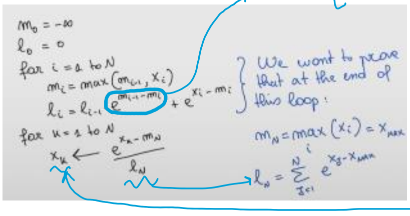
FlashAttention in Triton

Tuesday, March 18, 2025 5:17 PM

Flash Attention 1:
- Computations are on GPU
- Accessing HBM (GPU Global memory) or CPU DRAM is MUCH slower than accessing GPU Shared Memory
- Problem: Calculating Attention is I/O Bound -> as it keeps needing to access GPU Global Memory
- Solution: Computing Attention INSIDE Shared Memory itself, avoid repeated Global Memory accesses
- Consequence: Computation of Attention is MUCH CLOSER to GPU cores, Bandwidth bottleneck addressed
- 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):



$m_0 = -\infty$
$\ell_0 = 0$
for $i = 1$ to N
    $m_i = \max(m_{i-1}, x_i)$
    $\ell_i = \ell_{i-1} \boxed{e^{m_{i-1} - m_i}} + e^{x_i - m_i}$
for $k = 1$ to N
    $x_k \leftarrow \frac{e^{x_k - m_N}}{\ell_N}$

We want to prove that at the end of this loop:

$m_N = \max_i (x_i) = x_{max}$

$\ell_N = \sum_{j=1}^{N} e^{x_j - x_{max}}$

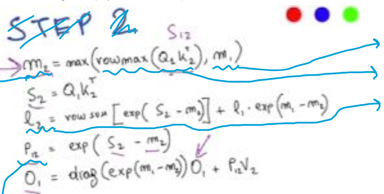
Correction Factor

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

Note on Local Memory and Global Memory:
- 'Local', 'Global' in scope and present in Device Memory (not actually physically partitioned in Device Memory)

new max

Online Softmax algorithm (implemented in code):



~~STEP 2~~

S_{12}

$m_2 = \max(\text{rowmax}(Q_1 K_2^T), m_1)$

$S_2 = Q_1 K_2^T$

$\ell_2 = \text{rowsum}[\exp(S_2 - m_2)] + \ell_1 \cdot \exp(m_1 - m_2)$

$P_{12} = \exp(S_2 - m_2)$

$O_1 = \text{diag}(\exp(m_1 - m_2)) O_1 + P_{12} V_2$

previous max

Current row max

Accumulate Normalization Factor terms at each step
We divide by the Accumulated Normalization Factor at the end to get the final Output

Update Output at each step. For 1st step, output = PV (only PV)

Applying Softmax_Star() on each Q.K^T

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

CUDA Mem Model maps to the Nvidia GPU Mem Hierarchy

Nvidia GPU Memory Hierarchy:

On-Chip

Registers [Fastest]
V
L1 Data Cache + Shared Memory
V
L2 Cache
V
Device Memory (HBM) [Slowest]

Off-Chip

CUDA Memory Model (how CUDA perceives GPU memory):

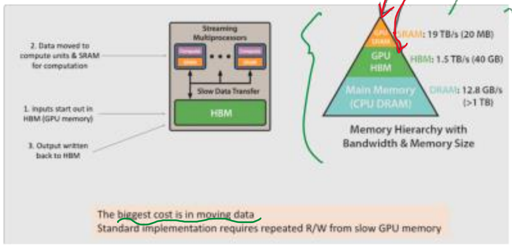
Registers & Local Memory (Private to its corresponding Thread)
V
Shared Memory (Every Thread in the Thread Block has access)
V
Global Memory (Every Thread in all Grids have access)

goes in

goes in

On-chip Shared Memory (is user-controlled L1 Cache, only Threads within the SAME Thread Block can access)

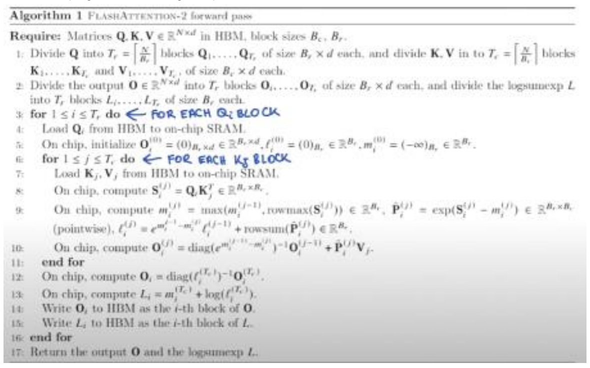
Off-chip Global Memory (shared across multiple Thread Blocks)



Streaming Multiprocessors
Compute
SRAM
Slow Data Transfer
HBM
2. Data moved to compute units & SRAM for computation
1. Inputs start out in HBM (GPU memory)
3. Output written back to HBM
GPU SRAM
SRAM: 19 TB/s (20 MB)
GPU HBM
HBM: 1.5 TB/s (40 GB)
Main Memory (CPU DRAM)
DRAM: 12.8 GB/s (>1 TB)
Memory Hierarchy with Bandwidth & Memory Size

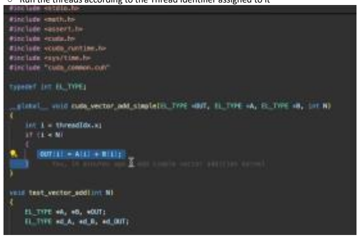
The biggest cost is in moving data
Standard implementation requires repeated R/W from slow GPU memory

FlashAttention2 (using the Online Softmax algo above)



```
Algorithm 1 FlashAttention-2 forward pass
Require: Matrices $Q, K, V \in \mathbb{R}^{N \times d}$ in HBM, block sizes $B_c$, $B_r$.
1: Divide $Q$ into $T_r = \lceil \frac{N}{B_r} \rceil$ blocks $Q_1, \dots, Q_{T_r}$ of size $B_r \times d$ each, and divide $K, V$ in to $T_c = \lceil \frac{N}{B_c} \rceil$ blocks
   $K_1, \dots, K_{T_c}$ and $V_1, \dots, V_{T_c}$, of size $B_c \times d$ each.
2: Divide the output $O \in \mathbb{R}^{N \times d}$ into $T_r$ blocks $O_i, \dots, O_{T_r}$ 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 $Q_i$ from HBM to on-chip SRAM.
5:     On chip, initialize $O_i^{(0)} = (0)_{B_r \times d} \in \mathbb{R}^{B_r \times d}$, $\ell_i^{(0)} = (0)_{B_r} \in \mathbb{R}^{B_r}$, $m_i^{(0)} = (-\infty)_{B_r} \in \mathbb{R}^{B_r}$.
6:     for $1 \le j \le T_c$ do  ← FOR EACH $K_j$ BLOCK
7:         Load $K_j, V_j$ from HBM to on-chip SRAM.
8:         On chip, compute $S_i^{(j)} = Q_i K_j^T \in \mathbb{R}^{B_r \times B_c}$.
9:         On chip, compute $m_i^{(j)} = \max(m_i^{(j-1)}, \text{rowmax}(S_i^{(j)})) \in \mathbb{R}^{B_r}$, $\tilde{P}_i^{(j)} = \exp(S_i^{(j)} - m_i^{(j)}) \in \mathbb{R}^{B_r \times B_c}$
           (pointwise), $\ell_i^{(j)} = e^{m_i^{(j-1)} - m_i^{(j)}} \ell_i^{(j-1)} + \text{rowsum}(\tilde{P}_i^{(j)}) \in \mathbb{R}^{B_r}$.
10:        On chip, compute $O_i^{(j)} = \text{diag}(e^{m_i^{(j-1)} - m_i^{(j)}})^{-1} O_i^{(j-1)} + \tilde{P}_i^{(j)} V_j$.
11:    end for
12:    On chip, compute $O_i = \text{diag}(\ell_i^{(T_c)})^{-1} O_i^{(T_c)}$.
13:    On chip, compute $L_i = m_i^{(T_c)} + \log(\ell_i^{(T_c)})$.
14:    Write $O_i$ to HBM as the $i$-th block of $O$.
15:    Write $L_i$ to HBM as the $i$-th block of $L$.
16: end for
17: Return the output $O$ and the logsumexp $L$.
```

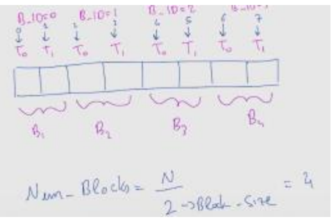
CUDA:
- We have to tell each Thread what to do, ie software engineer determines mapping between Thread and its Data Elements (CUDA knows only # of Blocks & # of Threads [in each Block])
- When running 'N'-threads in parallel (done on each core), when we ask CUDA to run the N-threads in parallel, it will: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



```c
#include <stdio.h>
#include <math.h>
#include <assert.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include <sys/time.h>
#include "cuda_common.cuh"

typedef int EL_TYPE;

__global__ void cuda_vector_add_simple(EL_TYPE *OUT, EL_TYPE *A, EL_TYPE *B, int N)
{
    int i = threadIdx.x;
    if (i < N)
    {
        OUT[i] = A[i] + B[i];
    }
}

void test_vector_add(int N)
{
    EL_TYPE *A, *B, *OUT;
    EL_TYPE *d_A, *d_B, *d_OUT;
```

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

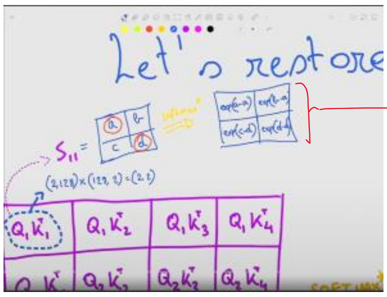


Eg: N = 8, Block Size = 2, GPU has 4 cores
So, # of Blocks = 8 / 2 = 4

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



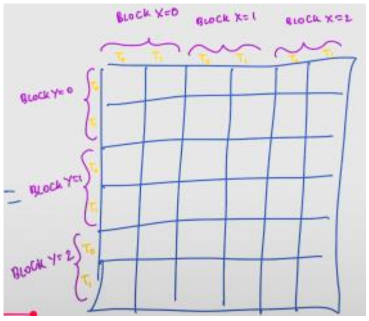
Let's restore
S_{11} =
a b c d
exp(a-a) exp(b-a) exp(c-d) exp(d-d)
(2,128)×(128,2)=(2,2)
$Q_1K_1^T$ $Q_1K_2^T$ $Q_1K_3^T$ $Q_1K_4^T$
$Q_2K_1^T$ $Q_2K_2^T$ $Q_2K_3^T$ $Q_2K_4^T$

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

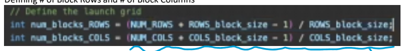
General formula to calculate # of Blocks:
# of Blocks = Total # of Data Elements (N) / Block Size

# 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



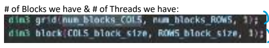
```c
// Define the launch grid
int num_blocks_ROWS = (NUM_ROWS + ROWS_block_size - 1) / ROWS_block_size;
int num_blocks_COLS = (NUM_COLS + COLS_block_size - 1) / COLS_block_size;
```

# 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)

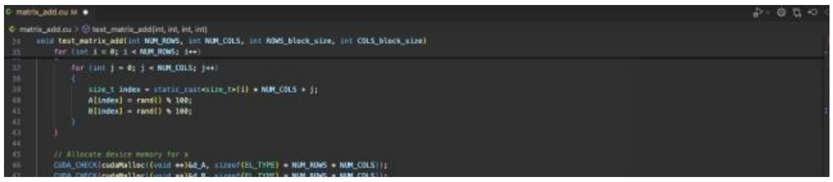
- # of Blocks we have & # of Threads we have:



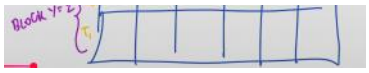
```c
dim3 grid(num_blocks_COLS, num_blocks_ROWS, 1);
dim3 block(COLS_block_size, ROWS_block_size, 1);
```

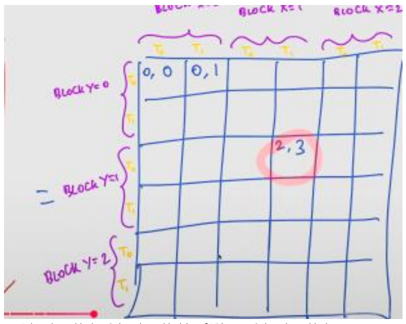
# of Blocks

# of Threads

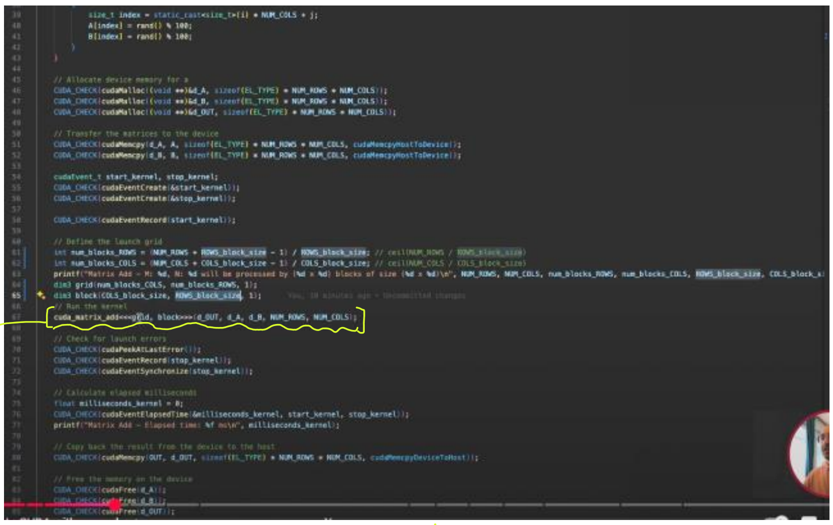


```c
matrix_add.cu > test_matrix_add(int, int, int, int)
void test_matrix_add(int NUM_ROWS, int NUM_COLS, int ROWS_block_size, int COLS_block_size)
    for (int i = 0; i < NUM_ROWS; i++)
    {
        for (int j = 0; j < NUM_COLS; j++)
        {
            size_t index = static_cast<size_t>(i) * NUM_COLS + j;
            A[index] = rand() % 100;
            B[index] = rand() % 100;
        }
    }

    // Allocate device memory for a
    CUDA_CHECK(cudaMalloc((void **)&d_A, sizeof(EL_TYPE) * NUM_ROWS * NUM_COLS));
```





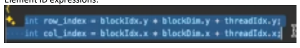
4 threads per block or 2 threads per block here? Video says 2 threads per block



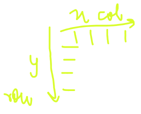
```
            size_t index = static_cast<size_t>(i) * NUM_COLS + j;
            A[index] = rand() % 100;
            B[index] = rand() % 100;
        }
    }

    // Allocate device memory for a
    CUDA_CHECK(cudaMalloc((void **)&d_A, sizeof(EL_TYPE) * NUM_ROWS * NUM_COLS));
    CUDA_CHECK(cudaMalloc((void **)&d_B, sizeof(EL_TYPE) * NUM_ROWS * NUM_COLS));
    CUDA_CHECK(cudaMalloc((void **)&d_OUT, sizeof(EL_TYPE) * NUM_ROWS * NUM_COLS));

    // Transfer the matrices to the device
    CUDA_CHECK(cudaMemcpy(d_A, A, sizeof(EL_TYPE) * NUM_ROWS * NUM_COLS, cudaMemcpyHostToDevice));
    CUDA_CHECK(cudaMemcpy(d_B, B, sizeof(EL_TYPE) * NUM_ROWS * NUM_COLS, cudaMemcpyHostToDevice));

    cudaEvent_t start_kernel, stop_kernel;
    CUDA_CHECK(cudaEventCreate(&start_kernel));
    CUDA_CHECK(cudaEventCreate(&stop_kernel));

    CUDA_CHECK(cudaEventRecord(start_kernel));

    // Define the launch grid
    int num_blocks_ROWS = (NUM_ROWS + ROWS_block_size - 1) / ROWS_block_size; // ceil(NUM_ROWS / ROWS_block_size)
    int num_blocks_COLS = (NUM_COLS + COLS_block_size - 1) / COLS_block_size; // ceil(NUM_COLS / COLS_block_size)
    printf("Matrix Add - M: %d, N: %d will be processed by (%d x %d) blocks of size (%d x %d)\n", NUM_ROWS, NUM_COLS, num_blocks_ROWS, num_blocks_COLS, ROWS_block_size, COLS_block_s
    dim3 grid(num_blocks_COLS, num_blocks_ROWS, 1);
    dim3 block(COLS_block_size, ROWS_block_size, 1);        You, 10 minutes ago • Uncommitted changes
    // Run the kernel
    cuda_matrix_add<<<grid, block>>>(d_OUT, d_A, d_B, NUM_ROWS, NUM_COLS);

    // Check for launch errors
    CUDA_CHECK(cudaPeekAtLastError());
    CUDA_CHECK(cudaEventRecord(stop_kernel));
    CUDA_CHECK(cudaEventSynchronize(stop_kernel));

    // Calculate elapsed milliseconds
    float milliseconds_kernel = 0;
    CUDA_CHECK(cudaEventElapsedTime(&milliseconds_kernel, start_kernel, stop_kernel));
    printf("Matrix Add - Elapsed time: %f ms\n", milliseconds_kernel);

    // Copy back the result from the device to the host
    CUDA_CHECK(cudaMemcpy(OUT, d_OUT, sizeof(EL_TYPE) * NUM_ROWS * NUM_COLS, cudaMemcpyDeviceToHost));

    // Free the memory on the device
    CUDA_CHECK(cudaFree(d_A));
    CUDA_CHECK(cudaFree(d_B));
    CUDA_CHECK(cudaFree(d_OUT));
```

Defines the grid

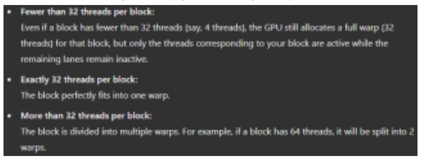
- Element ID expressions:



```
    int row_index = blockIdx.y * blockDim.y + threadIdx.y;
    int col_index = blockIdx.x * blockDim.x + threadIdx.x;
```

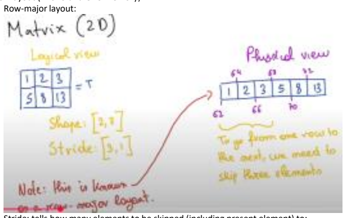


Relationshio between warp and block (depends on threads per block)



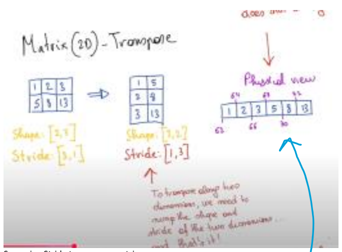
- **Fewer than 32 threads per block:**
  Even if a block has fewer than 32 threads (say, 4 threads), the GPU still allocates a full warp (32 threads) for that block, but only the threads corresponding to your block are active while the remaining lanes remain inactive.
- **Exactly 32 threads per block:**
  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.

Tensor Layout (in CPU and GPU memory):
- Row-major layout:

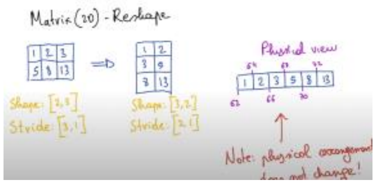


  Stride: tells how many elements to be skipped (including present element) to:
  - Go to the next row dimension (eg: 3 here)
  - Go to the next column dimension (eg: 1 here)

  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

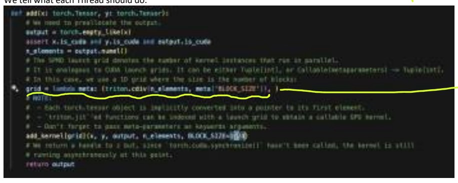


Tensor Reshaping

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

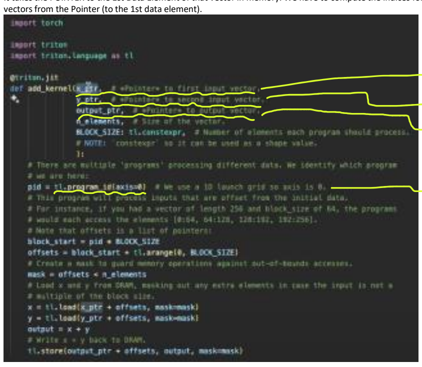


```
def add(x: torch.Tensor, y: torch.Tensor):
    # We need to preallocate the output.
    output = torch.empty_like(x)
    assert x.is_cuda and y.is_cuda and output.is_cuda
    n_elements = output.numel()
    # The SPMD launch grid denotes the number of kernel instances that run in parallel.
    # It is analogous to CUDA launch grids. It can be either Tuple[int], or Callable(metaparameters) -> Tuple[int].
    # In this case, we use a 1D grid where the size is the number of blocks:
    grid = lambda meta: (triton.cdiv(n_elements, meta['BLOCK_SIZE']), )
    # NOTE:
    #  - Each torch.tensor object is implicitly converted into a pointer to its first element.
    #  - `triton.jit`'ed functions can be indexed with a launch grid to obtain a callable GPU kernel.
    #  - Don't forget to pass meta-parameters as keywords arguments.
    add_kernel[grid](x, y, output, n_elements, BLOCK_SIZE=1024)
    # We return a handle to z but, since `torch.cuda.synchronize()` hasn't been called, the kernel is still
    # running asynchronously at this point.
    return output
```

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).



```
import torch

import triton
import triton.language as tl

@triton.jit
def add_kernel(x_ptr,  # *Pointer* to first input vector.
               y_ptr,  # *Pointer* to second input vector.
               output_ptr,  # *Pointer* to output vector.
               n_elements,  # Size of the vector.
               BLOCK_SIZE: tl.constexpr,  # Number of elements each program should process.
               # NOTE: `constexpr` so it can be used as a shape value.
               ):
    # There are multiple 'programs' processing different data. We identify which program
    # we are here:
    pid = tl.program_id(axis=0)  # We use a 1D launch grid so axis is 0.
    # This program will process inputs that are offset from the initial data.
    # For instance, if you had a vector of length 256 and block_size of 64, the programs
    # would each access the elements [0:64, 64:128, 128:192, 192:256].
    # Note that offsets is a list of pointers:
    block_start = pid * BLOCK_SIZE
    offsets = block_start + tl.arange(0, BLOCK_SIZE)
    # Create a mask to guard memory operations against out-of-bounds accesses.
    mask = offsets < n_elements
    # Load x and y from DRAM, masking out any extra elements in case the input is not a
    # multiple of the block size.
    x = tl.load(x_ptr + offsets, mask=mask)
    y = tl.load(y_ptr + offsets, mask=mask)
    output = x + y
    # Write x + y back to DRAM.
    tl.store(output_ptr + offsets, output, mask=mask)
```

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

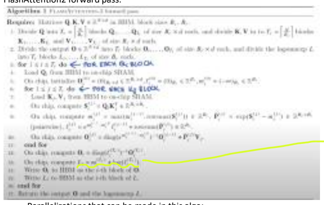
FlashAttention2 forward pass:

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 $\mathbf{Q}_1, \ldots, \mathbf{Q}_{T_r}$ 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 $\mathbf{K}_1, \ldots, \mathbf{K}_{T_c}$ and $\mathbf{V}_1, \ldots, \mathbf{V}_{T_c}$, of size $B_c \times d$ each.

PyTorch operations:
- Always derived from torch.autograd.Function
- Operations like Softmax, ReLU, etc. are always Implemented as torch.autograd.Function

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

FlashAttention2 forward pass:



```
Algorithm 1 FlashAttention-2 forward pass
Require: Matrices Q, K, V ∈ R^{N×d} in HBM, block sizes B_c, B_r.
 1: Divide Q into T_r = ⌈N/B_r⌉ blocks Q_1, ..., Q_{T_r} of size B_r × d each, and divide K, V in to T_c = ⌈N/B_c⌉ blocks
    K_1, ..., K_{T_c} and V_1, ..., V_{T_c}, of size B_c × d each.
 2: Divide the output O ∈ R^{N×d} into T_r blocks O_i, ..., O_{T_r} of size B_r × d each, and divide the logsumexp L
    into T_r blocks L_i, ..., L_{T_r} of size B_r each.
 3: for 1 ≤ i ≤ T_r do ← FOR EACH Q_i BLOCK
 4:   Load Q_i from HBM to on-chip SRAM.
 5:   On chip, initialize O_i^(0) = (0)_{B_r×d} ∈ R^{B_r×d}, ℓ_i^(0) = (0)_{B_r} ∈ R^{B_r}, m_i^(0) = (−∞)_{B_r} ∈ R^{B_r}.
 6:   for 1 ≤ j ≤ T_c do ← FOR EACH K_j BLOCK
 7:     Load K_j, V_j from HBM to on-chip SRAM.
 8:     On chip, compute S_i^(j) = Q_i K_j^T ∈ R^{B_r×B_c}.
 9:     On chip, compute m_i^(j) = max(m_i^(j−1), rowmax(S_i^(j))) ∈ R^{B_r}, P̃_i^(j) = exp(S_i^(j) − m_i^(j)) ∈ R^{B_r×B_c}
        (pointwise), ℓ_i^(j) = e^{m_i^(j−1) − m_i^(j)} ℓ_i^(j−1) + rowsum(P̃_i^(j)) ∈ R^{B_r}.
10:     On chip, compute O_i^(j) = diag(e^{m_i^(j−1) − m_i^(j)})^{−1} O_i^(j−1) + P̃_i^(j) V_j.
11:   end for
12:   On chip, compute O_i = diag(ℓ_i^(T_c))^{−1} O_i^(T_c).
13:   On chip, compute L_i = m_i^(T_c) + log(ℓ_i^(T_c)).
14:   Write O_i to HBM as the i-th block of O.
15:   Write L_i to HBM as the i-th block of L.
16: end for
17: Return the output O and the logsumexp L.
```

- 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)

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

Go over Attention paper to clarify dimensions for all

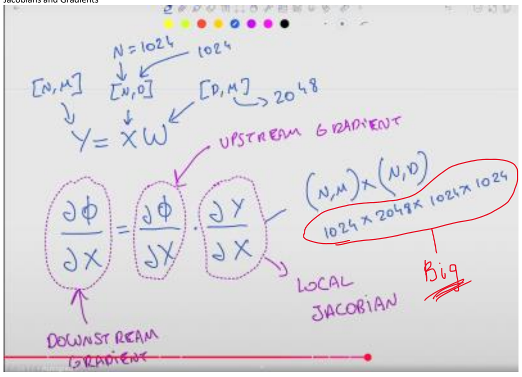
PyTorch operations:
- Always derived from torch.autograd.Function
- Operations like Softmax, ReLU, etc. are always Implemented as torch.autograd.Function
- The operations should provide two methods:
    1. forward(): for forward pass, should compute Outputs of the operation
    2. backward(): for backward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

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

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

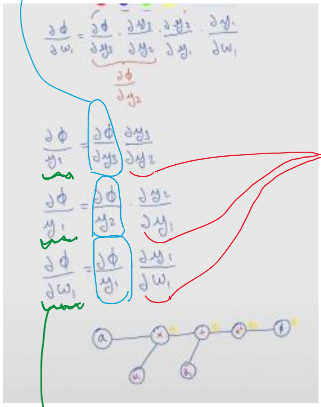
FlashAttention2 Backward pass:
What's needed:
1. Normalization Factor, Row Max (we already have this from the Forward Pass' 'm_i + log(l_i)' term)
2. PyTorch's Autograd
3. How to compute Gradients, Jacobians

Jacobians and Gradients



$$\frac{\partial \phi}{\partial X} = \frac{\partial \phi}{\partial Y} \cdot \frac{\partial Y}{\partial X}$$

Upstream Gradient



$$\frac{\partial \phi}{\partial w_1} = \underbrace{\frac{\partial \phi}{\partial y_3} \cdot \frac{\partial y_3}{\partial y_2}}_{\frac{\partial \phi}{\partial y_2}} \cdot \frac{\partial y_2}{\partial y_1} \cdot \frac{\partial y_1}{\partial w_1}$$

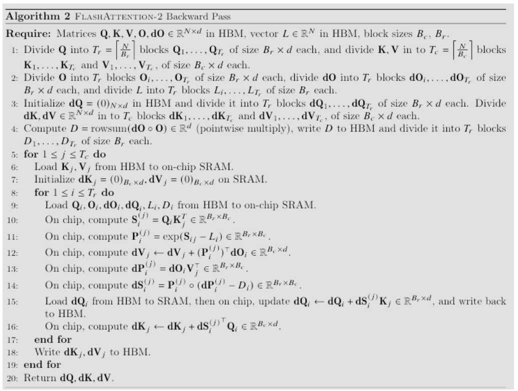
$$\frac{\partial \phi}{\partial y_2} = \frac{\partial \phi}{\partial y_3} \cdot \frac{\partial y_3}{\partial y_2}$$

$$\frac{\partial \phi}{\partial y_1} = \frac{\partial \phi}{\partial y_2} \cdot \frac{\partial y_2}{\partial y_1}$$

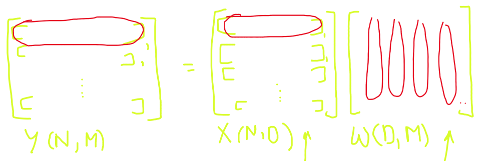
$$\frac{\partial \phi}{\partial w_1} = \frac{\partial \phi}{\partial y_1} \cdot \frac{\partial y_1}{\partial w_1}$$

Local Jacobian

Downstream Gradient



```
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 $B_c$, $B_r$.
 1: Divide $\mathbf{Q}$ into $T_r = \lceil \frac{N}{B_r} \rceil$ blocks $\mathbf{Q}_1, \dots, \mathbf{Q}_{T_r}$ of size $B_r \times d$ each, and divide $\mathbf{K}, \mathbf{V}$ in to $T_c = \lceil \frac{N}{B_c} \rceil$ 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 $\mathbf{O}$ into $T_r$ blocks $\mathbf{O}_i, \dots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, divide $\mathbf{dO}$ into $T_r$ blocks $\mathbf{dO}_i, \dots, \mathbf{dO}_{T_r}$ of size
    $B_r \times d$ each, and divide $L$ into $T_r$ blocks $L_i, \dots, L_{T_r}$ of size $B_r$ each.
 3: 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.
 4: Compute $D = \mathrm{rowsum}(\mathbf{dO} \circ \mathbf{O}) \in \mathbb{R}^d$ (pointwise multiply), write $D$ to HBM and divide it into $T_r$ blocks
    $D_1, \dots, D_{T_r}$ of size $B_r$ each.
 5: for $1 \le j \le T_c$ do
 6:   Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.
 7:   Initialize $\mathbf{dK}_j = (0)_{B_c\times d}, \mathbf{dV}_j = (0)_{B_c\times d}$ on SRAM.
 8:   for $1 \le i \le T_r$ do
 9:     Load $\mathbf{Q}_i, \mathbf{O}_i, \mathbf{dO}_i, \mathbf{dQ}_i, L_i, D_i$ from HBM to on-chip SRAM.
10:     On chip, compute $\mathbf{S}_i^{(j)} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r\times B_c}$.
11:     On chip, compute $\mathbf{P}_i^{(j)} = \exp(\mathbf{S}_{ij} - L_i) \in \mathbb{R}^{B_r\times B_c}$.
12:     On chip, compute $\mathbf{dV}_j \leftarrow \mathbf{dV}_j + (\mathbf{P}_i^{(j)})^\top \mathbf{dO}_i \in \mathbb{R}^{B_c\times d}$.
13:     On chip, compute $\mathbf{dP}_i^{(j)} = \mathbf{dO}_i \mathbf{V}_j^\top \in \mathbb{R}^{B_r\times B_c}$.
14:     On chip, compute $\mathbf{dS}_i^{(j)} = \mathbf{P}_i^{(j)} \circ (\mathbf{dP}_i^{(j)} - D_i) \in \mathbb{R}^{B_r\times B_c}$.
15:     Load $\mathbf{dQ}_i$ from HBM to SRAM, then on chip, update $\mathbf{dQ}_i \leftarrow \mathbf{dQ}_i + \mathbf{dS}_i^{(j)} \mathbf{K}_j \in \mathbb{R}^{B_r\times d}$, and write back
        to HBM.
16:     On chip, compute $\mathbf{dK}_j \leftarrow \mathbf{dK}_j + {\mathbf{dS}_i^{(j)}}^\top \mathbf{Q}_i \in \mathbb{R}^{B_c\times d}$.
17:   end for
18:   Write $\mathbf{dK}_j, \mathbf{dV}_j$ to HBM.
19: end for
20: Return $\mathbf{dQ}, \mathbf{dK}, \mathbf{dV}$.
```

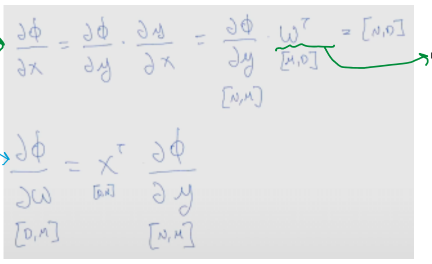


First row here is multiplied with every column here to make: First row of Y

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

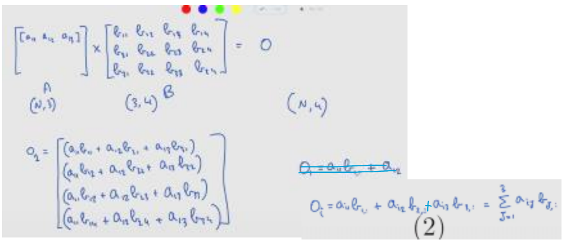
Can be shown (notebook) that:
Downstream Gradient wrt input = Upstream Gradient wrt output * Weight.T
Similarly,
Downstream Gradient wrt weight = Input.T * Upstream Gradient wrt output



$$\frac{\partial \phi}{\partial x} = \frac{\partial \phi}{\partial y} \cdot \frac{\partial y}{\partial x} = \underset{[N,M]}{\frac{\partial \phi}{\partial y}} \cdot \underset{[M,D]}{W^T} = [N,D]$$

In place of Jacobian

$$\underset{[D,M]}{\frac{\partial \phi}{\partial W}} = \underset{[D,N]}{X^T} \, \underset{[N,M]}{\frac{\partial \phi}{\partial y}}$$



$$\underset{(N,3)}{A} \begin{bmatrix} [a_{11} & a_{12} & a_{13}] \\ & & \end{bmatrix} \times \underset{(3,4)}{B} \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \end{bmatrix} = \underset{(N,4)}{O}$$

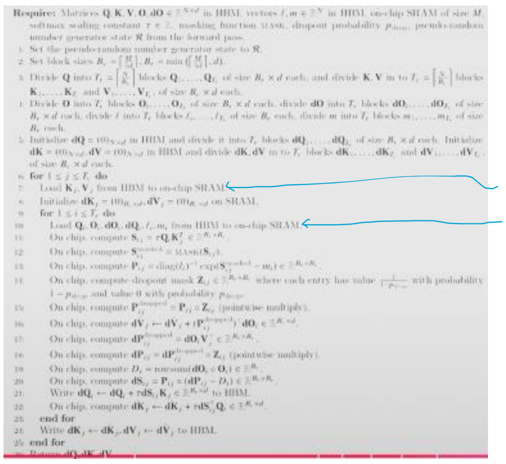
$$O_1 = \begin{bmatrix} (a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31}) \\ (a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32}) \\ (a_{11}b_{13} + a_{12}b_{23} + a_{13}b_{33}) \\ (a_{11}b_{14} + a_{12}b_{24} + a_{13}b_{34}) \end{bmatrix}$$

~~$O_1 = a_{11}b_{1:} + a_{12}$~~

$$O_i = a_{i1}b_{1,:} + a_{i2}b_{2,:} + a_{i3}b_{3,:} = \sum_{j=1}^{3} a_{ij} b_{j,:} \qquad (2)$$

FlashAttention1 Backward Pass algorithm

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



```
Require: Matrices $\mathbf{Q},\mathbf{K},\mathbf{V},\mathbf{O},\mathbf{dO} \in \mathbb{R}^{N\times d}$ in HBM, vectors $\ell, m \in \mathbb{R}^N$ in HBM, on-chip SRAM of size $M$,
    softmax scaling constant $\tau \in \mathbb{R}$, masking function MASK, dropout probability $p_{drop}$, pseudo-random
    number generator state $\mathcal{R}$ from the forward pass.
1: Set the pseudo-random number generator state to $\mathcal{R}$.
2: Set block sizes $B_c = \lceil \frac{M}{4d} \rceil$, $B_r = \min(\lceil \frac{M}{4d} \rceil, d)$.
3: Divide $\mathbf{Q}$ into $T_r = \lceil \frac{N}{B_r} \rceil$ blocks $\mathbf{Q}_1,\dots,\mathbf{Q}_{T_r}$ of size $B_r \times d$ each, and divide $\mathbf{K},\mathbf{V}$ in to $T_c = \lceil \frac{N}{B_c} \rceil$ 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.
4: Divide $\mathbf{O}$ into $T_r$ blocks $\mathbf{O}_i,\dots,\mathbf{O}_{T_r}$ of size $B_r \times d$ each, divide $\mathbf{dO}$ into $T_r$ blocks $\mathbf{dO}_i,\dots,\mathbf{dO}_{T_r}$ of size
    $B_r \times d$ each, divide $\ell$ into $T_r$ blocks $\ell_i,\dots,\ell_{T_r}$ of size $B_r$ each, divide $m$ into $T_r$ blocks $m_1,\dots,m_{T_r}$ of size
    $B_r$ each.
5: 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. Initialize
    $\mathbf{dK} = (0)_{N\times d}, \mathbf{dV} = (0)_{N\times d}$ in HBM and divide $\mathbf{dK},\mathbf{dV}$ 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.
6: for $1 \le j \le T_c$ do
7:   Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.
8:   Initialize $\tilde{\mathbf{dK}}_j = (0)_{B_c\times d}, \tilde{\mathbf{dV}}_j = (0)_{B_c\times d}$ on SRAM.
9:   for $1 \le i \le T_r$ do
10:    Load $\mathbf{Q}_i, \mathbf{O}_i, \mathbf{dO}_i, \mathbf{dQ}_i, \ell_i, m_i$ from HBM to on-chip SRAM.
11:    On chip, compute $\mathbf{S}_{ij} = \tau\mathbf{Q}_i\mathbf{K}_j^T \in \mathbb{R}^{B_r\times B_c}$.
12:    On chip, compute $\mathbf{S}_{ij}^{masked} = \text{MASK}(\mathbf{S}_{ij})$.
13:    On chip, compute $\mathbf{P}_{ij} = \text{diag}(\ell_i)^{-1}\exp(\mathbf{S}_{ij}^{masked} - m_i) \in \mathbb{R}^{B_r\times B_c}$.
14:    On chip, compute dropout mask $\mathbf{Z}_{ij} \in \mathbb{R}^{B_r\times B_c}$ where each entry has value $\frac{1}{1-p_{drop}}$ with probability
       $1 - p_{drop}$ and value 0 with probability $p_{drop}$.
15:    On chip, compute $\mathbf{P}_{ij}^{dropped} = \mathbf{P}_{ij} \circ \mathbf{Z}_{ij}$ (pointwise multiply).
16:    On chip, compute $\tilde{\mathbf{dV}}_j \leftarrow \tilde{\mathbf{dV}}_j + (\mathbf{P}_{ij}^{dropped})^T\mathbf{dO}_i \in \mathbb{R}^{B_c\times d}$.
17:    On chip, compute $\mathbf{dP}_{ij}^{dropped} = \mathbf{dO}_i\mathbf{V}_j^T \in \mathbb{R}^{B_r\times B_c}$.
18:    On chip, compute $\mathbf{dP}_{ij} = \mathbf{dP}_{ij}^{dropped} \circ \mathbf{Z}_{ij}$ (pointwise multiply).
19:    On chip, compute $D_i = \text{rowsum}(\mathbf{dO}_i \circ \mathbf{O}_i) \in \mathbb{R}^{B_r}$.
20:    On chip, compute $\mathbf{dS}_{ij} = \mathbf{P}_{ij} \circ (\mathbf{dP}_{ij} - D_i) \in \mathbb{R}^{B_r\times B_c}$.
21:    Write $\mathbf{dQ}_i \leftarrow \mathbf{dQ}_i + \tau\mathbf{dS}_{ij}\mathbf{K}_j \in \mathbb{R}^{B_r\times d}$ to HBM.
22:    On chip, compute $\tilde{\mathbf{dK}}_j \leftarrow \tilde{\mathbf{dK}}_j + \tau\mathbf{dS}_{ij}^T\mathbf{Q}_i \in \mathbb{R}^{B_c\times d}$.
23:  end for
24:  Write $\mathbf{dK}_j \leftarrow \tilde{\mathbf{dK}}_j, \mathbf{dV}_j \leftarrow \tilde{\mathbf{dV}}_j$ to HBM.
25: end for
26: Return $\mathbf{dQ}, \mathbf{dK}, \mathbf{dV}$.
```

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':

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

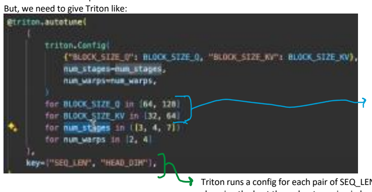
$$dk_j = \sum_i dS_{ij}q_i = \sum_i P_{ij}(dP_{ij} - D_i)q_i = \sum_i \frac{e^{q_i^T k_j}}{L_i}(do_i^T v_j - D_i)q_i.$$

   where D_i is:

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

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

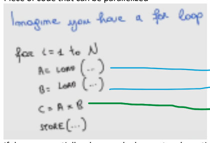
Triton Autotuning:
- Triton does Thread Coarscening for us, unlike CUDA, based on:
  - Block size
  - Number of Warps
- But, we need to give Triton like:



```
@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
- 

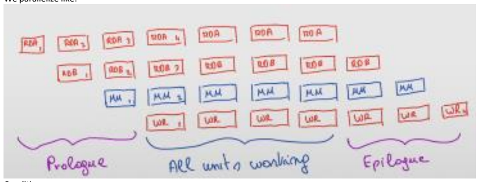
```
Imagine you have a for loop

for i = 1 to N
    A = LOAD (...)
    B = LOAD (...)
    C = A * B
    STORE (...)
```

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
4. Basics on NVIDIA GPU Hardware Architecture, NASA : https://www.nas.nasa.gov/hecc/support/kb/basics-on-nvidia-gpu-hardware-architecture_704.html#:~:text=Memory%20access%20latency%20is%20lower,227%20KB%20of%20shared%20memory, accessed 10/22
5. USC EE508 Hardware Foundations of ML Notes

==