

Tuesday, March 18, 2025 5:17 PM

- Computations are on GPU
- Accessing HBM (DRAM, Global, largest memory) is MUCH slower than accessing Shared Memory
- Problem: Calculating Attention is  $O(D^2)$  > 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 slower 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

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

Let's restore

$S_{11} =$

a	b
c	d

inverse

$\text{inv}(a)$	$\text{inv}(b)$
$\text{inv}(c)$	$\text{inv}(d)$

$(2, 12) \times (22, 2) = (2, 2)$

$Q_1, K_1$	$Q_1, K_2$	$Q_1, K_3$	$Q_1, K_4$
$Q_2, K_1$	$Q_2, K_2$	$Q_2, K_3$	$Q_2, K_4$

x\_max does over each row or over entire matrix?  
=> over each row

$m_0 = -\infty$   
 $l_0 = 0$   
 for  $i = 1$  to  $N$   
 $m_i = \max(m_{i-1}, x_i)$   
 $l_i = l_{i-1} + e^{m_{i-1} - m_i} + e^{x_i - m_i}$   
 for  $k = 1$  to  $N$   
 $x_k \leftarrow \frac{e^{x_k - m_N}}{l_N}$   
 $l_N = \sum_{j=1}^N e^{x_j - x_{\max}}$

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

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

**STEP 2**

new max  $\rightarrow m_1 = \max(\text{row}(\max(Q_{12} k_{12}), m_1))$  previous max

$S_2 = Q_{12} k_{12}$  Current row max

$l_2 = \text{row sum} [\exp(S_2 - m_1)] + l_1, \exp(m_1 - m_1)$  Accumulate Normalization Factor terms at each step

$P_1 = \frac{\exp(S_2 - m_1)}{\text{row sum} [\exp(S_2 - m_1)] + l_1}$  We divide by the Accumulated Normalization Factor at the end to get the final output

$O_1 = \text{diag}(\exp(m_1 - m_1)) O_1 + P_1 V_2$

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

Applying Softmax\_Star() on each  $Q \times T$

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

---

**Algorithm 1** FLASHATTENTION-2 forward pass

```

1: Require: Matrices  $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$  in HBM, block sizes  $B_r, B_c$ .
2: 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  $\mathbf{K}, \mathbf{V}$  into  $T_r = \lceil \frac{N}{B_r} \rceil$  blocks  $\mathbf{K}_1, \dots, \mathbf{K}_{T_r}$  and  $\mathbf{V}_1, \dots, \mathbf{V}_{T_r}$  of size  $B_r \times d$  each.
3: 2. Divide the output  $\mathbf{O} \in \mathbb{R}^{N \times d}$  into  $T_r$  blocks  $\mathbf{O}_1, \dots, \mathbf{O}_{T_r}$  of size  $B_r \times d$  each, and divide the logsumexp  $L$  into  $T_r$  blocks  $L_1, \dots, L_{T_r}$  of size  $B_r$ .
4: for  $1 \leq i \leq T_r$  do ← FOR EACH  $\mathbf{Q}_i$  BLOCK
5:   Load  $\mathbf{Q}_i$  from HBM to on-chip SRAM.
6:   On chip, initialize  $\mathbf{O}^{(i)} = (0)_{B_r \times d} \in \mathbb{R}^{B_r \times d}$ ,  $\mathbf{m}^{(i)} = (0)_{B_r} \in \mathbb{R}^{B_r}$ ,  $\mathbf{m}^{(i)} = (-\infty)_{B_r} \in \mathbb{R}^{B_r}$ .
7:   for  $1 \leq j \leq T_c$  do ← FOR EACH  $\mathbf{K}_j$  BLOCK
8:     Load  $\mathbf{K}_j, \mathbf{V}_j$  from HBM to on-chip SRAM.
9:     On chip, compute  $\mathbf{S}^{(ij)} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$ .
10:    On chip, compute  $\mathbf{m}_j^{(i)} = \max(\mathbf{m}^{(i,j-1)}, \text{rowmax}(\mathbf{S}^{(ij)})) \in \mathbb{R}^{B_r}$ ,  $\mathbf{P}_j^{(i)} = \exp(\mathbf{S}^{(ij)} - \mathbf{m}_j^{(i)}) \in \mathbb{R}^{B_r \times B_c}$  (pointwise),  $\ell_j^{(i)} = e^{\mathbf{m}_j^{(i)} - \mathbf{m}^{(i,j-1)}} + \text{rowsum}(\mathbf{P}_j^{(i)}) \in \mathbb{R}^{B_r}$ .
11:    On chip, compute  $\mathbf{O}_j^{(i)} = \text{diag}(e^{\mathbf{m}_j^{(i)} - \mathbf{m}^{(i,j-1)}})^{-1} \mathbf{O}^{(i,j-1)} + \mathbf{P}_j^{(i)} \mathbf{V}_j$ .
12:   end for
13:   On chip, compute  $\mathbf{O}_i = \text{diag}(\ell_i^{(T_c)})^{-1} \mathbf{O}^{(i,T_c)}$ .
14:   On chip, compute  $L_i = \mathbf{m}_i^{(T_c)} + \log(\ell_i^{(T_c)})$ .
15:   Write  $\mathbf{O}_i$  to HBM as the  $i$ -th block of  $\mathbf{O}$ .
16:   Write  $L_i$  to HBM as the  $i$ -th block of  $L$ .
17: end for
18: Return the output  $\mathbf{O}$  and the logsumexp  $L$ .

```

- We have to tell each Thread what to do, i.e. 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:
  - o Allocate N threads
  - o Assign a unique 'Thread Identifier' to each thread
  - o Run the threads according to the Thread Identifier assigned to it

```

// test_vector_add.c
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/time.h>
#include <sys/types.h>

typedef int int_TType;

static void test_vector_add(int_TType *out, int_TType *in, int_TType *in2, int N)
{
    int i = 0;
    while (i < N)
    {
        *out++ = *in++ + *in2++;
        i++;
    }
}

// test_vector_add.c
int main()
{
    int_TType *out = (int_TType *) malloc(N * sizeof(int_TType));
    int_TType *in = (int_TType *) malloc(N * sizeof(int_TType));
    int_TType *in2 = (int_TType *) malloc(N * sizeof(int_TType));
    test_vector_add(out, in, in2, N);
    return 0;
}

```

- Threads are launched by CUDA in multiples of 32 (ie, launched as a warp, 1 warp has 32 threads)

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.

- remaining 24 threads to do nothing.
- 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:
  - o In earlier case, 8 threads were launched, meaning remaining 24 threads were also launched but remain IDLE as they don't do any work
  - o This is called Control Divergence and causes lesser throughput, which software engineer should aim to minimize

Diagram illustrating the mapping of memory blocks to cache blocks:

Memory blocks are mapped to cache blocks as follows:

- Memory blocks 0, 1, 2, 3 map to Cache block  $B_0$ .
- Memory blocks 4, 5, 6, 7 map to Cache block  $B_1$ .
- Memory blocks 8, 9, 10, 11 map to Cache block  $B_2$ .
- Memory blocks 12, 13, 14, 15 map to Cache block  $B_3$ .

Number of blocks =  $\frac{N}{2} \rightarrow \text{Block size} = 4$

So, # of Blocks =  $8 / 2 = 4$

Block ID: 0, 1, 2 or 3

General formula to calculate # of Blocks:  
 $\# \text{ of Blocks} = \text{Total \# of Data Elements (N)} / \text{Block Size}$

# of Threads per block (considering 1 Thread per Data Element)

1. Why does each Block have 2 Threads here?

```
// 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;
```

Fancy way of writing cell(NUM\_COLS / COLS, block\_size)

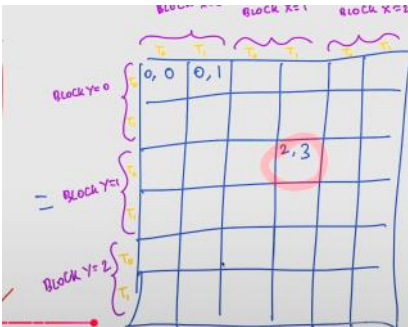
```
dim3 grid(num_blocks_COLS, num_blocks_ROWS, 1);
dim3 block(COLS_block_size, ROWS_block_size, 1);
```

# of Threa

```

96  m_matrix_add_sub m
97  {
98      test_matrix_add(int, int, int)
99      // void test_matrix_add(int ROWS_DIM, int NUM_COLS, int ROWS_BLOCK_SIZE, int COLS_BLOCK_SIZE)
100      for (int i = 0; i < NUM_ROWS; i++)
101      {
102          for (int j = 0; j < NUM_COLS; j++)
103          {
104              A[i][j] = static_cast<int>((i + NUM_COLS * j)
105              B[i][j] = rand() % 100;
106              C[i][j] = rand() % 100;
107          }
108      }
109  }

```



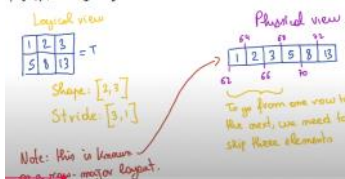
```

30 void test_matrix_add(int N, int NUM_ROWS, int ROWS_BLOCK_SIZE, int COLS_BLOCK_SIZE)
31 {
32     for (int i = 0; i < NUM_ROWS; i++)
33     {
34         for (int j = 0; j < NUM_COLS; j++)
35         {
36             size_t index = static_cast<size_t>(i * NUM_COLS + j);
37             A[index] = rand() % 100;
38             B[index] = rand() % 100;
39         }
40     }
41 }
42
43 // Allocate device memory for C
44 CUDA_CHECK(cudaMalloc(&cudA_A, sizeof(EL_TYPE) * NUM_ROWS * NUM_COLS));
45 CUDA_CHECK(cudaMalloc(&cudB_B, sizeof(EL_TYPE) * NUM_ROWS * NUM_COLS));
46 CUDA_CHECK(cudaMalloc(&cudC_OUT, sizeof(EL_TYPE) * NUM_ROWS * NUM_COLS));
47
48 // Transfer the matrices to the device
49 CUDA_CHECK(cudaMemcpy_d_A_A, sizeof(EL_TYPE) * NUM_ROWS * NUM_COLS, cudaMemcpyHostToDevice);
50 CUDA_CHECK(cudaMemcpy_d_B_B, sizeof(EL_TYPE) * NUM_ROWS * NUM_COLS, cudaMemcpyHostToDevice);
51
52 cudaEvent_t start_kernel, stop_kernel;
53 CUDA_CHECK(cudaEventCreate(&start_kernel));
54 CUDA_CHECK(cudaEventCreate(&stop_kernel));
55
56 CUDA_CHECK(cudaDeviceSynchronize());
57
58 // Define the launch grid
59 int num_blocks_ROWS = (NUM_ROWS + ROWS_BLOCK_SIZE - 1) / ROWS_BLOCK_SIZE; // ceil(num_ROWS / ROWS_BLOCK_SIZE)
60 int num_blocks_COLS = (NUM_COLS + COLS_BLOCK_SIZE - 1) / COLS_BLOCK_SIZE; // ceil(num_COLS / COLS_BLOCK_SIZE)
61 printf("Matrix Add : M: %d, N: %d will be processed by (%d x %d) blocks of size (%d x %d)", NUM_ROWS, NUM_COLS, num_blocks_ROWS, num_blocks_COLS, ROWS_BLOCK_SIZE, COLS_BLOCK_SIZE);
62 dim3 grid(num_blocks_ROWS, num_blocks_COLS, 1);
63 dim3 blockDim(ROWS_BLOCK_SIZE, COLS_BLOCK_SIZE, 1); // Yes, it is better than a declarative change
64
65 // Run the kernel
66 cudaMatrixAddOnGPU_blockRow_d_OUT_d_A_d_B_NUM_ROWS_NUM_COLS(grid, blockDim);
67
68 // Check for launch errors
69 CUDA_CHECK(cudaGetLastError());
70 CUDA_CHECK(cudaPeekAtLastError());
71 CUDA_CHECK(cudaDeviceSynchronize());
72 CUDA_CHECK(cudaEventSynchronize(&stop_kernel));
73
74 // Calculate elapsed milliseconds
75 float milliseconds_kernal = 0;
76 CUDA_CHECK(cudaEventElapsedTime(&milliseconds_kernal, start_kernel, stop_kernel));
77 printf("Matrix Add : Elapsed time: %f ms\n", milliseconds_kernal);
78
79 // Copy back the result from the device to the host
80 CUDA_CHECK(cudaMemcpy_d_OUT_d_OUT, sizeof(EL_TYPE) * NUM_ROWS * NUM_COLS, cudaMemcpyDeviceToHost);
81
82 // Free the memory on the device
83 CUDA_CHECK(cudaFree_d_A);
84 CUDA_CHECK(cudaFree_d_B);
85 CUDA_CHECK(cudaFree_d_OUT);

```

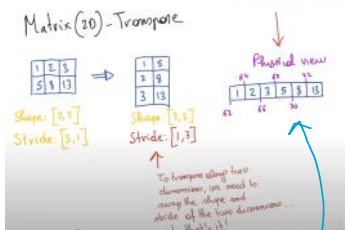
```
int row_index = blockIdx.y * blockDim.y + threadIdx.y;
int col_index = blockIdx.x * blockDim.x + threadIdx.x;
```

Matrix (2D)

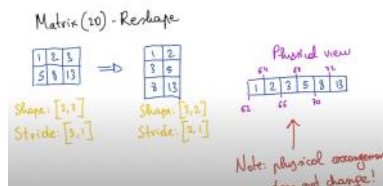


- Go to the next row dimension (eg: 3 here)
- Go to the next column dimension (eg: 1 here)

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



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



### Vector addition example

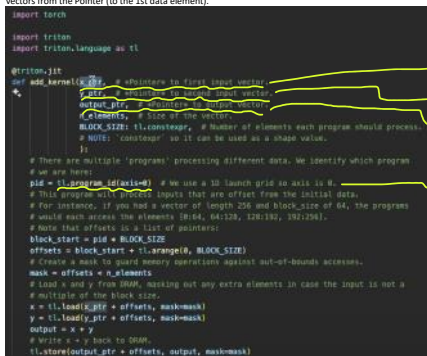
[illegible]

- # 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 'v'

→ Pointer to 1st element of vector x

Pointer to 1st element of vector 'V'

→ Pointer to 1st element of vector y

→ Pointer to 1st element of vector 'output'

```

// Pointer to 1st element of vector 'output'

```

→ The Block ID (CUDA) equivalent for

The Block\_ID (CUDA) equivalent for

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  $B_q, B_k$   
1. Divide  $\mathbf{Q}$  into  $T_q = \lceil \frac{N}{B_q} \rceil$  blocks  $\mathbf{Q}_1, \dots, \mathbf{Q}_{T_q}$  of size  $B_q \times d$  each, and divide  $\mathbf{K}, \mathbf{V}$  in to  $T_k = \lceil \frac{N}{B_k} \rceil$  blocks  $\mathbf{K}_1, \dots, \mathbf{K}_{T_k}$  and  $\mathbf{V}_1, \dots, \mathbf{V}_{T_k}$  of size  $B_k \times d$  each.  
2. Divide the output  $\mathbf{O}$  into  $T_o$  blocks  $\mathbf{O}_1, \dots, \mathbf{O}_{T_o}$  of size  $B_o \times d$  each, and divide the logsumexp  $\mathbf{L}$  into  $T_l$  blocks  $\mathbf{L}_1, \dots, \mathbf{L}_{T_l}$  of size  $B_l \times d$  each.  
3. **for**  $1 \leq i \leq T_q$  **do**  $\leftarrow$  FOR EACH  $\mathbf{Q}_i$  BLOCK  
4. Load  $\mathbf{Q}_i$  from HBM to on-chip SRAM.  
5. **for**  $1 \leq j \leq T_k$  **do**  $\leftarrow$  FOR EACH  $\mathbf{K}_j$  BLOCK  
6. On-chip compute  $\mathbf{Q}_i^T \mathbf{K}_j = (\mathbf{Q}_i^T \mathbf{K}_j)_{(B_q \times B_k) \times d}$  in  $\mathbb{R}^{B_q \times B_k}$ ,  $\mathbf{Q}_i^T \mathbf{K}_j = \exp(\mathbf{Q}_i^T \mathbf{K}_j) \times \mathbf{Q}_i^T \mathbf{K}_j \in \mathbb{R}^{B_q \times B_k}$ .  
7. Load  $\mathbf{V}_j$  from HBM to on-chip SRAM.  
8. On-chip compute  $\mathbf{Q}_i^T \mathbf{V}_j = (\mathbf{Q}_i^T \mathbf{V}_j)_{(B_q \times B_k) \times d}$  in  $\mathbb{R}^{B_q \times B_k}$ ,  $\mathbf{Q}_i^T \mathbf{V}_j = \exp(\mathbf{Q}_i^T \mathbf{V}_j) \times \mathbf{Q}_i^T \mathbf{V}_j \in \mathbb{R}^{B_q \times B_k}$ .  
9. **end for**  
10. On-chip compute  $\mathbf{O}_i^T = \text{diag}(\mathbf{Q}_i^T \mathbf{K}_j)^{-1} \times \mathbf{Q}_i^T \mathbf{V}_j$ .  
11. **end for**  
12. On-chip compute  $\mathbf{O}_i = \text{diag}(\mathbf{O}_i^T)^{-1} \times \mathbf{O}_i^T$ .  
13. On-chip compute  $\mathbf{L}_i = \text{diag}(\mathbf{O}_i^T \mathbf{O}_i) \times \mathbf{O}_i^T$ .  
14. Write  $\mathbf{O}_i$  to HBM in the  $i$ -th block of  $\mathbf{O}$ .  
15. Write  $\mathbf{L}_i$  to HBM in the  $i$ -th block of  $\mathbf{L}$ .  
16. **end for**  
17. Return the output  $\mathbf{O}$  and the logsumexp  $\mathbf{L}$ .

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)

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

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<sub>i</sub>,  
And use them during backward pass

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

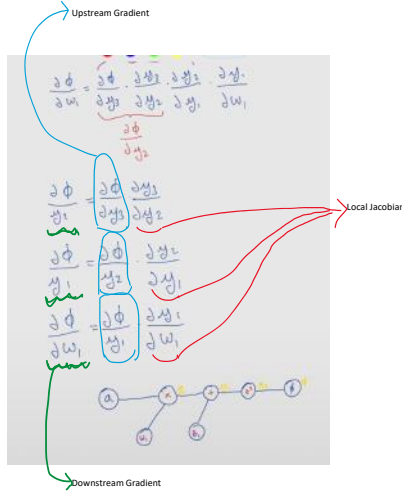
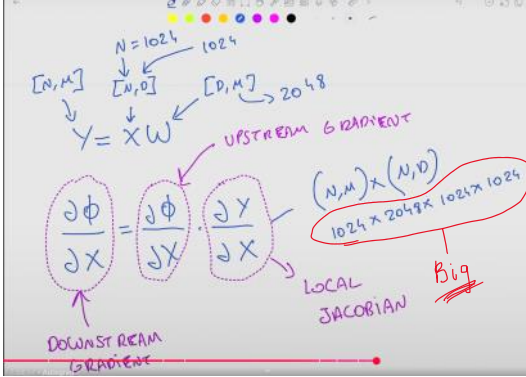
Causal vs Non-causal Attention:

- Causal (Masked):
  - o Token can ONLY attend to itself and tokens BEFORE it in the sequence, NOT any tokens AFTER it
  - o Prevents cheating in generative models (future tokens shouldn't inform current predictions)
  - o Usage: Autoregressive settings (eg: language generation)
  - o 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 i > j)
- Non-causal (Bidirectional):
  - o Lets EVERY token attend to EVERY position in the sequence, including future ones
  - o Usage: machine translation (encoder), BERT-style, not generating text one token at a time but learning bidirectional context
  - o 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



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  $\mathbf{L} \in \mathbb{R}^N$  in HBM, block sizes  $B_q, B_k, B_o$   
1. Divide  $\mathbf{Q}$  into  $T_q = \lceil \frac{N}{B_q} \rceil$  blocks  $\mathbf{Q}_1, \dots, \mathbf{Q}_{T_q}$  of size  $B_q \times d$  each, and divide  $\mathbf{K}, \mathbf{V}$  in to  $T_k = \lceil \frac{N}{B_k} \rceil$  blocks  $\mathbf{K}_1, \dots, \mathbf{K}_{T_k}$  and  $\mathbf{V}_1, \dots, \mathbf{V}_{T_k}$  of size  $B_k \times d$  each.  
2. Divide  $\mathbf{O}$  into  $T_o$  blocks  $\mathbf{O}_1, \dots, \mathbf{O}_{T_o}$  of size  $B_o \times d$  each, divide  $\mathbf{dO}$  into  $T_o$  blocks  $\mathbf{dO}_1, \dots, \mathbf{dO}_{T_o}$  of size  $B_o \times d$  each, and divide  $\mathbf{L}$  into  $T_l$  blocks  $\mathbf{L}_1, \dots, \mathbf{L}_{T_l}$  of size  $B_l$  each.  
3. Initialize  $\mathbf{dQ} = (\mathbf{O} \mathbf{dO})_{(N \times d) \times d}$  in  $\mathbb{R}^{N \times d}$  and divide it into  $T_q$  blocks  $\mathbf{dQ}_1, \dots, \mathbf{dQ}_{T_q}$  of size  $B_q \times d$  each. Divide  $\mathbf{dK}, \mathbf{dV} \in \mathbb{R}^{N \times d}$  in to  $T_k$  blocks  $\mathbf{dK}_1, \dots, \mathbf{dK}_{T_k}$  and  $\mathbf{dV}_1, \dots, \mathbf{dV}_{T_k}$  of size  $B_k \times d$  each.  
4. Compute  $\mathbf{D} = \text{rowsum}(\mathbf{dO} \circ \mathbf{O}) \in \mathbb{R}^d$  (pointwise multiply), write  $\mathbf{D}$  to HBM and divide it into  $T_l$  blocks  $\mathbf{D}_1, \dots, \mathbf{D}_{T_l}$  of size  $B_l$  each.  
5. **for**  $1 \leq j \leq T_k$  **do**  
6. Load  $\mathbf{K}_j, \mathbf{V}_j$  from HBM to on-chip SRAM.  
7. Initialize  $\mathbf{dK}_j = (\mathbf{O} \mathbf{dO})_{(B_k \times d) \times d}$ ,  $\mathbf{dV}_j = (\mathbf{O} \mathbf{dO})_{(B_k \times d) \times d}$  on SRAM.  
8. **for**  $1 \leq i \leq T_q$  **do**  
9. Load  $\mathbf{Q}_i, \mathbf{O}_i, \mathbf{dQ}_i, \mathbf{L}_i, \mathbf{D}_i$  from HBM to on-chip SRAM.  
10. On chip, compute  $\mathbf{S}_i^{(j)} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_q \times B_k}$ .  
11. On chip, compute  $\mathbf{P}_i^{(j)} = \exp(\mathbf{S}_i^{(j)} - \mathbf{L}_i) \in \mathbb{R}^{B_q \times B_k}$ .  
12. On chip, compute  $\mathbf{dV}_j \leftarrow \mathbf{dV}_j + (\mathbf{P}_i^{(j)})^T \mathbf{dO}_i \in \mathbb{R}^{B_k \times d}$ .  
13. On chip, compute  $\mathbf{dP}_i^{(j)} = \mathbf{dO}_i \mathbf{V}_j^T \in \mathbb{R}^{B_q \times B_k}$ .  
14. On chip, compute  $\mathbf{dS}_i^{(j)} = \mathbf{P}_i^{(j)} \circ (\mathbf{dP}_i^{(j)} - \mathbf{D}_i) \in \mathbb{R}^{B_q \times B_k}$ .  
15. Load  $\mathbf{dQ}_i$  from HBM to SRAM, then on chip, update  $\mathbf{dQ}_i \leftarrow \mathbf{dQ}_i + \mathbf{dS}_i^{(j)} \mathbf{K}_j \in \mathbb{R}^{B_q \times d}$ , and write back to HBM.  
16. On chip, compute  $\mathbf{dK}_j \leftarrow \mathbf{dK}_j + \mathbf{dS}_i^{(j)T} \mathbf{Q}_i \in \mathbb{R}^{B_k \times d}$ .  
17. **end for**  
18. Write  $\mathbf{dK}_j, \mathbf{dV}_j$  to HBM.  
19. **end for**  
20. Return  $\mathbf{dQ}, \mathbf{dK}, \mathbf{dV}$ .

$$\begin{bmatrix} y_1 \\ \vdots \\ y_M \end{bmatrix} = \begin{bmatrix} x_1 & \dots & x_N \\ \vdots & \ddots & \vdots \\ x_1 & \dots & x_N \end{bmatrix} \begin{bmatrix} w_1 \\ \vdots \\ w_M \end{bmatrix}$$

Y (N, M) = X (N, D) \* W (D, M)

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's 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 won't 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

$$\frac{\partial \phi}{\partial \mathbf{x}} = \frac{\partial \phi}{\partial \mathbf{y}} \cdot \frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \frac{\partial \phi}{\partial \mathbf{y}} \begin{bmatrix} \mathbf{W}^T \\ \mathbf{0} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{1} \end{bmatrix}$$

In place of Jacobian

$$\frac{\partial \phi}{\partial \mathbf{W}} = \mathbf{x}^T \frac{\partial \phi}{\partial \mathbf{y}} = \begin{bmatrix} \mathbf{0} \\ \mathbf{1} \end{bmatrix}$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \end{bmatrix} = 0$$

(1,3) (3,4) (N,4)

$$\mathbf{a}_1 = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} \\ a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} \\ a_{11}b_{13} + a_{12}b_{23} + a_{13}b_{33} \\ a_{11}b_{14} + a_{12}b_{24} + a_{13}b_{34} \end{bmatrix}$$
$$\mathbf{a}_1 = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} = \sum_{j=1}^3 a_{1j}b_{j1}$$

(2)

FlashAttention1 Backward Pass algorithm

**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  $r \in \mathbb{R}$ , masking function MASK, dropout probability  $p_{\text{drop}}$ , pseudo-random number generator state  $\mathcal{R}$  from the forward pass.

- Set the pseudo-random number generator state to  $\mathcal{R}$ .
- Set block sizes  $B_r = \lceil \frac{M}{r} \rceil, B_c = \min(\lceil \frac{M}{r} \rceil, d)$ .
- 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.
- 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, divide  $\ell$  into  $T_r$  blocks  $\ell_1, \dots, \ell_{T_r}$  of size  $B_r$  each, divide  $m$  into  $T_r$  blocks  $m_1, \dots, m_{T_r}$  of size  $B_r$  each.
- Initialize  $\mathbf{dQ} = (\mathbf{O})_{N \times d}$  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  $\mathbf{dK} = (\mathbf{O})_{N \times d}, \mathbf{dV} = (\mathbf{O})_{N \times d}$  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.
- for  $1 \leq j \leq T_r$  do
- Load  $\mathbf{K}_j, \mathbf{V}_j$  from HBM to on-chip SRAM
- for  $1 \leq i \leq T_c$  do
- Load  $\mathbf{Q}_i, \mathbf{O}_i, \mathbf{dO}_i, \mathbf{dQ}_i, \ell_i, m_i$  from HBM to on-chip SRAM
- On chip, compute  $\mathbf{S}_{ij} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$
- On chip, compute  $\mathbf{S}_{ij}^{\text{masked}} = \text{MASK}(\mathbf{S}_{ij})$
- On chip, compute  $\mathbf{P}_{ij} = \text{diag}(\mathbf{I})^{1/2} \exp(\mathbf{S}_{ij}^{\text{masked}} - m_i) \in \mathbb{R}^{B_r \times B_c}$
- On chip, compute dropout mask  $\mathbf{Z}_{ij} \in \mathbb{R}^{B_r \times B_c}$  where each entry has value  $\frac{1}{1+p_{\text{drop}}}$  with probability  $1 - p_{\text{drop}}$  and value 0 with probability  $p_{\text{drop}}$
- On chip, compute  $\mathbf{P}_{ij}^{\text{masked}} = \mathbf{P}_{ij} \odot \mathbf{Z}_{ij}$  (pointwise multiply)
- On chip, compute  $\mathbf{dV}_j \leftarrow \mathbf{dV}_j + (\mathbf{P}_{ij}^{\text{masked}})^T \mathbf{dO}_i \in \mathbb{R}^{B_c \times d}$
- On chip, compute  $\mathbf{dP}_{ij}^{\text{masked}} = \mathbf{dO}_i \mathbf{V}_j^T \in \mathbb{R}^{B_r \times B_c}$
- On chip, compute  $\mathbf{dP}_{ij} = \mathbf{dP}_{ij}^{\text{masked}} \odot \mathbf{Z}_{ij}$  (pointwise multiply)
- On chip, compute  $\mathbf{D}_j = \text{row-sum}(\mathbf{dP}_{ij} \odot \mathbf{D}_j) \in \mathbb{R}^{B_r \times 1}$
- On chip, compute  $\mathbf{dK}_j \leftarrow \mathbf{P}_{ij} \odot (\mathbf{dP}_{ij} - \mathbf{D}_j) \in \mathbb{R}^{B_c \times B_r}$
- Write  $\mathbf{dQ}_i \leftarrow \mathbf{dQ}_i + r \mathbf{dS}_{ij} \mathbf{K}_j \in \mathbb{R}^{B_r \times d}$  to HBM
- On chip, compute  $\mathbf{dK}_j \leftarrow \mathbf{dK}_j + r \mathbf{dS}_{ij}^T \mathbf{Q}_i \in \mathbb{R}^{B_c \times d}$
- end for
- Write  $\mathbf{dK}_j \leftarrow \mathbf{dK}_j, \mathbf{dV}_j \leftarrow \mathbf{dV}_j$  to HBM
- end for
- for  $1 \leq j \leq T_r$  do
- Write  $\mathbf{dQ}_j \leftarrow \mathbf{dQ}_j, \mathbf{dK}_j \leftarrow \mathbf{dK}_j$  to HBM
- end for

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

- 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
  - To compute dq and dk vectors, need  $\mathbf{D}_j$ :

$$dq_j = \sum_i dS_{ij} k_j = \sum_i P_{ij} (dP_{ij} - D_j) k_j = \sum_i \frac{e^{o_i^T k_j}}{L_i} (d\alpha_i^T v_j - D_j) k_j$$

$$dk_j = \sum_i dS_{ij} q_i = \sum_i P_{ij} (dP_{ij} - D_j) q_i = \sum_i \frac{e^{o_i^T k_j}}{L_i} (d\alpha_i^T v_j - D_j) q_i$$

where  $\mathbf{D}_j$  is:

$$\mathbf{D}_j = \mathbf{P}_{ij}^T \mathbf{dP}_{ij} = \sum_i \left( \frac{e^{o_i^T k_j}}{L_i} \right) d\alpha_i^T v_j = d\alpha_i^T \sum_j \frac{e^{o_i^T k_j}}{L_i} v_j = d\alpha_i^T o_i$$

So effectively, we:

- Calculate  $\mathbf{D}_j$
- Calculate dq
- Calculate dk
- Fix Q, loop over all KV
- Fix K, loop over all Q

Triton Autotuning:

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

```
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 = Loop (...)
  B = Loop (...)
  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. Hkgro/triton-flash-attention: <https://github.com/hkgro/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

Disclaimer:

This document contains personal study notes and summaries related to hkgro/triton-flash-attention. These notes are not my original work and are not intended to claim credit for the source material. All rights, authorship, and intellectual property belong to the respective original creators.