From online safe softmax to FlashAttention-2

Youming Zhao*
youming0.zhao@gmail.com

First draft: October 9, 2023 Last update: November 3, 2023

Abstract

In this paper, we present all the technical details you need to know for implementing FlashAttention-2 from scratch. We benchmark the latency for writes/reads to and from GPU global memory. Interestingly, reads from memory is 40-1000x faster than writes to memory. Based on this observation, We propose a strategy to avoid writes to global memory in the stage of the backward pass of FlashAttention-2. Since reads are much cheaper than writes, the tradeoff that we need to perform more reads in the inner loop is acceptable.

1 Overview

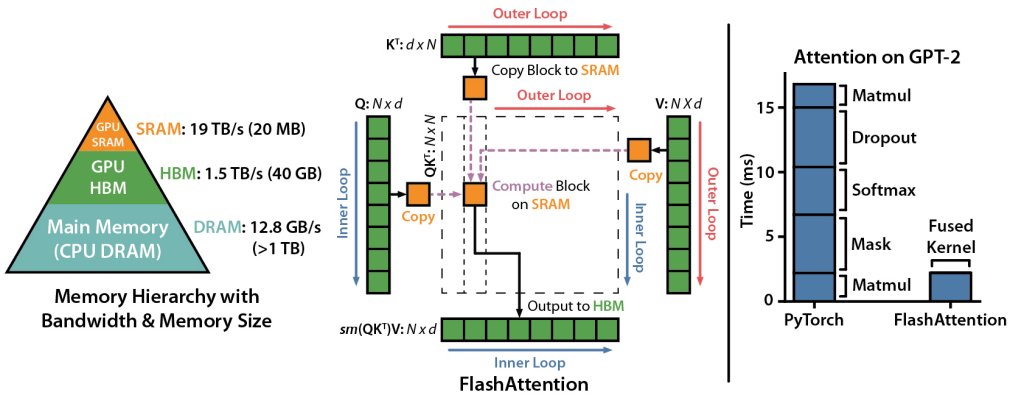

Figure 1: **Left**: Hardware characteristics, GPU shared memory (SRAM) is more than 10x faster than GPU global memory (HBM). **Middle**: FlashAttention employs tiling on shared memory to reduce memory accesses for speedup. **Right**: The performance of fusing all operations inside attention into a CUDA kernel. Image source: `https://github.com/Dao-AILab/flash-attention/blob/main/assets/flashattn_banner.jpg`

2 Self-attention

Self-attention is an important part of the well-known Transformer architecture. We denote the sequence length and the head dimension by N and d, respectively. Given a query matrix $\mathbf{Q} \in \mathbb{R}^{N \times d}$

*Work done as an intern at Helixon.

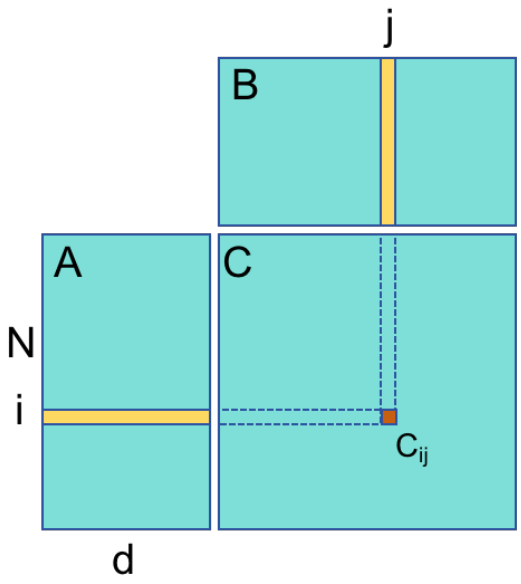

Figure 2: Matrix multiplication

and a key-value matrix pair $\mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$, self-attention consists of the following three steps.

$$\mathbf{S} = \mathbf{Q}\mathbf{K}^T \quad \text{(pre-softmax logits)} \tag{1}$$

$$\mathbf{P} = \text{softmax}(\frac{\mathbf{S}}{\sqrt{d}}) \quad \text{(attention-score matrix)} \tag{2}$$

$$\mathbf{O} = \mathbf{P}\mathbf{V} \quad \text{(attention output)} \tag{3}$$

where $\mathbf{S}$ is divided by $\sqrt{d}$ to prevent large pre-softmax logits from pushing the softmax function into regions where it has extremely small gradients Vaswani et al. (2017).

From the formulas (1) to (3), we can see that what standard attention implementation does is large matrix multiplication and a lot of memory accesses. Specifically, performing (1) needs to read $\mathbf{Q}$ and $\mathbf{K}$ from high bandwidth memory (HBM), compute $\mathbf{S} = \mathbf{Q}\mathbf{K}^T \in \mathbb{R}^{N \times N}$, and then write $\mathbf{S}$ to HBM. What follows is calculating $\mathbf{P}$ and $\mathbf{O}$ which also involves matrix multiplication and large matrices reads/writes on HBM, which takes $O(N^2)$ memory. Normally, $N \gg d$, for example, $N = 16$k and $d = 128$ in GPT-3. However, a large number of reads/writes on HBM is extremely time-consuming due to the slow HBM bandwidth, 1.5-2.0TB/s, compared to the fast shared memory bandwidth, 19TB/s for A100 GPU.

One of the core ideas of FlashAttention is computing attention on GPU's shared memory by blocks via tiling. In the next section, we will talk about how to parallelize matrix multiplication on GPU with Compute Unified Device Architecture (CUDA), particularly using shared memory.

3 Matrix multiplication using shared memory on GPU

Before we dive into parallelizing matrix multiplication with shared memory on GPU, we first describe how to implement matrix multiplication on CPU. As shown in Figure 2, an entry of the product $\mathbf{C} = \mathbf{A}\mathbf{B}$, i.e. $\mathbf{C}_{ij}$, is equal to the sum of the element-wise product of the i-th row of $\mathbf{A}$ and the j-th column of $\mathbf{B}$. The implementation of matrix multiplication on CPU is presented in Listing 1. The indexing might be a little confusing. Keep in mind that a 2D array is stored in the computer's memory one row following another.

Listing 1: Implementation of matrix multiplication suited for CPU

```
void MatrixMulOnCPU(float *A, float *B, float *C, int N, int d)
{// A 2D array is stored in the memory one row following another.
    for(int i=0; i<N; ++i)
```

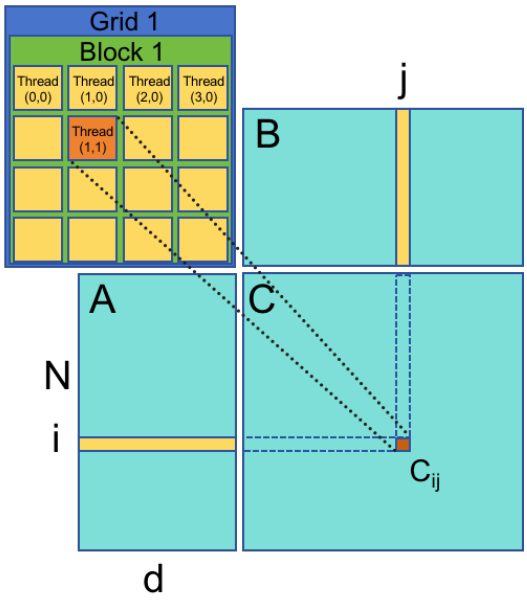

Figure 3: Matrix multiplication on GPU without using shared memory

```
    {
        for(int j=0; j<N; ++j)
        {
            double tmpSum = 0;
            for(int k=0; k < d; ++k)
            {
                tmpSum += A[i*d + k] * B[k*d + j];
            }
            C[i*d + j] = tmpSum;
        }
    }
}
```

As we can see from Figure. 2, the computation of the entries of $\mathbf{C}$ is fully parallelizable. Now we consider to handle the calculation of each entry with one thread on GPU via CUDA programming. Specifically, each thread reads one row of $\mathbf{A}$ and one column of $\mathbf{B}$ and calculates the corresponding entry of $\mathbf{C}$ as illustrated in Figure. 3. Also, the code is provided in Listing 2 which does not take advantage of shared memory. In CUDA, the keyword '__global__' is used to declare a CUDA kernel function. Logically speaking, a kernel function corresponds to a grid which comprises of multiple blocks and each block contains multiple threads. A CUDA kernel example suited for GPU is presented in Listing 2.

Listing 2: Implementation of matrix multiplication suited for GPU

```
__global__ void matrixMulOnGPU(float *A, float *B, float *C, int N, int d)
{// Each thread calculates one entry of C by accumulating results into tmpSum.
    int col = blockDim.x * blockIdx.x + threadIdx.x;
    int row = blockDim.y * blockIdx.y + threadIdx.y;
    double tmpSum = 0.0;
    for (int k=0; k<d; k++)
    {
        tmpSum += A[row*N + k] * B[col+k*N];
    }
    C[row*N + col] = tmpSum;
}
// define matrix dimension (N, d)
int N = 1024;
int d = 128;
// set up execution configuration
dim3 block(16, 16); // the number of threads per block
dim3 grid((N + block.x - 1) / block.x, (d + block.y - 1) / block.y); // the number of
     blocks per grid

matrixMulOnGPU<<<grid, block>>>(A, B, C, N, d); // invoke the kernel
```

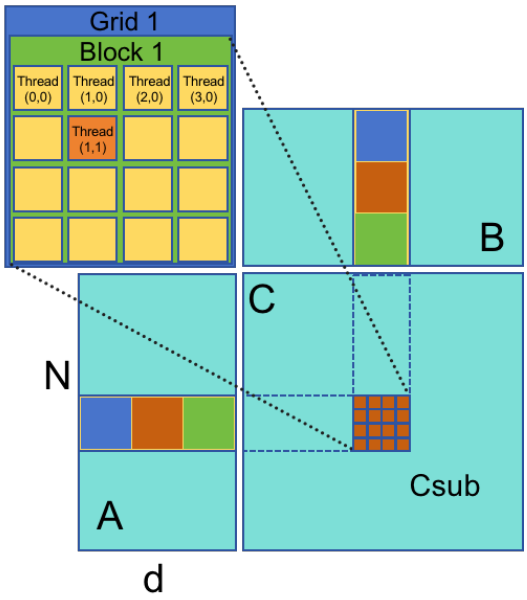

Figure 4: Matrix multiplication on GPU with shared memory. The small squares in **A** and **B** reside in shared memory.

However, Listing 2 involves $O(N^2)$ memory accesses, which slows wall-clock time significantly due to $N \gg d$. Shared memory is expected to be much faster than global memory. It can be used as scratchpad memory (or software managed cache) to minimize global memory accesses from a CUDA block[2].

Fortunately, shared memory can help us reduce memory accesses. Meanwhile, shared memory is fast to read and write. Threads within a block can cooperate by sharing data through some shared memory and by synchronizing their execution to coordinate memory accesses.

Specifically, as shown in Listing 3, each thread block is responsible for one sub-matrix $Csub$ of **C** and each thread works on one entry of $Csub$. As illustrated in Figure. 4, $Csub$ is equal to the product of two rectangular matrices: $Asub$, the sub-matrix of **A** of dimension $(block\_size, A.width)$ and $Bsub$, the sub-matrix of **A** of dimension $(B.height, block\_size)$. In order to fit into limited shared memory, these two rectangular matrices are separated into many square matrices of dimension $block\_size$. Then $Csub$ is calculated as the sum of the products of these square matrices. Each of these products is performed by first loading the two corresponding matrices from global memory to shared memory with each thread loading one entry of each matrix, and then each thread computes one element of the product. Each thread accumulates the result of each of these products into a register and once done writes the result to global memory.

By tiling the computation this way, fast shared memory can help us save a lot of global memory bandwidth because **A** is read $B.width/block\_size$ times from global memory and **B** is read $A.height/block\_size$ times.

Listing 3: Implementation of matrix multiplication with shared memory

```
#define BLOCK_SIZE 16
__global__ void matrixMulSharedMemory(float *A, float *B, float *C, int N, int d)
{   // Shared memory used to store Asub and Bsub, respectively.
    // shared memory is shared within one thread block.
    __shared__ float Asub[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bsub[BLOCK_SIZE][BLOCK_SIZE];
    // each thread has one tmpSum to accumulates the results
    double tmpSum = 0; // tmpSum is stored in registers
    // row and column in C
    unsigned int col = BLOCK_SIZE * blockIdx.x + threadIdx.x;
    unsigned int row = BLOCK_SIZE * blockIdx.y + threadIdx.y;
```

[2] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#shared-memory

```
    unsigned int col_sub = threadIdx.x; // Thread column within Csub
    unsigned int row_sub = threadIdx.y; // Thread row within Csub
    // Loop over all the sub-matrices of A and B required to compute Csub
    for (int m=0; m<d/BLOCK_SIZE; m++)
    {   // load Asub and Bsub from device memory to shared memory
        // each thread load one entry of each sub-matrix
        Asub[row_sub][col_sub] = A[row*d + m*BLOCK_SIZE + col_sub];
        Bsub[row_sub][col_sub] = B[col + (row_sub + m*BLOCK_SIZE) * N];
        // Synchronize to ensure the sub-matrices are loaded before computation.
        __syncthreads();
        // Multiply Asub and Bsub together
        for (int k=0; k<BLOCK_SIZE; k++)
        {
            tmpSum += Asub[row_sub][k] * Bsub[k][col_sub];
        }
        // Before loading the next two sub-matrices, synchronize to ensure
        // the computation involving these two sub-matrices is done.
        __syncthreads();
    }
    C[row*N + col] = tmpSum; // write Csub to device memory; each thread writes one
}
```

4 Safe softmax

Recall the definition of the softmax function

$$\text{softmax}(x_1, x_2, \ldots, x_N) = \left\{ \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}} \right\}_{i=1}^{N}. \tag{4}$$

We can observe that when x_i is large, e^{x_i} may overflow. For instance, the maximum normal number of *float16* supports is 65504[3], which implies that if $x_i \geq 11.09$, e^{x_i} will overflow. Therefore, people usually subtract $\max_i x_i$ from all x_i before applying the softmax operator. Thus, the safe softmax operator is defined as

$$\text{softmax}(x_1, x_2, \ldots, x_N) = \left\{ \frac{e^{x_i - m}}{\sum_{j=1}^{N} e^{x_j - m}} \right\}_{i=1}^{N} \tag{5}$$

where $m = \max_{i=1}^{N} x_i$. In this situation, we have $x_i - m \leq 0$ which is safe since the exponentiation is accurate for negative numbers. Then the computation of the safe softmax operator is summarized as a 3-pass algorithm in Algorithm 1. This algorithm needs to loop over 1 to N three times. x_i is the

Algorithm 1: the safe softmax operator

input : $m_0 = -\infty, l_0 = 0$
output : a_i denote the output of the safe softmax operator

for $i \leftarrow 1$ **to** N **do**
  $m_i = \max\{m_{i-1}, x_i\}$;
for $i \leftarrow 1$ **to** N **do**
  $l_i = l_{i-1} + e^{x_i - m_N}$;
for $i \leftarrow 1$ **to** N **do**
  $a_i = \frac{e^{x_i - m_N}}{l_N}$;

pre-softmax logits given by $\mathbf{QK}^T$. However, an A100 GPU has 108 streaming multiprocessors and each of them has only 164KB shared memory. That means we do not have enough shared memory to store them. Thus, we have to access them three times, which is not I/O efficient.

5 Online softmax

Milakov and Gimelshein (2018) proposed an online normalizer calculation for safe softmax. In Algorithm 2, the calculations of m and l are fused into one loop, which saves memory accesses. The

[3] https://en.wikipedia.org/wiki/Half-precision_floating-point_format

main idea is the surrogate trick which employs l'_i to represent the current normalizer (denominator) and l'_i is always the exact denominator for the i pre-softmax logits that have been traversed. This property gaurantees $l'_N = l_N$.

Algorithm 2: the safe softmax operator with online normalizer calculation

Input : $m_0 = -\infty, l'_0 = 0$
Output : a_i denote the output of the safe softmax operator

for $i \leftarrow 1$ **to** N **do**
  $m_i = \max\{m_{i-1}, x_i\}$
  $l'_i = l'_{i-1} e^{m_{i-1} - m_i} + e^{x_i - m_i}$
for $i \leftarrow 1$ **to** N **do**
  $a_i = \frac{e^{x_i - m_N}}{l'_N}$

6 FlashAttention

As we can see from Algorithm 2, we are unable to fuse the two for-loops into one-loop. Before diving into FlashAttention, we first recap the standard self-attention in Algorithm 3. With online safe softmax, since the computation of each row of the self-attention output is independent, we only analyze the k-th row of $\mathbf{O}$, i.e. $\mathbf{O}[k,:]$ for simplicity. We observe that the second for-loop depends on the final results m_N and l'_N. Is it possible to fuse them into one for-loop?

6.1 Forward pass

Algorithm 3: self-attention with online safe softmax

Input : $m_0 = -\infty, l'_0 = 0$
Input : $\mathbf{Q}[k,:]$ denotes the k-th row of the query matrix $\mathbf{Q}$
Input : $\mathbf{K}$ and $\mathbf{V}$ denote the key-value pair matrices
Output : $\mathbf{P}[k,:]$ denote the k-th of the attention score out of the safe softmax operator
Output : $\mathbf{O}[k,:]$ denotes the k-th row of the attention output matrix $\mathbf{O}$

for $i \leftarrow 1$ **to** N **do**
  $x_i = (\mathbf{Q}[k,:])^T \mathbf{K}^T[:,i]$
  $m_i = \max\{m_{i-1}, x_i\}$
  $l'_i = l'_{i-1} e^{m_{i-1} - m_i} + e^{x_i - m_i}$
for $i \leftarrow 1$ **to** N **do**
  $\mathbf{P}[k,i] = \frac{e^{x_i - m_N}}{l'_N}$
  $\mathbf{o}_i = \mathbf{o}_{i-1} + \mathbf{P}[k,i]\mathbf{V}[i,:]$
$\mathbf{O}[k,:] = \mathbf{o}_N$

We can express the second for-loop in a compact way as follows.

$$\mathbf{o}_i = \sum_{j=1}^{i} \mathbf{P}[k,j]\mathbf{V}[j,:] = \sum_{j=1}^{i} \frac{e^{x_j - m_N}}{l'_N} \mathbf{V}[j,:]. \tag{6}$$

It is natural to think of the surrogate trick employed by the online safe softmax. We perform this trick on (6) as follows.

$$\mathbf{o}'_i = \sum_{j=1}^{i} \frac{e^{x_j - m_i}}{l'_i} \mathbf{V}[j,:] \tag{7}$$

$$= \sum_{j=1}^{i-1} \frac{e^{x_j - m_i}}{l'_i} \mathbf{V}[j,:] + \frac{e^{x_i - m_i}}{l'_i} \mathbf{V}[i,:] \tag{8}$$

$$= \frac{l'_{i-1}}{l'_i e^{m_i - m_{i-1}}} \frac{l'_i e^{m_i - m_{i-1}}}{l'_{i-1}} \sum_{j=1}^{i-1} \frac{e^{x_j - m_i}}{l'_i} \mathbf{V}[j,:] + \frac{e^{x_i - m_i}}{l'_i} \mathbf{V}[i,:] \tag{9}$$

$$= \frac{l'_{i-1}}{l'_i e^{m_i - m_{i-1}}} \sum_{j=1}^{i-1} \frac{e^{x_j - m_{i-1}}}{l'_{i-1}} \mathbf{V}[j,:] + \frac{e^{x_i - m_i}}{l'_i} \mathbf{V}[i,:] \tag{10}$$

$$= \frac{l'_{i-1}}{l'_i e^{m_i - m_{i-1}}} \mathbf{o}'_{i-1} + \frac{e^{x_i - m_i}}{l'_i} \mathbf{V}[i,:] = \frac{l'_{i-1} e^{m_{i-1} - m_i}}{l'_i} \mathbf{o}'_{i-1} + \frac{e^{x_i - m_i}}{l'_i} \mathbf{V}[i,:]. \tag{11}$$

Given (11), self-attention can be fused into one for-loop as described in Algorithm 4. As can be

Algorithm 4: fused self-attention

Input : $m_0 = -\infty$, $d'_0 = 0$
Input : $\mathbf{Q}[k,:]$ denotes the k-th row of the query matrix $\mathbf{Q}$
Input : $\mathbf{K}$ and $\mathbf{V}$ denote the key-value pair matrices
Output : $\mathbf{O}[k,:]$ denotes the k-th row of the attention output matrix $\mathbf{O}$

for $i \leftarrow 1$ **to** N **do**
  $x_i = (\mathbf{Q}[k,:])^T \mathbf{K}^T[:,i]$
  $m_i = \max\{m_{i-1}, x_i\}$
  $l'_i = l'_{i-1} e^{m_{i-1} - m_i} + e^{x_i - m_i}$
  $\mathbf{o}'_i = \frac{l'_{i-1}}{l'_i e^{m_i - m_{i-1}}} \mathbf{o}'_{i-1} + \frac{e^{x_i - m_i}}{l'_i} \mathbf{V}[i,:]$
$\mathbf{O}[k,:] = \mathbf{o}'_N$

seen in Algorithm 4, the computation of each row of $\mathbf{O}$ is independent with other rows. Therefore, we can defer the operation regarding the denominator to the end of the loop, which is exactly one improvement for the forward pass in FlashAttention2 Dao (2023). Normally, we need to store m and l for the backward pass. However, due to $\frac{e^{x_i - m}}{l} = e^{x_i - m - \ln l}$ we only need to store $m + \ln l$ for each row of $\mathbf{O}$, which allows us to write less to global memory. Furthermore, since writes are expensive, this will yield more speedups. This is the second improvement in FlashAttention2.

Since all the operations in self-attention are associative, we can employ tiling to make full use of shared memory. Combining with the aforementioned improvements, the algorithm with tiling is described in Algorithm 5.

Algorithm 5: fused self-attention with tiling

Input : $m_0 = -\infty$, $l'_0 = 0$, block size b
Input : $\mathbf{Q}[k,:]$ denotes the k-th row of the query matrix $\mathbf{Q}$
Input : $\mathbf{K}$ and $\mathbf{V}$ denote the key-value pair matrices
Output : $\mathbf{O}[k,:]$ denotes the k-th row of the attention output matrix $\mathbf{O}$

for $i \leftarrow 1$ **to** N/b **do**
  $\mathbf{x}_i = (\mathbf{Q}[k,:])^T \mathbf{K}^T[:,(i-1)b : ib]$
  $m_i = \max\{m_{i-1}, \max\{\mathbf{x}_i\}\}$
  $l'_i = l'_{i-1} e^{m_{i-1} - m_i} + \sum(e^{\mathbf{x}_i - m_i})$
  $\mathbf{o}'_i = e^{m_{i-1} - m_i} \mathbf{o}'_{i-1} + e^{\mathbf{x}_i - m_i} \mathbf{V}[(i-1)b : ib,:]$
Return $\mathbf{O}[k,:] = l_{N/b} \mathbf{o}'_{N/b}$, $L = m + \ln l_{N/b}$

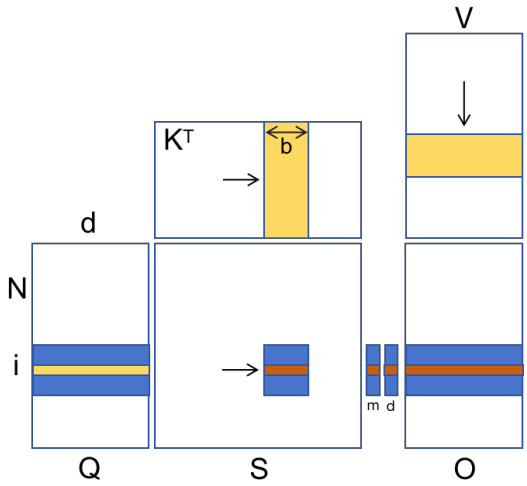

Figure 5: fused self-attention with tiling

Now we include all the details into the forward pass. Given the input sequences $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$, we want to compute the attention output $\mathbf{O} \in \mathbb{R}^{N \times d}$ as follows.

$$\mathbf{S} = \tau \mathbf{Q}\mathbf{K}^T \in \mathbb{R}^{N \times N}, \text{ where } \tau = \frac{1}{\sqrt{d}} \tag{12}$$

$$\mathbf{S}^{\text{masked}} = \text{MASK}(\mathbf{S}) \in \mathbb{R}^{N \times N} \tag{13}$$

$$\mathbf{P} = \text{softmax}(\mathbf{S}^{\text{masked}}) \in \mathbb{R}^{N \times N} \tag{14}$$

$$\mathbf{P}^{\text{dropped}} = \text{dropout}(\mathbf{P}, p_{\text{drop}}) \in \mathbb{R}^{N \times N} \tag{15}$$

$$\mathbf{O} = \mathbf{P}^{\text{dropped}} \mathbf{V} \in \mathbb{R}^{N \times d}. \tag{16}$$

Also, we present the complete forward pass, including masks and dropout, in Algorithm 6.

6.2 Backward pass

Now we derive the backward pass of attention. Without loss of generality, we consider a scalar loss function ℓ. Given the output gradient $\frac{\partial \ell}{\partial \mathbf{O}}$, we need to compute $\frac{\partial \ell}{\partial \mathbf{Q}}$, $\frac{\partial \ell}{\partial \mathbf{K}}$, and $\frac{\partial \ell}{\partial \mathbf{V}}$ because we need these intermediate gradients to calculate the final gradients $\frac{\partial \ell}{\partial \mathbf{W}_q}$, $\frac{\partial \ell}{\partial \mathbf{W}_k}$ and $\frac{\partial \ell}{\partial \mathbf{W}_v}$, where $\mathbf{Q} = \mathbf{X}\mathbf{W}_q$, $\mathbf{K} = \mathbf{X}\mathbf{W}_k$ and $\mathbf{V} = \mathbf{X}\mathbf{W}_v$.

It is straightforward to get $\frac{\partial \ell}{\partial \mathbf{V}} = \mathbf{P}^T \frac{\partial \ell}{\partial \mathbf{O}}$. Then we have

$$\frac{\partial \ell}{\partial v_j} = \sum_i P_{ij} \frac{\partial \ell}{\partial o_i} = \sum_i \frac{e^{q_i^T k_j}}{L_i} \frac{\partial \ell}{\partial o_i} \tag{17}$$

where L_i has been computed in the forward pass. It is easy to get $\frac{\partial \ell}{\partial \mathbf{P}} = \frac{\partial \ell}{\partial \mathbf{O}} \mathbf{V}^T$.

$$\frac{\partial \ell}{\partial P_{ij}} = v_j^T \frac{\partial \ell}{\partial o_i}. \tag{18}$$

Given the softmax function $\mathbf{y} = \text{softmax}(\mathbf{x})$, we have the gradient $\frac{\mathbf{dy}}{\mathbf{dx}} = \text{diag}(\mathbf{y}) - \mathbf{y}\mathbf{y}^T$. Since $P_i = \text{softmax}(S_i)$, we have

$$\frac{\partial \ell}{\partial S_i} = \frac{\partial P_i}{\partial S_i} \frac{\partial \ell}{\partial P_i} = (\text{diag}(P_i) - P_i P_i^T) \frac{\partial \ell}{\partial P_i} = P_i \odot \frac{\partial \ell}{\partial P_i} - (P_i^T \frac{\partial \ell}{\partial P_i}) P_i \tag{19}$$

where $\odot$ denotes element-wise product. Define a scalar

$$D_i = P_i^T \frac{\partial \ell}{\partial P_i} = \sum_j P_{ij} \frac{\partial \ell}{\partial P_{ij}} = \sum_j \frac{e^{q_i^T k_j}}{L_i} v_j^T \frac{\partial \ell}{\partial o_i} = o_i^T \frac{\partial \ell}{\partial o_i}. \tag{20}$$

Algorithm 6: FlashAttention-2 Forward Pass

Require : Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$, block sizes B_c, B_r, softmax scaling constant $\tau \in \mathbb{R}$, masking function MASK, dropout probability p_{drop}.

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 $T_c = \lceil \frac{N}{B_c} \rceil$ 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 $\mathbf{O}_1, \dots, \mathbf{Q}_{T_r}$ of size $B_r \times d$ each, and divide the logsumexp L into $L_1, \dots, L_{T_r}$, of size B_r each.

for $i \leftarrow 1$ **to** T_r **do**

  Load $\mathbf{Q}_i$ from global memory to shared memory.

  Initialize $\mathbf{O}_i^{(0)} = (0) \in \mathbb{R}^{B_c \times d}, \ell_i^{(0)} = (0) \in \mathbb{R}^{B_r}, m_i^{(0)} = (-\infty) \in \mathbb{R}^{B_r}$ on shared memory.

  for $j \leftarrow 1$ **to** T_c **do**

    Load $\mathbf{K}_j, \mathbf{V}_j$ from global memory to shared memory.

    On chip, compute $\mathbf{S}_i^{(j)} = \tau \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.

    On chip, compute $\mathbf{S}_i^{(j)} \leftarrow \text{MASK}(\mathbf{S}_i^{(j)})$.

    On chip, compute $m_i^{(j)} = \max\left(m_i^{(j-1)}, \text{rowmax}(\mathbf{S}_i^{(j)})\right) \in \mathbb{R}^{B_r}$.

    On chip, compute $\tilde{\mathbf{P}}_i^{(j)} = \exp\left(\mathbf{S}_i^{(j)} - m_i^{(j)}\right) \in \mathbb{R}^{B_r \times B_c}$.

    On chip, compute $\ell_i^{(j)} = e^{m_i^{(j-1)} - m_i^{(j)}} \ell_i^{(j-1)} + \text{rowsum}(\tilde{\mathbf{P}}_i^{(j)}) \in \mathbb{R}^{B_r}$.

    On chip, compute dropout mask $\mathbf{Z}_i^{(j)}$ where each entry has value $\frac{1}{1-p_{\text{drop}}}$ with probability $1 - p_{\text{drop}}$ and value 0 with probability p_{drop}.

    On chip, compute $\tilde{\mathbf{P}}_i^{(j)} \leftarrow \tilde{\mathbf{P}}_i^{(j)} \odot \mathbf{Z}_i^{(j)}$ (pointwise multiply).

    On chip, compute $\mathbf{O}_i^{(j)} = \text{diag}\left(e^{m_i^{(j-1)} - m_i^{(j)}}\right) \mathbf{O}_i^{(j-1)} + \tilde{\mathbf{P}}_i^{(j)} \mathbf{V}_j$.

  On chip, compute $\mathbf{O}_i = \text{diag}\left(\ell_i^{(T_c)}\right)^{-1} \mathbf{O}_i^{(T_c)}$.

  On chip, compute $L_i = m_i^{(T_c)} + \ln\left(\ell_i^{(T_c)}\right)$.

  Write $\mathbf{O}_i$ to global memory as the i-th block of $\mathbf{O}$.

  Write L_i to global memory as the i-th block of L.

Return $\mathbf{O}$ and the logsumexp L.

Thus,

$$\frac{\partial \ell}{\partial S_{ij}} = (\frac{\partial \ell}{\partial P_{ij}} - D_i)P_{ij} = (v_j^T \frac{\partial \ell}{\partial o_i} - o_i^T \frac{\partial \ell}{\partial o_i}) \frac{e^{q_i^T k_j}}{L_i} = (v_j - o_i)^T \frac{\partial \ell}{\partial o_i} \frac{e^{q_i^T k_j}}{L_i}. \tag{21}$$

Furthermore,

$$\frac{\partial \ell}{\partial q_i} = \frac{\partial S_i}{\partial q_i} \frac{\partial \ell}{\partial S_i} = \mathbf{K}^T \frac{\partial \ell}{\partial S_i} = \sum_j \frac{\partial \ell}{\partial S_{ij}} k_j = \sum_j (v_j - o_i)^T \frac{\partial \ell}{\partial o_i} \frac{e^{q_i^T k_j}}{L_i} k_j \tag{22}$$

where the second equality follows from $S_i = (q_i^T \mathbf{K}^T)^T = \mathbf{K} q_i$. Then,

$$\frac{\partial \ell}{\partial k_j} = \frac{\partial S_{:,j}}{\partial k_j} \frac{\partial \ell}{\partial S_{:,j}} = \mathbf{Q}^T \frac{\partial \ell}{\partial S_{:,j}} = \sum_i \frac{\partial \ell}{\partial S_{ij}} q_i = \sum_i (v_j - o_i)^T \frac{\partial \ell}{\partial o_i} \frac{e^{q_i^T k_j}}{L_i} q_i. \tag{23}$$

In the backward pass of FlashAttention and FlashAttention-2, (see Algorithm 4 in Dao et al. (2022)), the blocks $\mathbf{dQ_i}$ are written to global memory in inner loop, which is inefficient. Alternatively, to compute $\mathrm{d}\mathbf{Q}$, we can swap i and j as follows.

$$\frac{\partial \ell}{\partial q_j} = \frac{\partial S_j}{\partial q_j} \frac{\partial \ell}{\partial S_j} = \mathbf{K}^T \frac{\partial \ell}{\partial S_j} = \sum_i \frac{\partial \ell}{\partial S_{ji}} k_i = \sum_i (v_i - o_j)^T \frac{\partial \ell}{\partial o_j} \frac{e^{q_j^T k_i}}{L_j} k_i. \tag{24}$$

This improvement has been described in Algorithm 7.

Now we summarize the above gradients in matrix form as the original paper does.

$$\mathbf{dV} = (\mathbf{P}^{\text{dropped}})^T \mathbf{dO} \tag{25}$$
$$\mathbf{dP}^{\text{dropped}} = \mathbf{dO}\mathbf{V}^T \tag{26}$$
$$\mathbf{dP} = \mathbf{dP}^{\text{dropped}} \odot \mathbf{Z} \tag{27}$$
$$\mathbf{dS}^{\text{masked}} = \mathbf{P} \odot \mathbf{dP} - \text{diag}(\mathbf{dP}\mathbf{P}^T)\mathbf{P} \tag{28}$$
$$= \mathbf{P} \odot \mathbf{dP} - \text{diag}(\mathbf{dO}\mathbf{V}^T\mathbf{P}^T)\mathbf{P} \tag{29}$$
$$= \mathbf{P} \odot \mathbf{dP} - \text{diag}(\mathbf{dO}\mathbf{O}^T)\mathbf{P} \tag{30}$$
$$\mathbf{dS} = \mathbf{dS}^{\text{masked}} \tag{31}$$
$$\mathbf{dQ} = \mathbf{dS}\mathbf{K} \tag{32}$$
$$\mathbf{dK} = \mathbf{dS}^T\mathbf{Q} \tag{33}$$

where $\mathbf{Z}$ denotes the dropout mask where has value $1/(1-p_{\text{drop}})$ with probability $1-p_{\text{drop}}$ and value 0 with probability p_{drop}. (31) follows from the fact that the gradient of a constant function is always 0.

7 IO complexity analysis

The standard attention needs to read $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N\times d}$ and write/read $\mathbf{S}, \mathbf{P} \in \mathbb{R}^{N\times N}$. Therefore, this incurs $\mathbf{\Theta}(Nd + N^2)$ global memory accesses.

Following Algorithm 6, we see that each element of $\mathbf{Q}$ will be loaded once. The elements of $\mathbf{K}$ and $\mathbf{V}$ will be loaded T_r passes. Hence, the total number of global memory accesses is $\mathbf{\Theta}(Nd + NdT_r) = \mathbf{\Theta}(NdT_r)$. Now we derive B_r and T_r for $\mathbf{Q}$ and $\mathbf{O}$.

$$B_r \times d = O(M) \Longleftrightarrow B_r = O(\frac{M}{d}) \tag{34}$$

Similarly, we have B_c and T_c for $\mathbf{K}, \mathbf{V}$ as follows.

$$B_c \times d = O(M) \Longleftrightarrow B_c = O(\frac{M}{d}) \tag{35}$$

Meanwhile, we need to write and read $\mathbf{S}_i^{(j)}$ and $\tilde{\mathbf{P}}_i^{(j)}$ on shared memory. Then,

$$B_c \times B_r = O(M) \Longleftrightarrow B_c = O(\frac{M}{B_r}) \tag{36}$$

Assume we set:

$$B_r = \mathbf{\Theta}(\frac{M}{d}), \qquad B_c = \mathbf{\Theta}(\min\{\frac{M}{d}, \frac{M}{B_r}\}) = \mathbf{\Theta}(\min\{\frac{M}{d}, d\}) \tag{37}$$

Then, we get

$$T_r = \frac{N}{B_r} = \mathbf{\Theta}(\frac{Nd}{M}) \tag{38}$$

Finally, we have

$$\mathbf{\Theta}(NdT_r) = \mathbf{\Theta}(\frac{N^2d^2}{M}). \tag{39}$$

Typically, $d^2 < M$. For example, each thread block has 48KB shared memory on A100 GPU. It can store 12K float32 values. A typical d is in $[64, 128]$. When $d = 64$, $d^2 \approx 4\text{K} <$ 12K.

References

Dao, T. (2023). FlashAttention-2: Faster attention with better parallelism and work partitioning.

Dao, T., Fu, D. Y., Ermon, S., Rudra, A., and Ré, C. (2022). FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems*.

Algorithm 7: FlashAttention-2 Backward Pass

Require : Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V}, \mathbf{dO} \in \mathbb{R}^{N \times d}$, vector $L \in \mathbb{R}^N$ in global memory, block sizes $B_c = B_r$, masking function MASK, dropout probability p_{drop}, pseudo-random number generator state $\mathcal{R}$ from the forward pass.

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 $T_c = \lceil \frac{N}{B_c} \rceil$ 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 $\mathbf{O}_1, \dots, \mathbf{Q}_{T_r}$ of size $B_r \times d$ each, divide $\mathbf{dO}$ into T_r blocks $\mathbf{dO}_1, \dots, \mathbf{dO}_{T_r}$, and divide L into $L_1, \dots, L_{T_r}$, of size B_r each.

Divide $\mathbf{dQ}$ into T_r blocks $\mathbf{dQ}_1, \dots, \mathbf{dQ}_{T_r}$ of size $B_r \times d$ each. Divide $\mathbf{dK}, \mathbf{dV} \in \mathbb{R}^{N \times d}$ into 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 $D = \text{rowsum}(\mathbf{dO} \odot \mathbf{O}) \in \mathbb{R}^N$ (pointwise multiply), write D to global memory and divide it into T_r blocks $D_1, \dots, D_{T_r}$ of size B_r each.

for $j \leftarrow 1$ **to** T_c **do**

  Load $\mathbf{K}_j, \mathbf{V}_j, \mathbf{dO}_j, L_j, D_j$ from global memory to shared memory.

  Initialize $\mathbf{dQ}_j = (0)_{B_c \times d}, \mathbf{dK}_j = (0)_{B_c \times d}, \mathbf{dV}_j = (0)_{B_c \times d}$ on shared memory.

  for $i \leftarrow 1$ **to** T_r **do**

    Load $\mathbf{Q}_i, \mathbf{K}_i, \mathbf{V}_i, \mathbf{dO}_i, L_i, D_i$ from global memory to shared memory.

    On chip, compute $\mathbf{S}_i^{(j)} = \text{MASK}(\tau \mathbf{Q}_i \mathbf{K}_j^T) \in \mathbb{R}^{B_r \times B_c}$.

    On chip, compute $\mathbf{S}_j^{(i)} = \text{MASK}(\tau \mathbf{Q}_j \mathbf{K}_i^T) \in \mathbb{R}^{B_c \times B_r}$.

    On chip, compute $\mathbf{P}_i^{(j)} = \exp(\mathbf{S}_i^{(j)} - L_i) \in \mathbb{R}^{B_r \times B_c}$.

    On chip, compute $\mathbf{P}_j^{(i)} = \exp(\mathbf{S}_j^{(i)} - L_j) \in \mathbb{R}^{B_c \times B_r}$.

    On chip, compute $\mathbf{dV}_j \leftarrow \mathbf{dV}_j + (\mathbf{P}_i^{(j)})^T \mathbf{dO}_i \in \mathbb{R}^{B_r \times d}$.

    On chip, compute $\mathbf{dP}_{i,\text{dropped}}^{(j)} = \mathbf{dO}_i \mathbf{V}_j^T \in \mathbb{R}^{B_r \times B_c}$.

    On chip, compute $\mathbf{dP}_{j,\text{dropped}}^{(i)} = \mathbf{dO}_j \mathbf{V}_i^T \in \mathbb{R}^{B_c \times B_r}$.

    On chip, compute $\mathbf{dP}_i^{(j)} = \mathbf{dP}_{i,\text{dropped}}^{(j)} \odot \mathbf{Z}_i^{(j)}$.

    On chip, compute $\mathbf{dP}_j^{(i)} = \mathbf{dP}_{j,\text{dropped}}^{(i)} \odot \mathbf{Z}_j^{(i)}$.

    On chip, compute $\mathbf{dS}_i^{(j)} = \mathbf{P}_i^{(j)} \odot (\mathbf{dP}_i^{(j)} - D_i) \in \mathbb{R}^{B_r \times B_c}$.

    On chip, compute $\mathbf{dS}_j^{(i)} = \mathbf{P}_j^{(i)} \odot (\mathbf{dP}_j^{(i)} - D_j) \in \mathbb{R}^{B_c \times B_r}$.

    On chip, compute $\mathbf{dQ}_j \leftarrow \mathbf{dQ}_j + \tau \mathbf{dS}_j^{(i)} \mathbf{K}_i \in \mathbb{R}^{B_c \times d}$.

    On chip, compute $\mathbf{dK}_j \leftarrow \mathbf{dK}_j + \tau (\mathbf{dS}_i^{(j)})^T \mathbf{Q}_i \in \mathbb{R}^{B_c \times d}$.

  Write $\mathbf{dQ}_j, \mathbf{dK}_j, \mathbf{dV}_j$ to global memory.

Return $\mathbf{dQ}, \mathbf{dK}, \mathbf{dV}$.

Milakov, M. and Gimelshein, N. (2018). Online normalizer calculation for softmax. *CoRR*, abs/1805.02867.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. u., and Polosukhin, I. (2017). Attention is all you need. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.