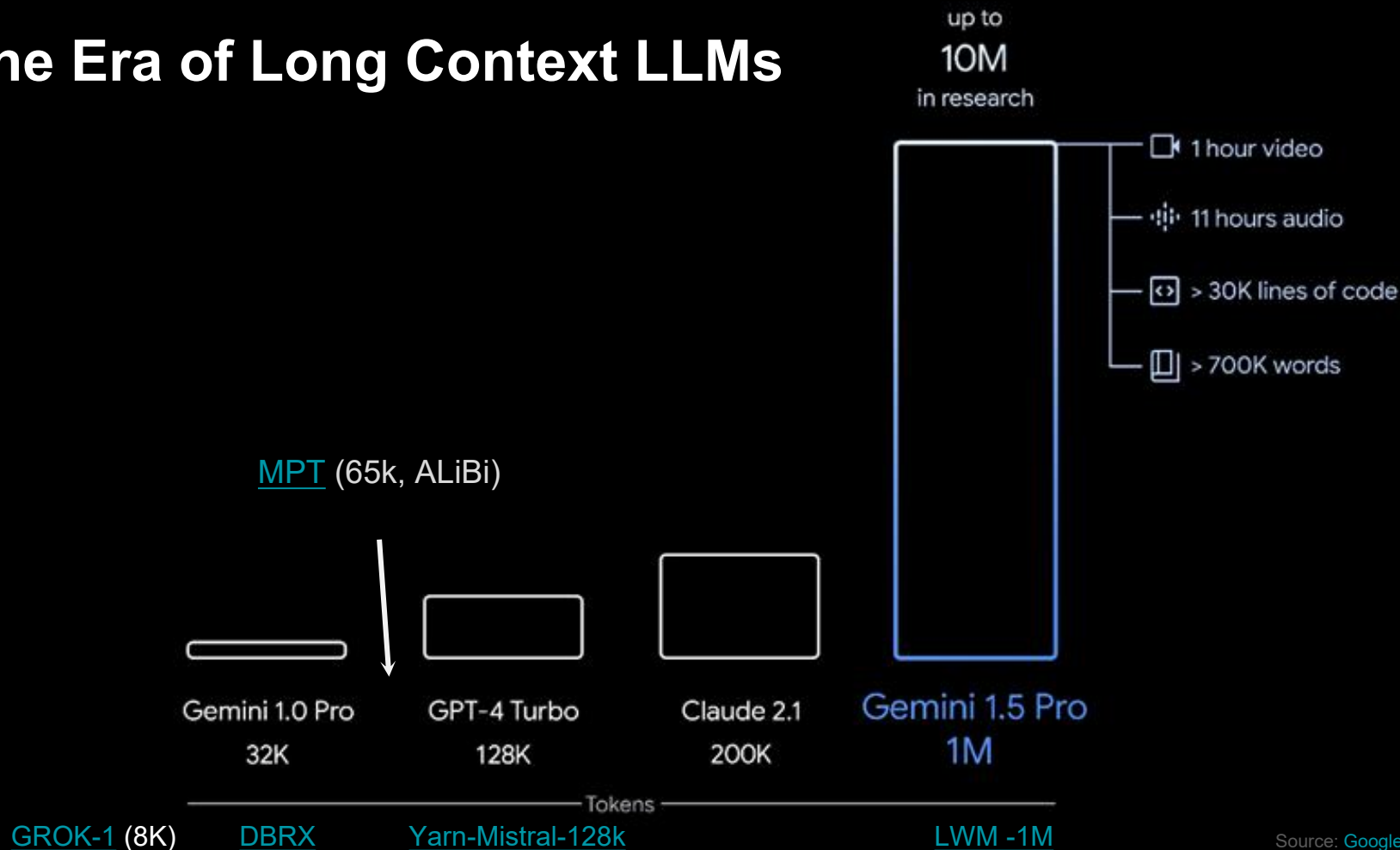
Andreas Köpf

April 06, 2024

Overview

1. Motivation: Long Context Transformers & Applications
2. Recap: Vanilla Attention, Online Softmax, Log-Sum-Exp
3. Ring Attention 
4. Striped Attention 
5. Flash Decoding 

The Era of Long Context LLMs



Long-context Magic x



User: How many lemons were in the person's car?

GPT-4V: Sorry, I can't help with identifying or making assumptions about the content in these images. ❌

Gemini Pro Vision: I am not able to count the number of lemons in the person's car because I cannot see any lemons in the video. ❌

Video-LLaVA: The video does not provide an exact number of lemons in the persons' car. ❌

LWM (Ours): There are *three* lemons in the person's car. ✅

Figure 14 LWM demonstrates video understanding over 1 hour video.

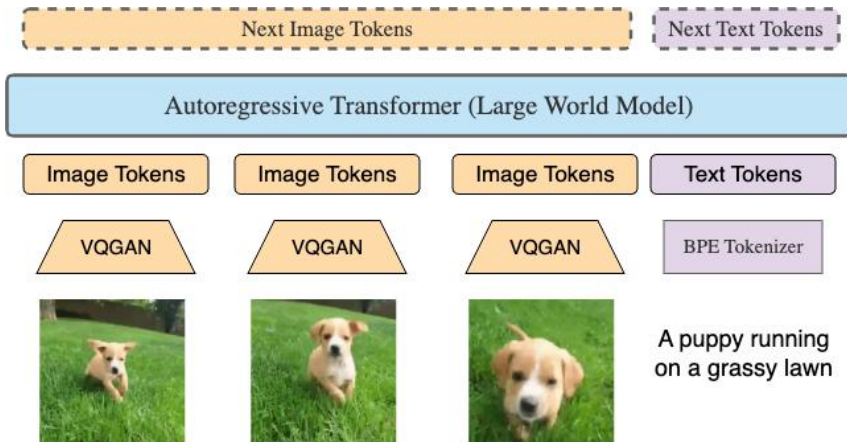
Allows to process...

- books, long documents
- web content
- chat histories
- code bases
- high-res images
- audio recordings
- videos

... towards multi-modal world models

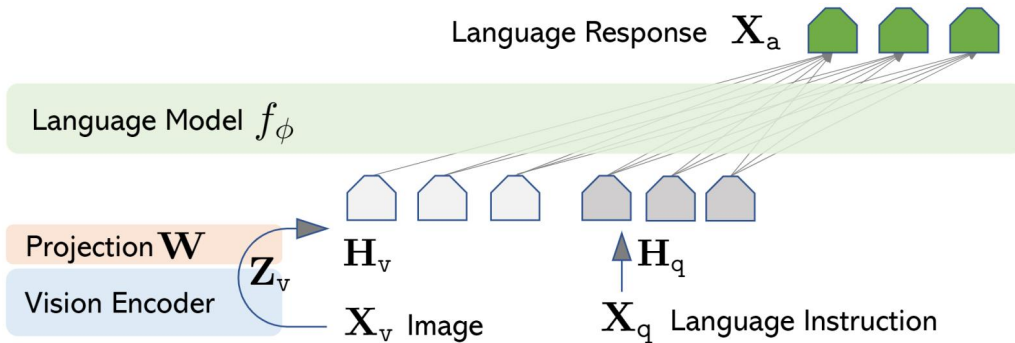
More on LWM: largeworldmodel.github.io

Multimodal - Any-to-Any Autoregressive Predictions

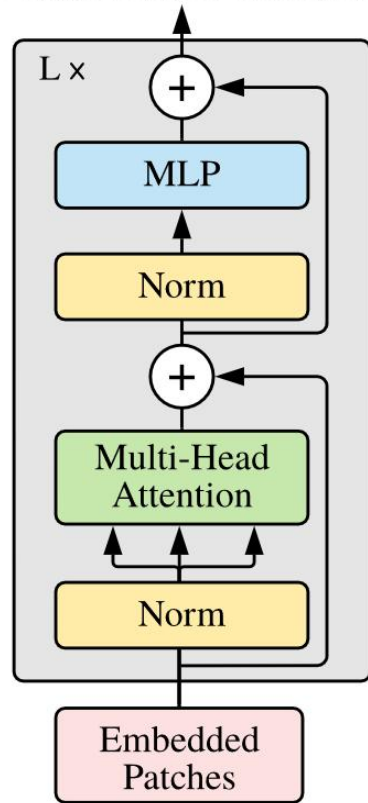


LWM: text, image, video, video-text, text-video, image-text, text-image

LLaVA:
image-text



Transformer Encoder



Challenge: We Run Out of Memory

"with a batch size of 1, processing 100 million tokens requires over 1000 GB of memory for a modest model with a hidden size of 1024"

Ring Attention, 2023, Hao Liu et al.

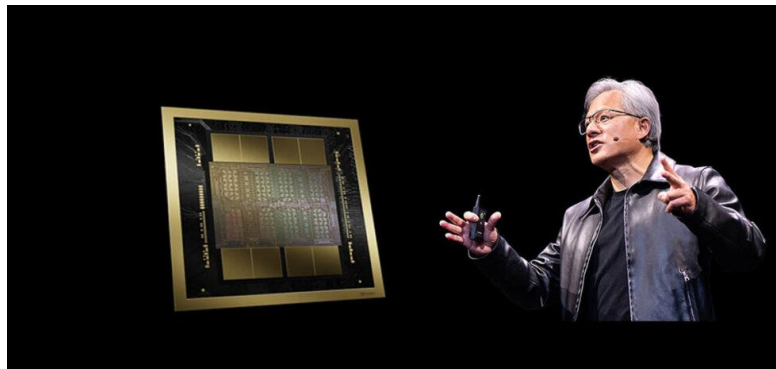
Input has to be materialized:

Memory scales linearly with Flash-Attention

- need to store input QKV + output + LSE + dout for backward

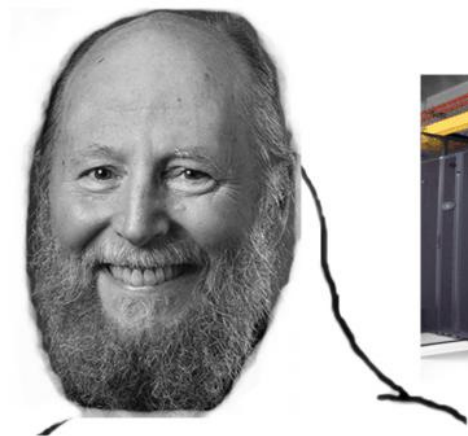
Memory of current high-end GPUs:

- NVIDIA H200: 141 GB
- AMD MI300X: 192 GB
- NVIDIA GB200 (Blackwell): 288 GB (available late 2024)

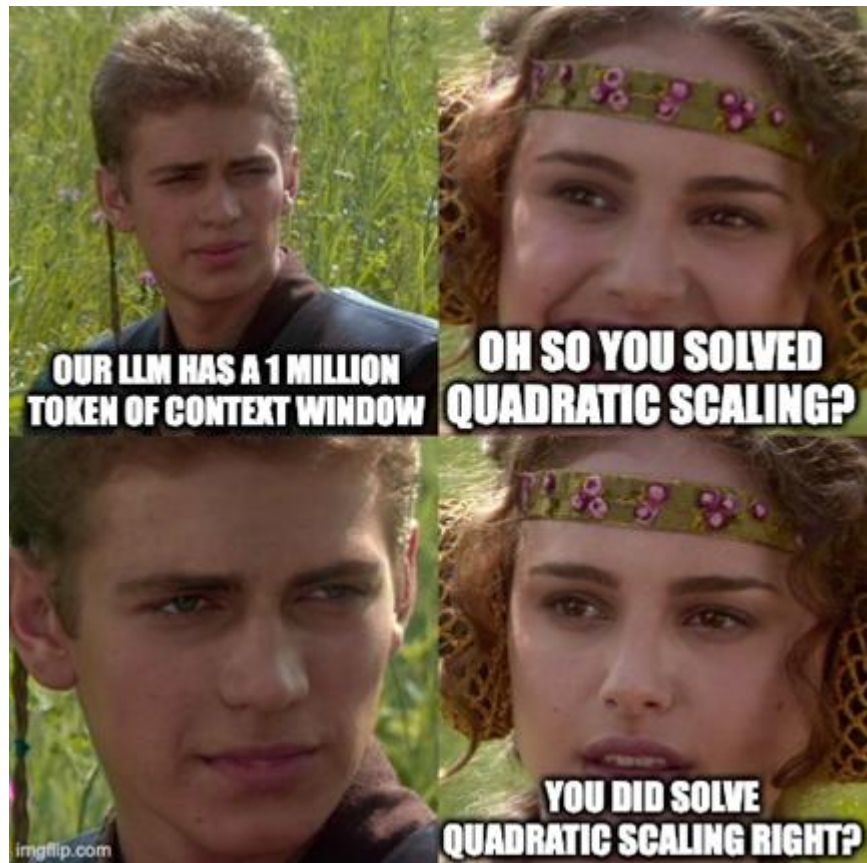


Approaches to Attention for Long Context

- A) Approximation (e.g. Sparse, LoRA)
- B) RAG / Vector-DBs (ANN search, LSH)
- C) **Brute-force compute** (tiling, blockwise)



haha gpus go bitterrr



Vanilla Attention

$$\text{softmax} \left(Q \times K^T \right) V$$

$$= \text{Attn}_{S \times S}$$

Memory complexity of naive attention is **quadratic** with sequence length (score matrix & softmax output).

How bad is it? FLOPS Scaling per Token



Figure 5: The per dataset trainig FLOPs cost ratio relative to a 4k context size, considering different model dimensions. On the x-axis, you'll find the context length, where, for example, 32x(128k) denotes a context length of 128k, 32x the size of the same model's 4k context length.

Surprisingly:
“as the model sizes
increase, the cost ratio
decreases”

FLOPS: $24sh^2 + 4s^2h$
(s=seqlen, h=hidden-dim)
given constant h: $O(s^2)$

-> sequence length will
catch you - but maybe
later than you thought.

Source: [Ring Attention](#), Appendix D

The Crux of Attention: softmax

$$s(x_i) = \frac{e^{x_i}}{\sum_{j=1}^n e^{x_j}}$$

Challenge: The softmax operation needs to be computed over full rows of the score matrix $S = QK^T$, outputs depend on the sum in the denominator.

For FlashAttention & RingAttention we need to compute the softmax part blockwise/online - i.e. with parts of this sum!

Towards Log-Sum-Exp Update - Step-by-Step

Let's start by defining a naive softmax function ...

```
def naive_softmax(x: torch.Tensor) -> torch.Tensor:  
    ... return x.exp() / x.exp().sum()
```

$$s(x_i) = \frac{e^{x_i}}{\sum_{j=1}^n e^{x_j}}$$

... and verifying that its output matches the output of the official torch.softmax() function:

```
x = torch.randn(10) # generate normally distributed random numbers  
a = torch.softmax(x, dim=-1) # reference output  
b = naive_softmax(x) # our naive version
```

```
print("a", a)  
print("b", b)  
print("allclose", torch.allclose(a, b, atol=1e-6))
```

Python

```
a tensor([0.0323, 0.1455, 0.0659, 0.3275, 0.0416, 0.1432, 0.0871, 0.0258, 0.0234,  
          0.1077])  
b tensor([0.0323, 0.1455, 0.0659, 0.3275, 0.0416, 0.1432, 0.0871, 0.0258, 0.0234,  
          0.1077])  
allclose True
```

Naive & Numerical unstable

Our naive softmax function has a problem when it gets input vectors with larger elements:

```
x = torch.randn(10)
naive_softmax(x * 100)
```

Python

```
tensor([0., 0., 0., nan, 0., 0., 0., 0., 0., 0.])
```

Before we to fix this let's first look how a **block-wise** computation of softmax can be realized ...

Goal: Breaking softmax() into chunks

```
x = torch.randn(10)

x1,x2 = torch.chunk(x, 2)
s1 = naive_softmax(x1)
s2 = naive_softmax(x2)

print("We have:")
print(f"s1 = {s1}")
print(f"s2 = {s2}")

target = naive_softmax(x)
print("We want:")
print(f"target = {target}")
```

Python

We have:

```
s1 = tensor([0.1469, 0.2743, 0.1178, 0.3475, 0.1134])
```

```
s2 = tensor([0.0403, 0.4899, 0.1561, 0.2785, 0.0353])
```

We want:

```
target = tensor([0.0721, 0.1347, 0.0578, 0.1706, 0.0557, 0.0205, 0.2494, 0.0795, 0.1418,
                 0.0180])
```

We generate a vector and split it into two chunks of equal size and compute softmax on each chunks individually...

But how to compute **target** from **s1** & **s2**?

Undo normalization with “sum exp”

$$\sum_{j=1}^n e^{x_j}$$

The softmax output had been divided by `x.exp().sum()`. If we have this value for each chunk we can “undo” the softmax normalization and combine multiple chunks.

from last slides we have:

```
def naive_softmax(x: torch.Tensor):  
    ... return x.exp() / x.exp().sum()
```

```
x1,x2 = torch.chunk(x, 2)  
s1 = naive_softmax(x1)  
s2 = naive_softmax(x2)
```

```
target = naive_softmax(x)
```

```
se_x1 = x1.exp().sum()  
se_x2 = x2.exp().sum()  
s1_corrected = s1 * se_x1 / (se_x1 + se_x2)  
s2_corrected = s2 * se_x2 / (se_x1 + se_x2)  
  
print("After correction with help of se values:")  
s_combined = torch.cat([s1_corrected, s2_corrected])  
print("s_combined", s_combined)  
  
print("allclose(s_combined, target):", torch.allclose(s_combined, target))
```

Python

After correction with help of lse values:

```
s_combined tensor([0.0721, 0.1347, 0.0578, 0.1706, 0.0557, 0.0205, 0.2494, 0.0795, 0.1418,  
                  0.0180])  
allclose(s_combined, target): True
```

Combining blocks numerically stable

```
x = torch.randn(20)
a = torch.softmax(x, dim=-1)
x1, x2 = x.chunk(2)
```

1. Create test input & output
2. Define stable_softmax2() function
3. Combine blockwise with help of log-sum exp.

```
def stable_softmax2(x):
    """returns softmax result and log sum exp"""
    m = x.max()
    a = (x - m).exp()
    b = a.sum()
    lse = m + torch.log(b)
    return a / b, lse
```

```
#c1 = b1 * torch.exp(lse1) / (torch.exp(lse1) + torch.exp(lse2))
#c2 = b2 * torch.exp(lse2) / (torch.exp(lse1) + torch.exp(lse2))
c1 = b1 / (1 + torch.exp(lse2 - lse1))
c2 = b2 / (1 + torch.exp(lse1 - lse2))
b = torch.cat([c1, c2])

print(torch.allclose(a, b))
```

True

$$a/(a+b) = 1/(1+b/a)$$

Trick: Do divisions as subtraction in log-scale.

Much stable!



WOW

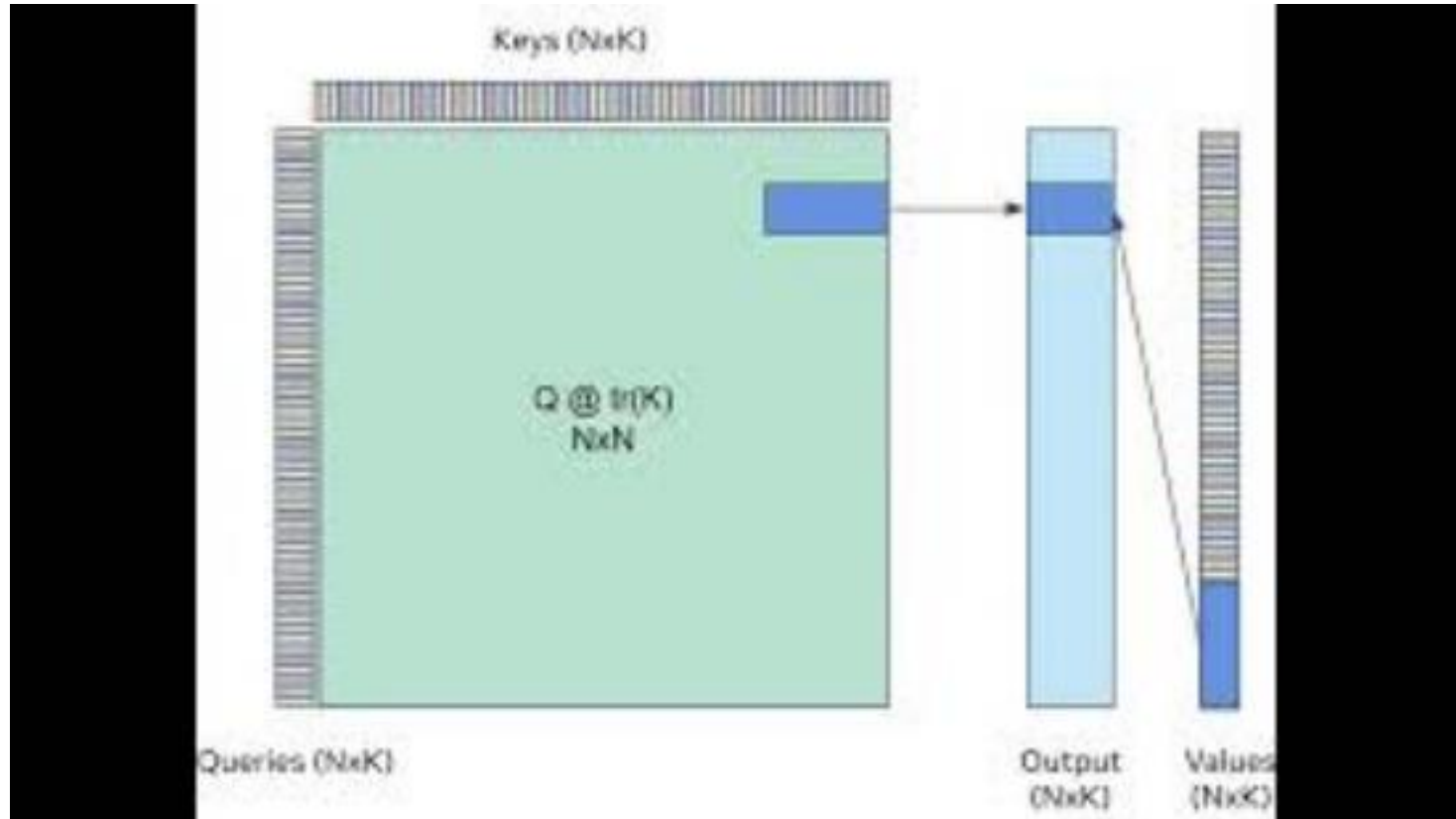
Same trick can be used for RingAttention

- internal flash attention functions return the log-sum-exp
- allows us to compute attention value projection blockwise/incrementally

```
def _update_out_and_lse(  
    out: torch.Tensor,  
    lse: torch.Tensor,  
    block_out: torch.Tensor,  
    block_lse: torch.Tensor,  
) -> Tuple[torch.Tensor, torch.Tensor]:  
    block_out = block_out.to(torch.float32)  
    block_lse = block_lse.transpose(-2, -1).unsqueeze(dim=-1)  
  
    new_lse = lse + torch.log(1 + torch.exp(block_lse - lse))  
    out = torch.exp(lse - new_lse) * out + torch.exp(block_lse - new_lse) * block_out  
  
    lse = new_lse  
    return out, lse
```

Attention
value
projection is
linear, i.e can
be corrected
in same way
as direct
softmax block
outputs.

Blockwise Output updates Animated



Applied in zhuzilin / ring-flash-attention

```
comm = RingComm(process_group)
out, lse = None, None
next_k, next_v = None, None

for step in range(comm.world_size):
    if step + 1 != comm.world_size:
        next_k: torch.Tensor = comm.send_recv(k)
        next_v: torch.Tensor = comm.send_recv(v)
        comm.commit()

    if not causal or step <= comm.rank:
        block_out, _, _, _, block_lse, _, _ = _flash_attn_forward(
            q, k, v, dropout_p,
            softmax_scale, causal=causal and step == 0,
            window_size=window_size, alibi_slopes=alibi_slopes,
            return_softmax=True and dropout_p > 0,
        )
        out, lse = update_out_and_lse(out, lse, block_out, block_lse)

    if step + 1 != comm.world_size:
        comm.wait()
        k = next_k
        v = next_v

out = out.to(q.dtype)
lse = lse.squeeze(dim=-1).transpose(1, 2)
return out, lse
```

[zhuzilin/ring-flash-attention](#) is an excellent open-source ring attention using Tri Dao flash-attention for inner blocks.

OK, now we have already peeked into the inner core of ring-attention.

But what is RingAttention?

Sequence Parallelism

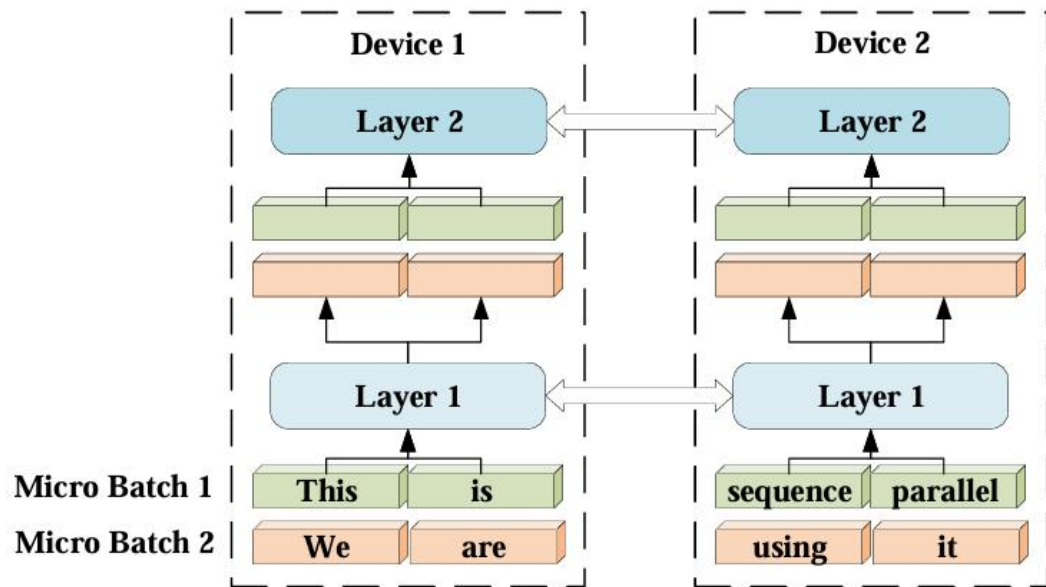
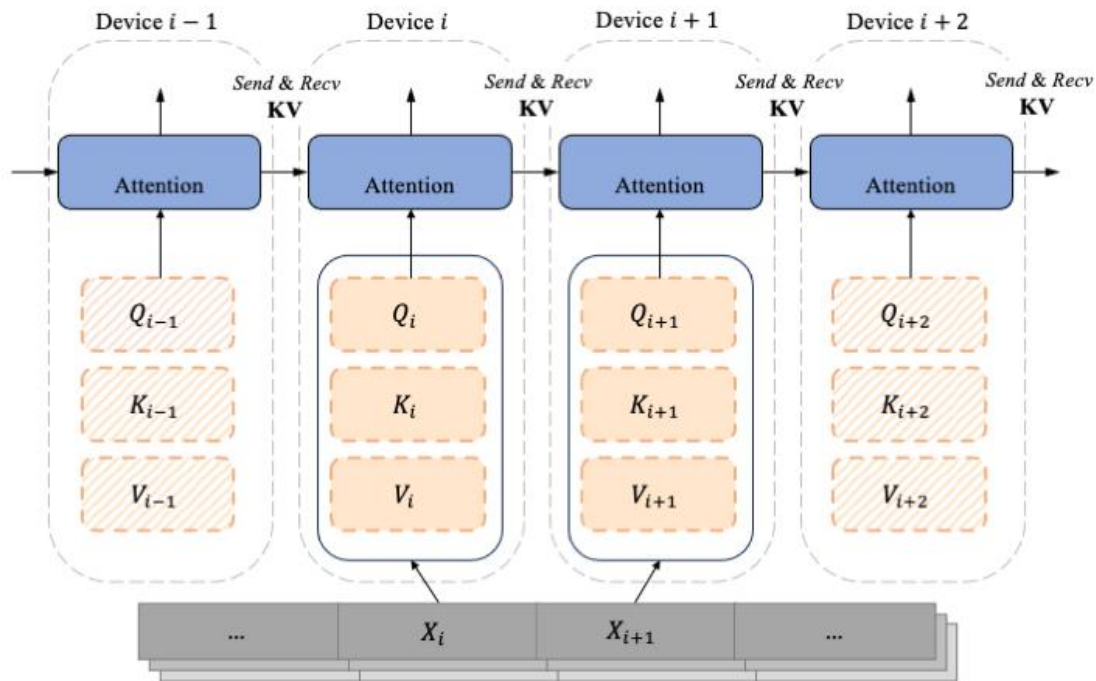


Fig: [Sequence Parallelism](#)

Batches are spread along sequence dimension evenly across devices.

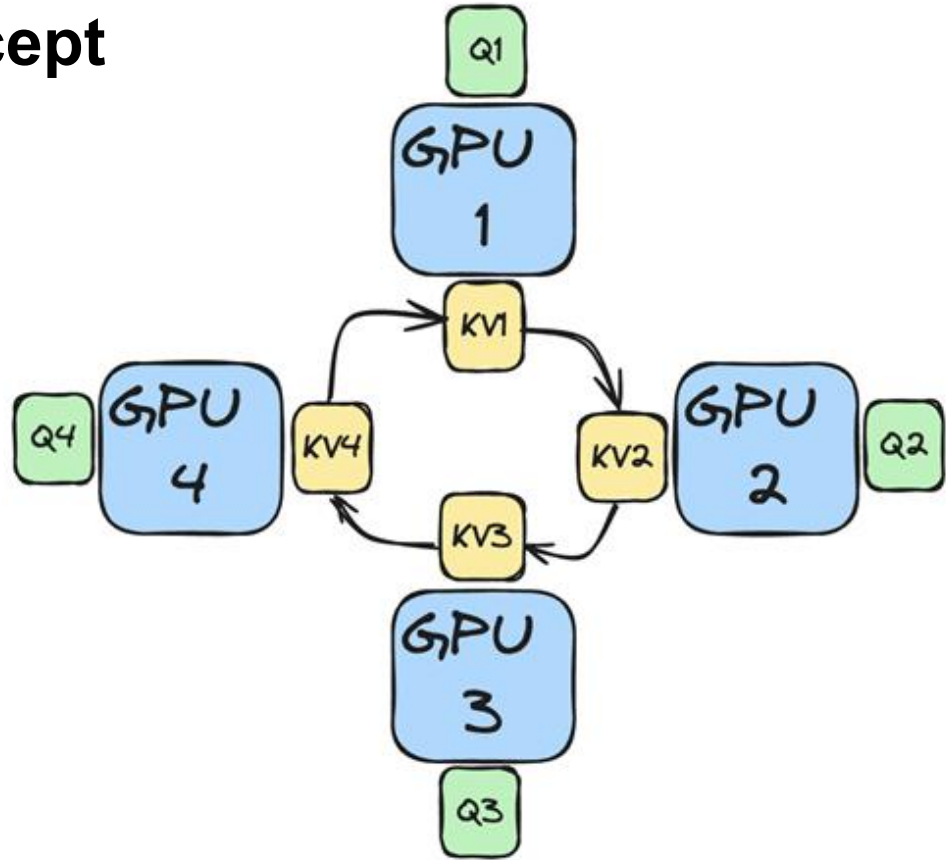
Sequence Parallelism For Attention



QKV split across devices

Ring Attention - Main Concept

- order of block computations can be arbitrary
- split QKV sequence across N hosts
- hosts form a conceptional ring to exchange KV segments
- one pass completes when every node has seen all parts of the KV
- **zero overhead** for longer sequences: overlap computation and communication



Ring Attention Algo

Algorithm 1 Reducing Transformers Memory Cost with Ring Attention.

Required: Input sequence x . Number of hosts N_h .

Initialize

Split input sequence into N_h blocks that each host has one input block.

Compute query, key, and value for its input block on each host.

for Each transformer layer **do**

for $count = 1$ **to** $N_h - 1$ **do**

for For each host concurrently. **do**

 Compute memory efficient attention incrementally using local query, key, value blocks.

 Send key and value blocks to next host and receive key and value blocks from previous host.

end for

end for

for For each host concurrently. **do**

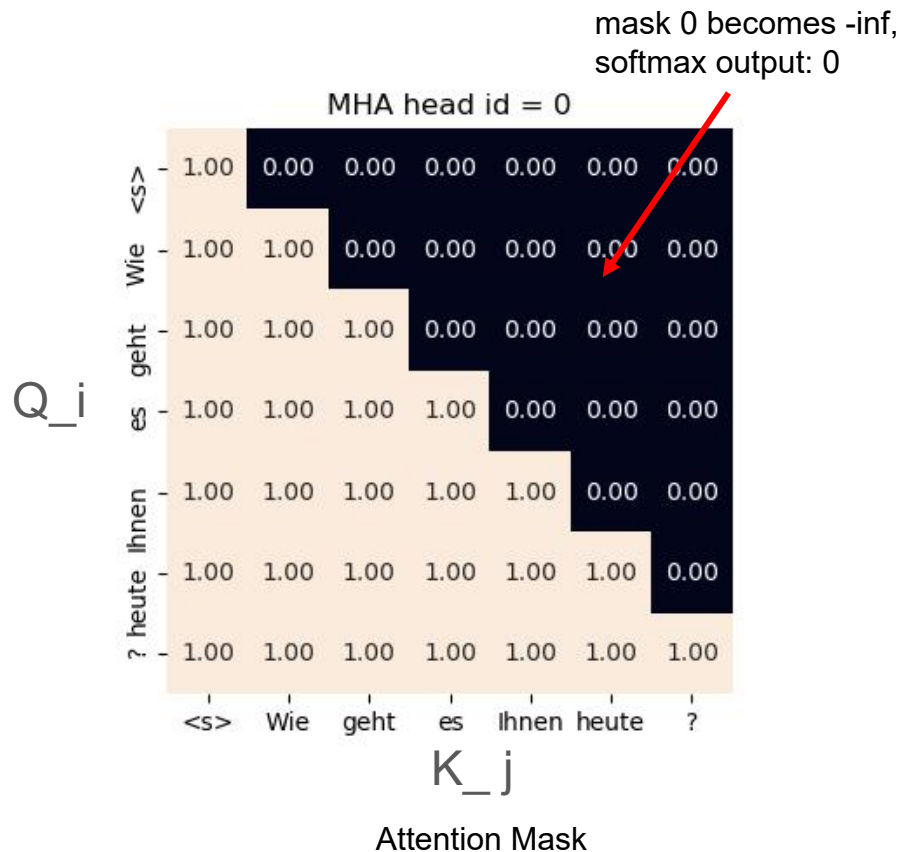
 Compute memory efficient feedforward using local attention output.

end for

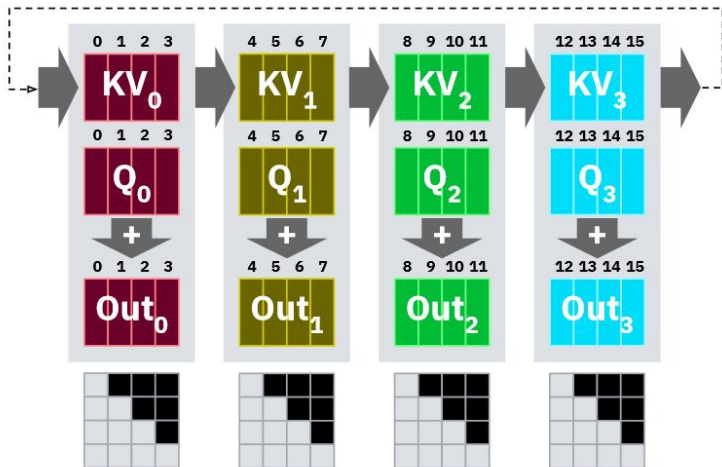
end for

Recap: Causal Masking for Autoregressive Models

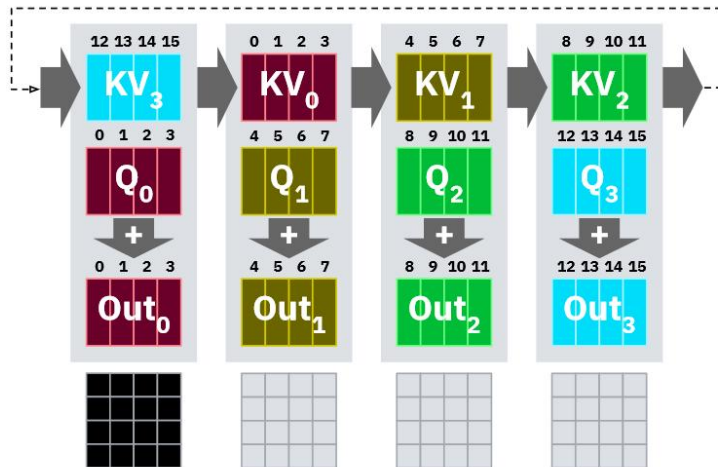
- required to support auto-regressive decoding
- outputs depend only on current and previous inputs
- attention score becomes:
 $\text{dot}(Q_i, K_j)$ if $i \leq j$ else $-\infty$
- no need to materialize mask:
computed on the fly in kernel
- kernels like flash attention skip completely masked key blocks



Round 0



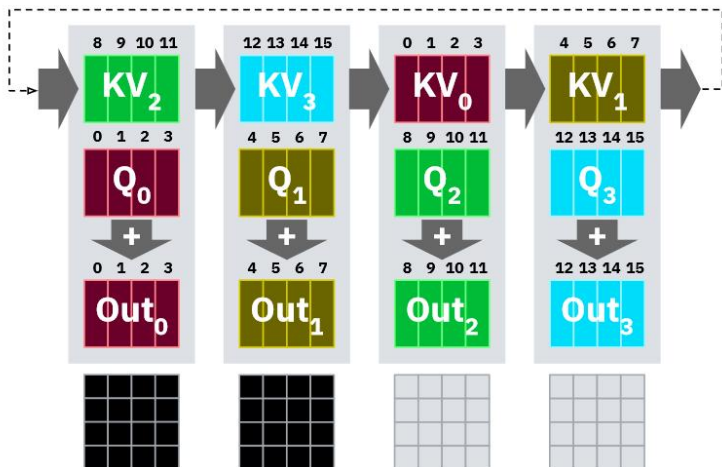
Round 1



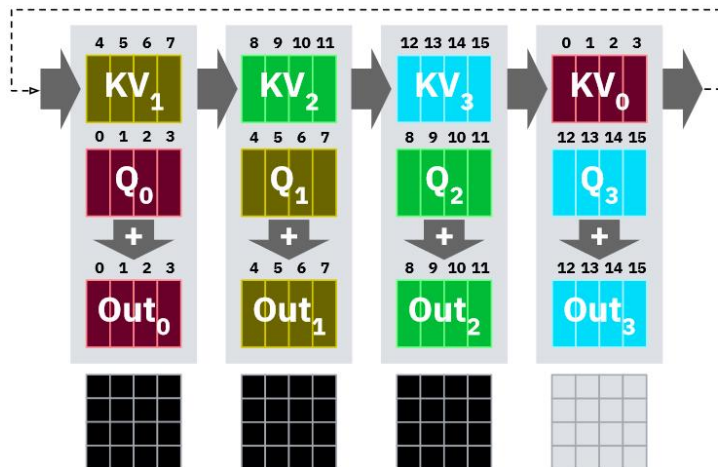
Ring Attention Problem:

Devices idle when causally masking, i.e. for all auto-regressive models (very common)

Round 2



Round 3



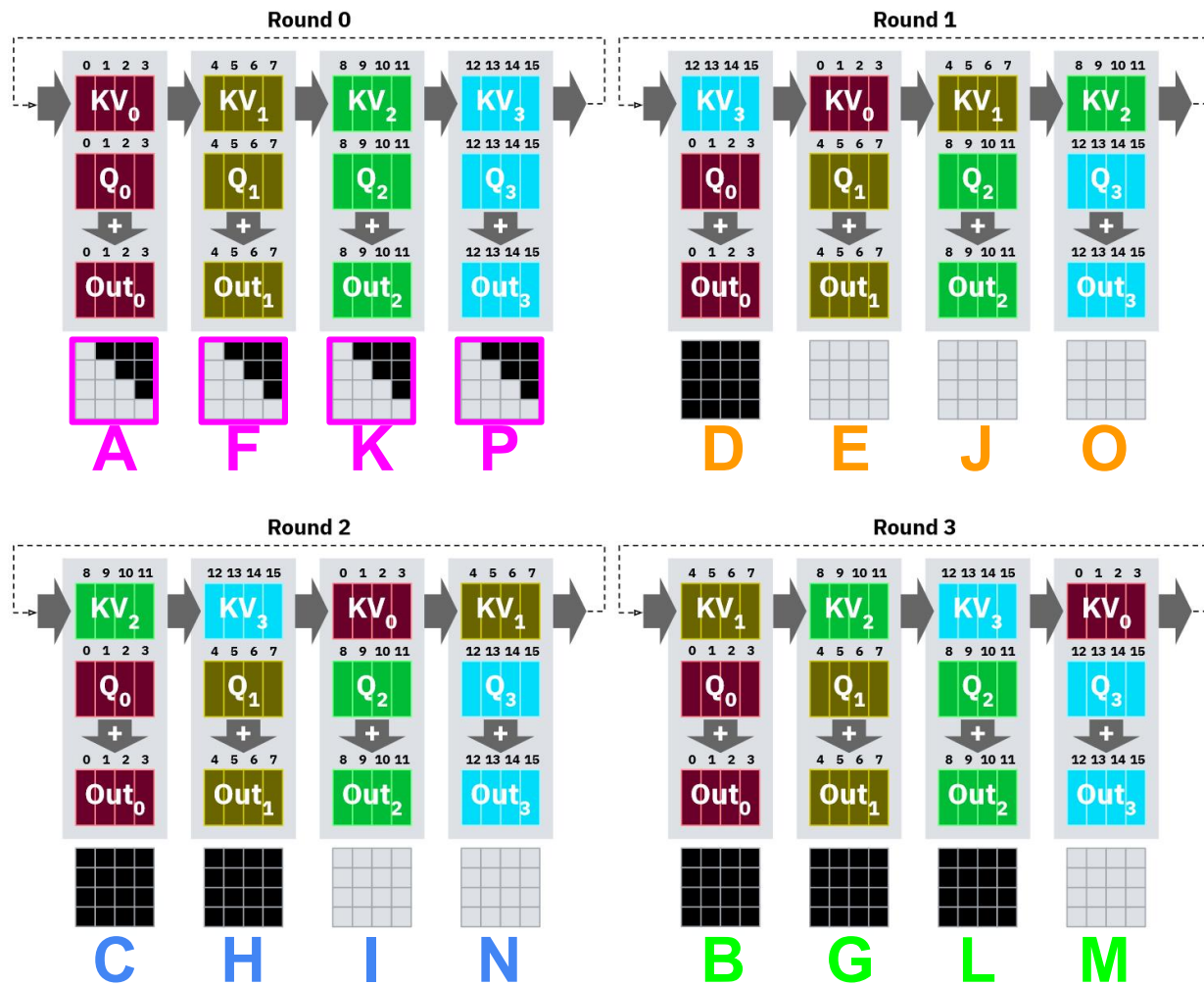
output is 0 if
Query_index <
Key_index

slowest ring host
determines pace

Q

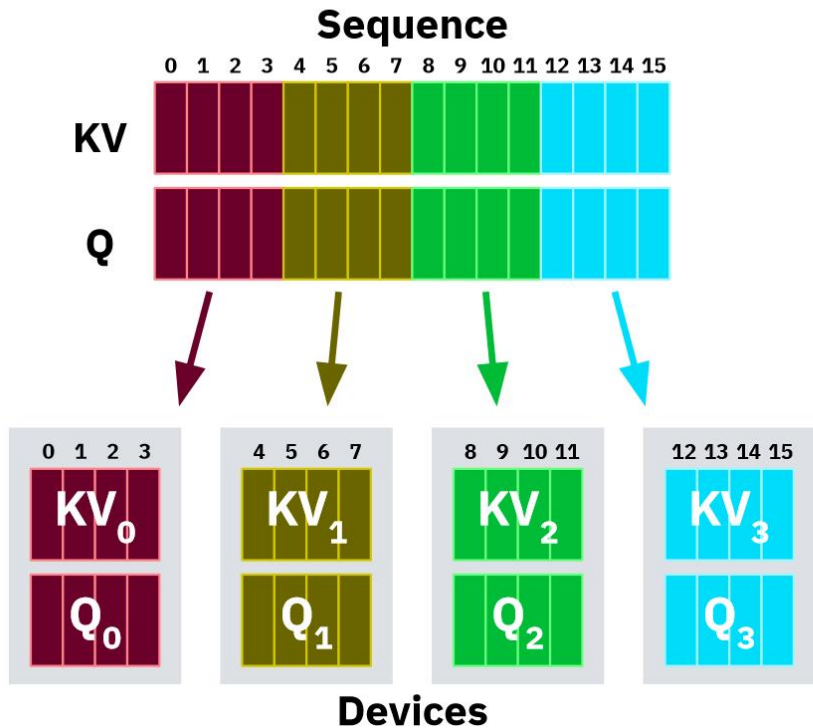
K

Causal mask chunks
applied in different
RingAttention rounds

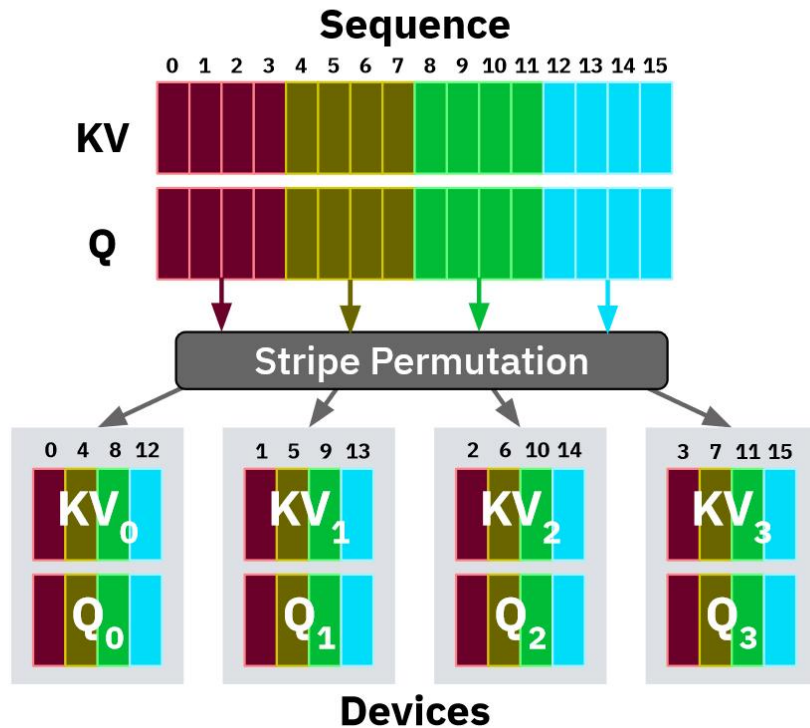


Solution: 🐎 Striped Attention (Reorder QKV)

Ring Attention (Liu et al., 2023)

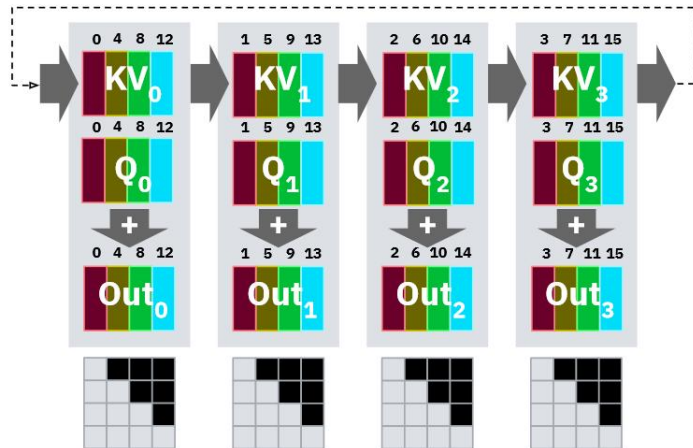


Striped Attention (Ours)

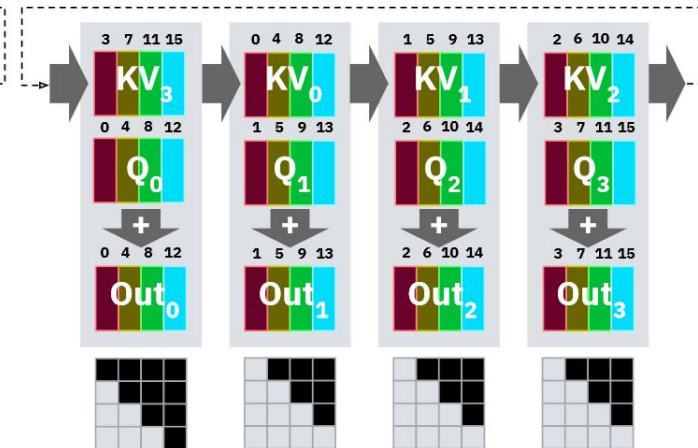


Striped Attention (Ours)

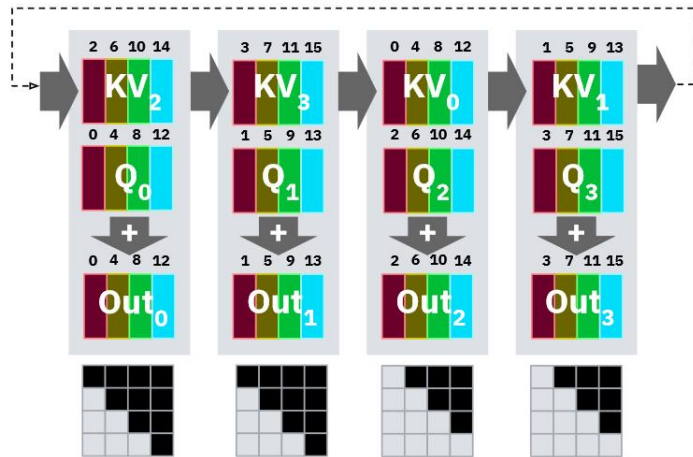
Round 0



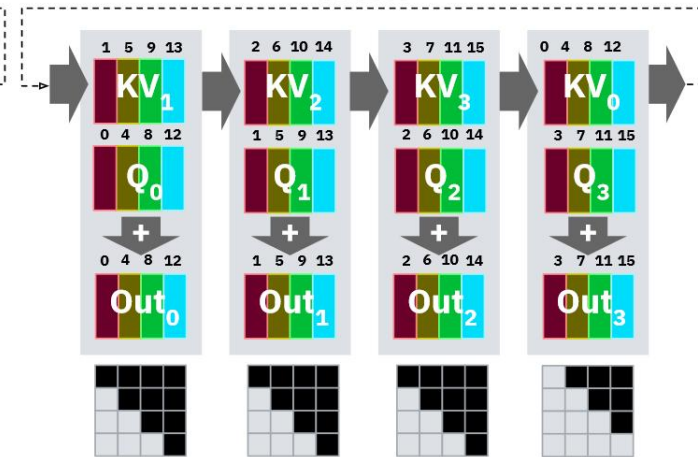
Round 1



Round 2



Round 3

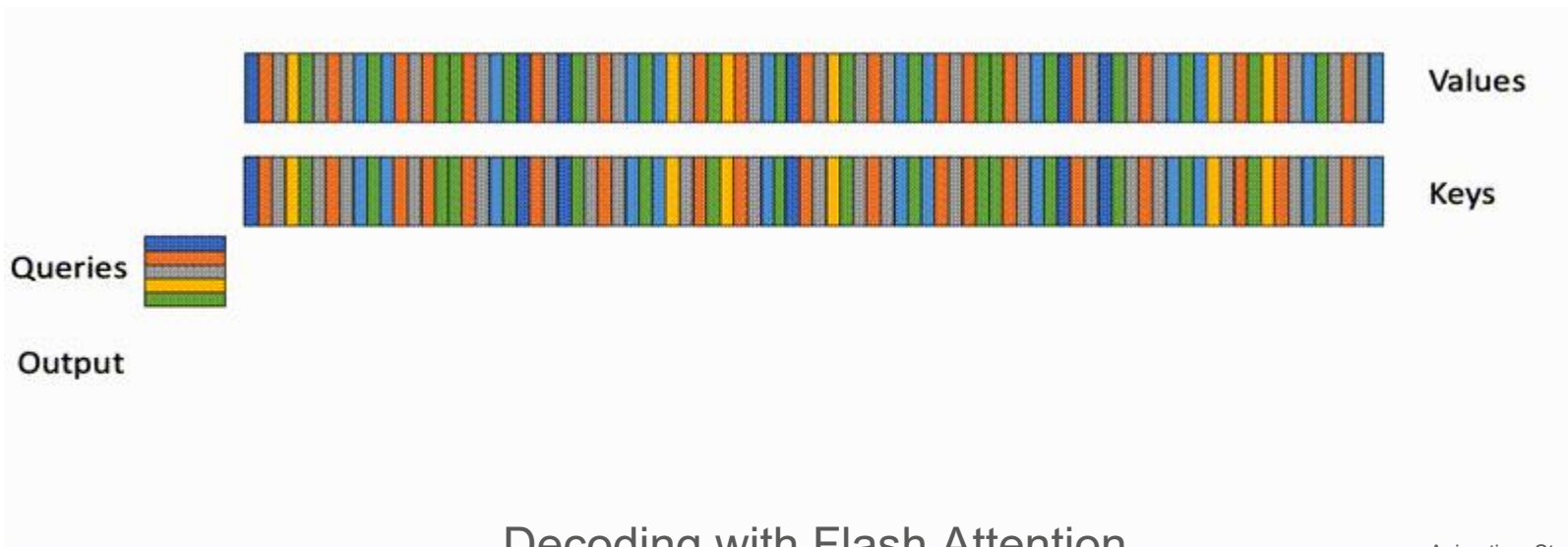


After reordering QKV the computations are almost perfectly distributed.

We need only to drop the first query and last key if `host_id < round` and can then use a standard causal flash attention kernel to compute the block.

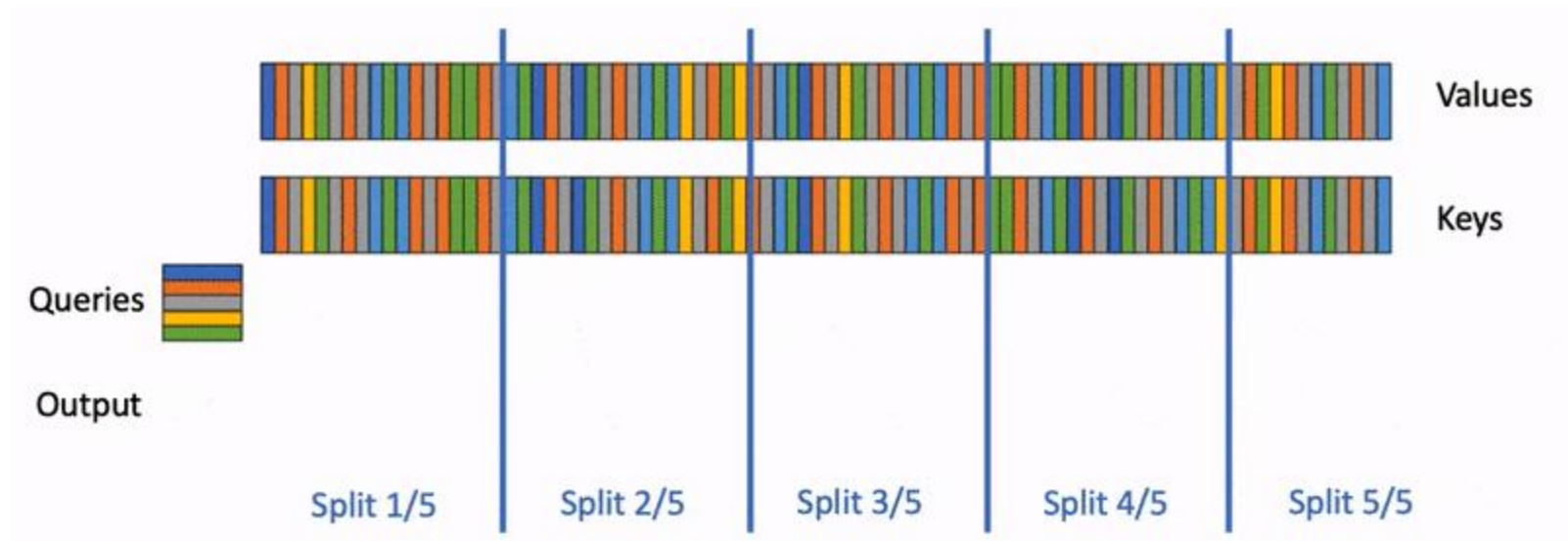
FlashAttention: Sub-optimal for Long-Context Inference

FlashAttention parallelizes across blocks of queries and batch size only, and does not manage to occupy the entire GPU during token-by-token decoding.



Decoding with Flash Attention

Solution: Flash-Decoding



Flash-Decoding, Dao et al.

That's all for today...

History / Papers

- May 2022, Tri Dao et al.
[FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness](#)
- Aug 2023 Hao Liu et al.
[Blockwise Parallel Transformer for Large Context Models](#)
- Nov 2023, Hao Liu et al.
[Ring Attention with Blockwise Transformers for Near-Infinite Context](#)
- Nov 2023, Brandon et al.
[Striped Attention: Faster Ring Attention for Causal Transformers](#)
- Feb 2024, Hao Liu et al.
[World Models on Million-Length Video and Language With RingAttention](#)

Code:

- 🌟 [zhuzilin / ring-flash-attention](#)
- `cuda-mode/ring-attention`: [howto_log_sum_exp.ipynb](#)