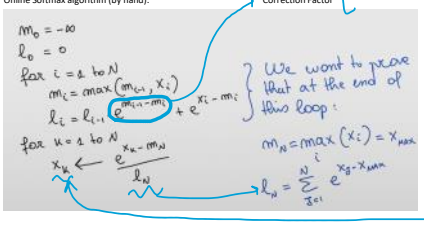
FlashAttention in Triton

Tuesday, March 18, 2025 5:17 PM

Flash Attention 1:
- Computations are on GPU
- Accessing HBM (DRAM, Global, largest memory) is MUCH slower than accessing Shared Memory
- Problem: Calculating Attention is I/O Bound -> as it keeps needing to access Global Memory
- Solution: Computing Attention INSIDE Shared Memory itself, not accessing Global Memory at all
- Consequence: Computation of Attention is MUCH CLOSER to GPU cores
- How we'll do it:
  - Split Attention input into smaller blocks that can FIT INSIDE Shared Memory using Tiling/Block Matmul
  - Compute Attention using K, Q, V for EACH Block (of the entire Matrix) in the Shared Memory (One Block at a time, Shared Memory can fit only 1 Block), then COPY this back to Global Memory

FlashAttention1 vs FlashAttention2:
- Structure of 2 'for' loops in FA1:
  - 1st loop: K (Keys)
  - 2nd loop: Q (Queries)
- Structure of 2 'for' loops in FA2:
  - 1st loop: Q (Queries)
  - 2nd loop: K (Keys)

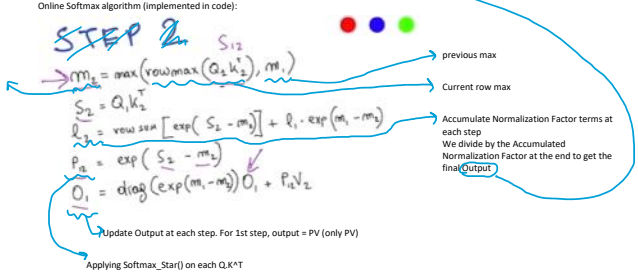
Online Softmax algorithm (by hand):



Correction Factor

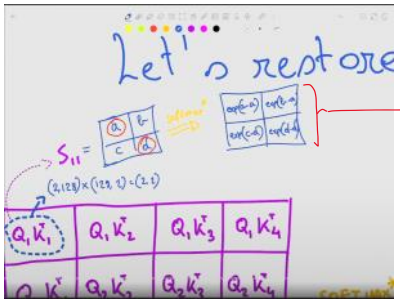
If:
- Current Max > Previous Max: Correction Factor corrects
- If Current Max = Previous Max: Correction Factor becomes 1, no correction takes place

Online Softmax algorithm (implemented in code):



- Q and K are matrix BLOCKS (tiles/blocks from tiling/blocking)

DOUBTS



x_max does over each row or over entire matrix?
=> over each row

FlashAttention2 (using the Online Softmax algo above)

Algorithm 1 FLASHATTENTION-2 forward pass

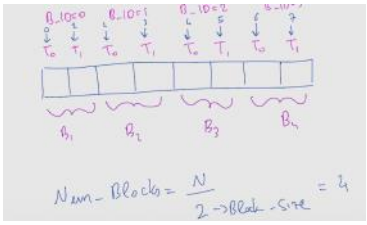
Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM, block sizes B_c, B_r.

1: Divide $\mathbf{Q}$ into $T_r = \left\lceil \frac{N}{B_r} \right\rceil$ blocks $\mathbf{Q}_1, \dots, \mathbf{Q}_{T_r}$ of size $B_r \times d$ each, and divide $\mathbf{K}, \mathbf{V}$ in to $T_c = \left\lceil \frac{N}{B_c} \right\rceil$ blocks $\mathbf{K}_1, \dots, \mathbf{K}_{T_c}$ and $\mathbf{V}_1, \dots, \mathbf{V}_{T_c}$, of size $B_c \times d$ each.
2: Divide the output $\mathbf{O} \in \mathbb{R}^{N \times d}$ into T_r blocks $\mathbf{O}_i, \dots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, and divide the logsumexp L into T_r blocks $L_i, \dots, L_{T_r}$ of size B_r each.
3: **for** $1 \le i \le T_r$ **do** ← FOR EACH Q_i BLOCK
4: Load $\mathbf{Q}_i$ from HBM to on-chip SRAM.
5: On chip, initialize $\mathbf{O}_i^{(0)} = (0)_{B_r \times d} \in \mathbb{R}^{B_r \times d}$, $\ell_i^{(0)} = (0)_{B_r} \in \mathbb{R}^{B_r}$, $m_i^{(0)} = (-\infty)_{B_r} \in \mathbb{R}^{B_r}$.
6: **for** $1 \le j \le T_c$ **do** ← FOR EACH K_j BLOCK
7: Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.
8: On chip, compute $\mathbf{S}_i^{(j)} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.
9: On chip, compute $m_i^{(j)} = \max(m_i^{(j-1)}, \text{rowmax}(\mathbf{S}_i^{(j)})) \in \mathbb{R}^{B_r}$, $\tilde{\mathbf{P}}_i^{(j)} = \exp(\mathbf{S}_i^{(j)} - m_i^{(j)}) \in \mathbb{R}^{B_r \times B_c}$ (pointwise), $\ell_i^{(j)} = e^{m_i^{(j-1)} - m_i^{(j)}} \ell_i^{(j-1)} + \text{rowsum}(\tilde{\mathbf{P}}_i^{(j)}) \in \mathbb{R}^{B_r}$.
10: On chip, compute $\mathbf{O}_i^{(j)} = \text{diag}(e^{m_i^{(j-1)} - m_i^{(j)}})^{-1} \mathbf{O}_i^{(j-1)} + \tilde{\mathbf{P}}_i^{(j)} \mathbf{V}_j$.
11: **end for**
12: On chip, compute $\mathbf{O}_i = \text{diag}(\ell_i^{(T_c)})^{-1} \mathbf{O}_i^{(T_c)}$.
13: On chip, compute $L_i = m_i^{(T_c)} + \log(\ell_i^{(T_c)})$.
14: Write $\mathbf{O}_i$ to HBM as the i-th block of $\mathbf{O}$.
15: Write L_i to HBM as the i-th block of L.
16: **end for**
17: Return the output $\mathbf{O}$ and the logsumexp L.

CUDA:
- We have to tell each Thread what to do, ie software engineer determines mapping between Thread and its Data Elements (CUDA knows only # of Blocks & # of Threads [in each Block])
- When running 'N'-threads in parallel (done on each core), when we ask CUDA to run the Nthreads in parallel, it will:
  - Allocate N threads
  - Assign a unique 'Thread Identifier' to each thread
  - Run the threads according to the Thread Identifier assigned to it
-

Why we have the 'if (i < N):' in cuda_vector_add_simple() method:
- Threads are launched by CUDA in multiples of 32 (ie, launched as a warp, 1 warp has 32 threads)
  Eg: if no of threads intended by software engineer is 34 (ie, 34 elements in vector, 32+2), CUDA launches 64 threads

So, supposing we want to launch 8 threads (as we have 8 elements in vector), we have to tell the remaining 24 threads to NOT do anything.
- The N threads being launched have the same Control Unit. This means: they have run the same program, but on different data elements (SIMT programming model)
- Control Divergence:
  - In earlier case, 8 threads were launched, meaning remaining 24 threads were also launched, but remain IDLE as they don't do any work
  - This is called Control Divergence and causes lesser throughput, which software engineer should aim to minimize

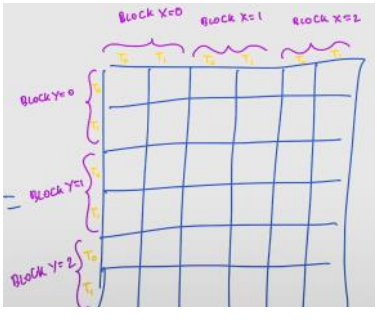


Eg: N = 8, Block Size = 2, GPU has 4 cores
So, # of Blocks = 8 / 2 = 4

Expression to access each element in N:
element_id = B_id * Block_size + thread_id
- Takes 0 or 1
- Block ID: 0, 1, 2 or 3

General formula to calculate # of Blocks:
# of Blocks = Total # of Data Elements (N) / Block Size
- # of Threads per block (considering 1 Thread per Data Element)

DOUBTS
1. Why does each Block have 2 Threads here?



Defining launch grid for 2D matrix (like on the left):
- Defining # of Block Rows and # of Block Columns

```
// Define the launch grid
int num_blocks_ROWS = (NUM_ROWS + ROWS_block_size - 1) / ROWS_block_size;
int num_blocks_COLS = (NUM_COLS + COLS_block_size - 1) / COLS_block_size;
```

# of Element Columns (decided from total Elements 'N')

# of Element Columns we set per Column Block

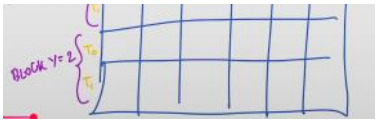
Fancy way of writing ceil(NUM_COLS / COLS_block_size)

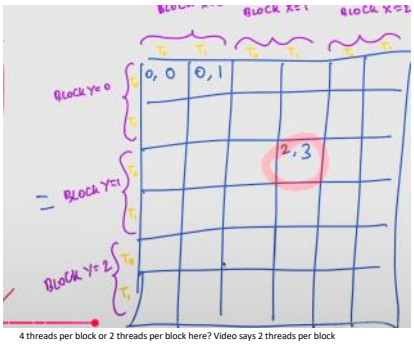
- # of Blocks we have & # of Threads we have:

```
dim3 grid(num_blocks_COLS, num_blocks_ROWS, 1);
dim3 block(COLS_block_size, ROWS_block_size, 1);
```

# of Blocks

# of Threads



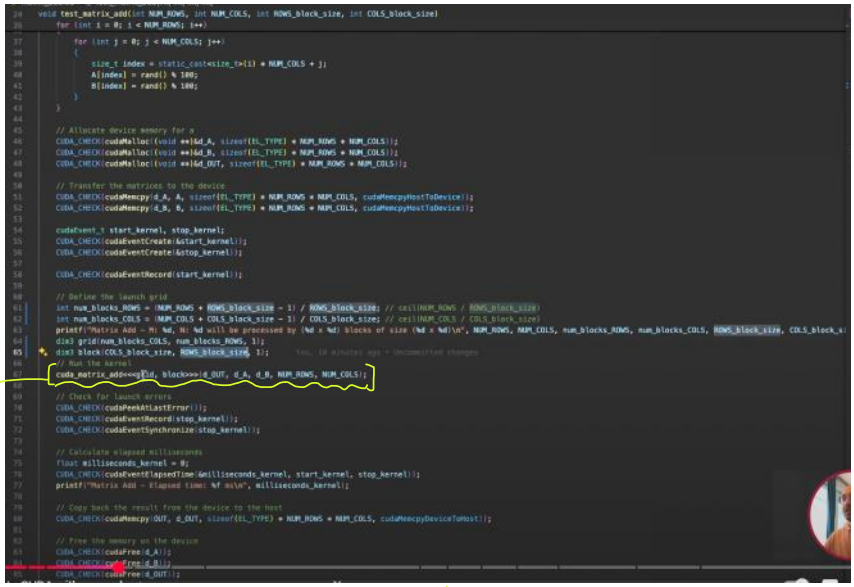


4 threads per block or 2 threads per block here? Video says 2 threads per block

Relationshio between warp and block (depends on threads per block)

- **Fewer than 32 threads per block:**
  Even if a block has fewer than 32 threads (say, 4 threads), the GPU still allocates a full warp (32 threads) for that block, but only the threads corresponding to your block are active while the remaining lanes remain inactive.
- **Exactly 32 threads per block:**
  The block perfectly fits into one warp.
- **More than 32 threads per block:**
  The block is divided into multiple warps. For example, if a block has 64 threads, it will be split into 2 warps.

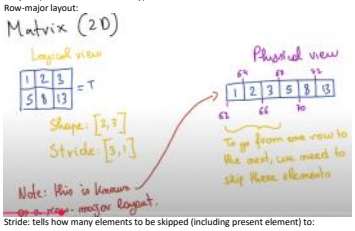
-



- Element ID expressions:

Tensor Layout (in CPU and GPU memory):

- Row-major layout:

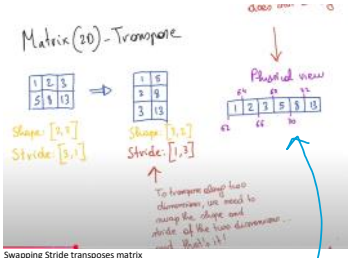


Stride: tells how many elements to be skipped (including present element) to:

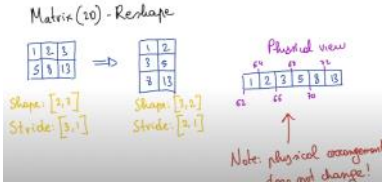
- Go to the next row dimension (eg: 3 here)
- Go to the next column dimension (eg: 1 here)

In summary:

- we skip 3 elements to go to next row
- we skip 1 element to go to next element in same row



Swapping Stride transposes matrix



Tensor Reshaping

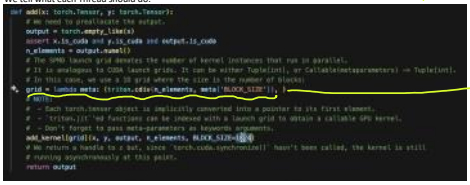
In conclusion,
Stride is used for:

- Indexing Tensor (pointer to starting element of physical location)
- Allows reshaping Tensor for free
- Allows Transposing: by swapping dims

Triton:

- Unlike CUDA (where we need to define how many Blocks and how many Threads per Block), Triton only needs # of Blocks (it decides # of Threads per Block on its own)
  We tell what each Thread should do.

Vector addition example

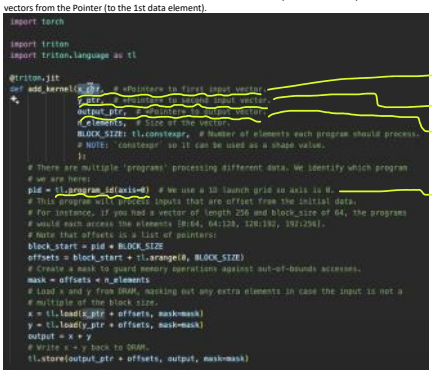


Defining triton grid:

- # of blocks = ceil(n_elements, block_size)

But here, block size is not # of Threads per block (unlike CUDA)
It is: # of Data Elements each block processes (in CUDA, assumption is: 1 thread per data element in the same cycle, that's not the assumption here)

- Unlike Pytorch, Triton does not take entire vectors just as they are (full matrices, broadcasting etc operations). It takes the POINTER to the 1st Data Element of that vector in memory. We have to compute the indices for the vectors from the Pointer (to the 1st data element).



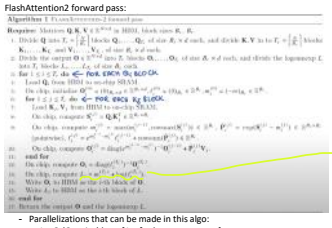
Pointer to 1st element of vector 'x'

Pointer to 1st element of vector 'y'

Pointer to 1st element of vector 'output'

The Block_ID (CUDA) equivalent for triton is program_id

FlashAttention2 forward pass:



During backward pass, we need to softmax AGAIN, for that we need:
1. Normalization Factor
2. Max over rows term

As we don't want to compute these AGAIN, we save both terms during forward pass into L_i,
And use them during backward pass

- Parallelizations that can be made in this algo:
  1. Q (Queries) loop [1st for loop, outermost]:
     i. 1 kernel works on 1 Q iteration, and inside it works on all K iterations.
     ii. Next kernel works on next Q iteration, and inside it on all K iterations.
     iii. And so on
  2. Above algo is for a single Head and a single Sequence:
     i. Can parallelize for each Sequence
     ii. Can parallelize for each Head in each Sequence (as each Sequence can have multiple Heads)
- Effectively, what we're doing:
  Parallelize each Sequence (in a Batch of Sequences)->
  inside each sequence, parallelize each Head ->
  inside each Head, parallelize each Query Block (Q)
- So, total # of triton programs (in triton, the name for kernel is program) running in parallel = BATCH_SIZE * NUM_OF_HEADS * (SEQ_LEN / BLOCK_SIZE_Q)

Go over Attention paper to clarify dimensions for all

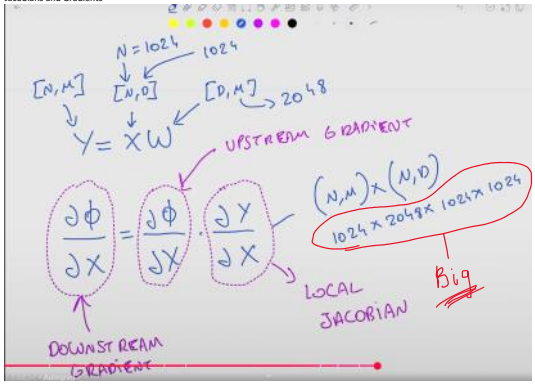
PyTorch operations:
- Always derived from torch.autograd.Function
- Operations like Softmax, ReLU, etc. are always implemented as torch.autograd.Function
- The operations should provide two methods:
  1. forward(): for forward pass, should compute Outputs of the operation
  2. backward(): for backward pass, should compute gradients of Loss wrt Inputs of the operation

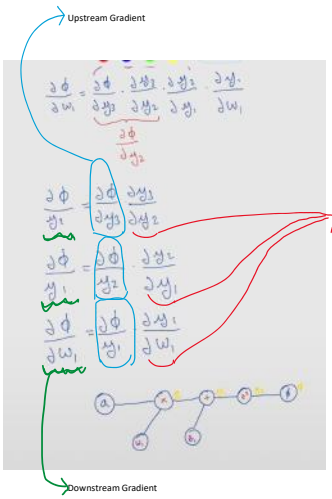
Causal vs Non-causal Attention:
- Causal (Masked):
  - Token can ONLY attend to itself and tokens BEFORE it in the sequence, NOT any tokens AFTER it
  - Prevents cheating in generative models (future tokens shouldn't inform current predictions)
  - Usage: Autoregressive settings (eg: language generation)
  - Math: For token at index 'i', attention to tokens at index 'j' > 'i' is zeroed out. Eg: if you have Q,K^T, you mask out the upper-right region (where j > i)
- Non-causal (Bidirectional):
  - Lets EVERY token attend to EVERY position in the sequence, including future ones
  - Usage: machine translation (encoder), BERT-style, not generating text one token at a time but learning bidirectional context
  - Math: no masking applied

Umar's implementation:
1. Not implemented in FP8, FP8 is faster actually on new GPUs
2. FlashAttention backward pass implemented for Causal AND Non-causal Attention, while Triton website only gives Causal
3. Explicit use of Softmax scale done where needed
4. Tips:
   a. If want to learn Triton optimizations: check Triton documentation

FlashAttention2 Backward pass:
What's needed:
1. Normalization Factor, Row Max (we already have this from the Forward Pass' 'm_i + log(l_i)' term)
2. PyTorch's Autograd
3. How to compute Gradients, Jacobians

Jacobians and Gradients

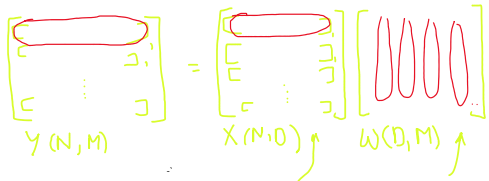




Algorithm 2 FlashAttention-2 Backward Pass

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V}, \mathbf{O}, \mathbf{dO} \in \mathbb{R}^{N \times d}$ in HBM, vector $L \in \mathbb{R}^N$ in HBM, block sizes B_c, B_r.

Divide $\mathbf{Q}$ into $T_r = \left\lceil \frac{N}{B_r} \right\rceil$ blocks $\mathbf{Q}_1, \dots, \mathbf{Q}_{T_r}$ of size $B_r \times d$ each, and divide $\mathbf{K}, \mathbf{V}$ in to $T_c = \left\lceil \frac{N}{B_c} \right\rceil$ blocks $\mathbf{K}_1, \dots, \mathbf{K}_{T_c}$ and $\mathbf{V}_1, \dots, \mathbf{V}_{T_c}$, of size $B_c \times d$ each.
Divide $\mathbf{O}$ into T_r blocks $\mathbf{O}_1, \dots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, divide $\mathbf{dO}$ into T_r blocks $\mathbf{dO}_1, \dots, \mathbf{dO}_{T_r}$ of size $B_r \times d$ each, and divide L into T_r blocks $L_1, \dots, L_{T_r}$ of size B_r each.
Initialize $\mathbf{dQ} = (0)_{N \times d}$ in HBM and divide it into T_r blocks $\mathbf{dQ}_1, \dots, \mathbf{dQ}_{T_r}$ of size $B_r \times d$ each. Divide $\mathbf{dK}, \mathbf{dV} \in \mathbb{R}^{N \times d}$ in to T_c blocks $\mathbf{dK}_1, \dots, \mathbf{dK}_{T_c}$ and $\mathbf{dV}_1, \dots, \mathbf{dV}_{T_c}$, of size $B_c \times d$ each.
Compute $D = \text{rowsum}(\mathbf{dO} \circ \mathbf{O}) \in \mathbb{R}^d$ (pointwise multiply), write D to HBM and divide it into T_r blocks $D_1, \dots, D_{T_r}$ of size B_r each.
for $1 \le j \le T_c$ **do**
  Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.
  Initialize $\mathbf{dK}_j = (0)_{B_c \times d}, \mathbf{dV}_j = (0)_{B_c \times d}$ on SRAM.
  for $1 \le i \le T_r$ **do**
    Load $\mathbf{Q}_i, \mathbf{O}_i, \mathbf{dO}_i, \mathbf{dQ}_i, L_i, D_i$ from HBM to on-chip SRAM.
    On chip, compute $\mathbf{S}_{ij}^{(j)} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.
    On chip, compute $\mathbf{P}_{ij}^{(j)} = \exp(\mathbf{S}_{ij} - L_i) \in \mathbb{R}^{B_r \times B_c}$.
    On chip, compute $\mathbf{dV}_j \leftarrow \mathbf{dV}_j + (\mathbf{P}_{ij}^{(j)})^T \mathbf{dO}_i \in \mathbb{R}^{B_c \times d}$.
    On chip, compute $\mathbf{dP}_{ij}^{(j)} = \mathbf{dO}_i \mathbf{V}_j^T \in \mathbb{R}^{B_r \times B_c}$.
    On chip, compute $\mathbf{dS}_{ij}^{(j)} = \mathbf{P}_{ij}^{(j)} \circ (\mathbf{dP}_{ij}^{(j)} - D_i) \in \mathbb{R}^{B_r \times B_c}$.
    Load $\mathbf{dQ}_i$ from HBM to SRAM, then on chip, update $\mathbf{dQ}_i \leftarrow \mathbf{dQ}_i + \mathbf{dS}_{ij}^{(j)} \mathbf{K}_j \in \mathbb{R}^{B_r \times d}$, and write back to HBM.
    On chip, compute $\mathbf{dK}_j \leftarrow \mathbf{dK}_j + \mathbf{dS}_{ij}^{(j)T} \mathbf{Q}_i \in \mathbb{R}^{B_c \times d}$.
  end for
  Write $\mathbf{dK}_j, \mathbf{dV}_j$ to HBM.
end for
Return $\mathbf{dQ}, \mathbf{dK}, \mathbf{dV}$.

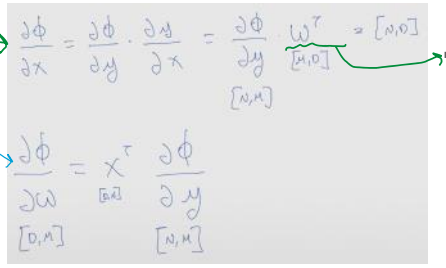


First row here is multiplied with every column here to make: First row of Y

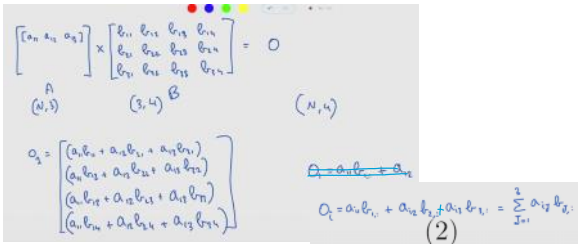
So, for Jacobian [dy/dx]:
- derivative of Y's First row with X' First row gives an output, but Y's First row with all other X's rows gives ZERO
- Hence, Jacobian turns out to be quite sparse

Also, Jacobian : can be very large, possible that it wont fit into GPU RAM

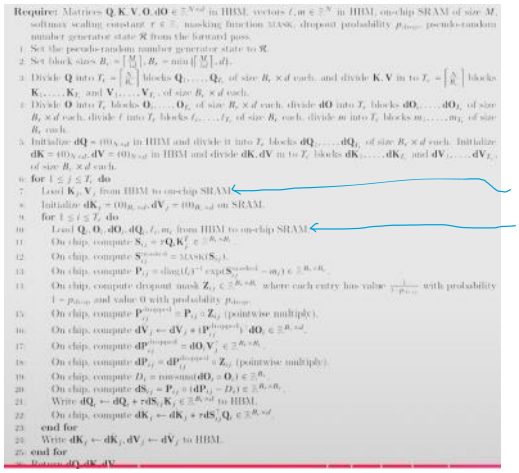
Can be shown (notebook) that:
Downstream Gradient wrt input = Upstream Gradient wrt output * Weight.T
Similarly,
Downstream Gradient wrt weight = Input.T * Upstream Gradient wrt output



In place of Jacobian



FlashAttention1 Backward Pass algorithm



Umar's Implementation:
Works with both Causal as well as Non-causal Attention

Outer Loop thru all K and V blocks

Inner loop thru all Q blocks

1. Inner for Loop split into 2 parts, as otherwise HBM needs to be written to at every Inner Loop iteration -> too costly
   Split like:
   - Each dq depends upon Loops over Ks:
     Fix Q block, loop over all KV blocks
   - Each dk depends upon Loops over Qs:
     Fix K block, loop over all Q blocks
2. To compute dq and dk vectors, need 'D_i':

$$dq_i = \sum_j dS_{ij} k_j = \sum_j P_{ij}(dP_{ij} - D_i)k_j = \sum_j \frac{e^{q_i^T k_j}}{L_i}(do_i^T v_j - D_i)k_j.$$

$$dk_j = \sum_i dS_{ij} q_i = \sum_i P_{ij}(dP_{ij} - D_i)q_i = \sum_i \frac{e^{q_i^T k_j}}{L_i}(do_i^T v_j - D_i)q_i.$$

where D_i is:

$$D_i = P_{i:}^T dP_{i:} = \sum_j \frac{e^{q_i^T k_j}}{L_i} do_i^T v_j = do_i^T \sum_j \frac{e^{q_i^T k_j}}{L_i} v_j = do_i^T o_i$$

So effectively, we:
- Calculate D_i
- Calculate dq
- Calculate dk
- Fix Q, loop over all KV
- Fix K, loop over all Q

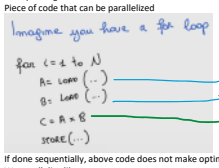
Triton Autotuning:
- Triton does Thread Coarscening for us, unlike CUDA, based on:
  - Block size
  - Number of Warps
- But, we need to give Triton like:

```
@triton.autotune(
    [
        triton.Config(
            {"BLOCK_SIZE_Q": BLOCK_SIZE_Q, "BLOCK_SIZE_KV": BLOCK_SIZE_KV},
            num_stages=num_stages,
            num_warps=num_warps,
        )
        for BLOCK_SIZE_Q in [64, 128]
        for BLOCK_SIZE_KV in [32, 64]
        for num_stages in [3, 4, 7]
        for num_warps in [2, 4]
    ],
    key=["SEQ_LEN", "HEAD_DIM"],
```

Have to try various configs for BLOCK_SIZE based on what works best (heuristic) based on timing

Triton runs a config for each pair of SEQ_LEN and HEAD_DIM, choosing the best throughput running in least amount of time

Software Pipelining:
- Piece of code that can be parallelized
-

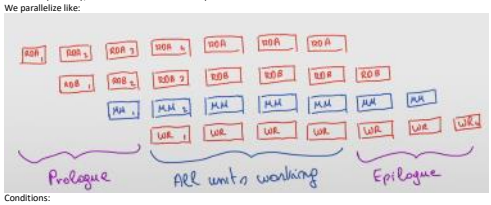


spawn

spawn another

Check if both spawns are done before doing matmul

- If done sequentially, above code does not make optimal use of GPU
- We parallelize like:



- Conditions:
  - Parallelization can be done using 'async' operations
  - More memory needed for this as SRAM needs to hold more memory:
    - When Compute Unit does MM1, at the same time step (3rd time step):
      - Reads for two matrices (RDA2 and RDB2) are already done
      - Reads for two matrices are half done already (RDA3)
  - Num of Iterations of for loop have to be MUCH GREATER THAN num of Stages of software pipeline
    (4 stages in example above: [RDA, RDB, MM, WR])
    # Iterations in loop >>> # stages in software pipeline, for pipelining to work well

==========

Credit:

1. hkproj/triton-flash-attention: https://github.com/hkproj/triton-flash-attention
2. Tri Dao et. al (2022) FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. arXiv:2205.14135
3. Tri Dao (2023) FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. arXiv:2307.08691

==========