1 Flash Attention

The memory efficient attention refers to this paper [2]. The LSE trick refers to this blog [1].

1.1 The triton implementation

Softmax function is computed as follows:

$$\text{softmax} = \frac{\exp(x_i - c)}{\sum_i \exp(x_i - c)}$$

$$c = \max(x_1, \ldots, x_n)$$

The LogSumExp(LSE) is a smooth maximum.

$$\text{LSE}(x_1, \ldots, x_n) = \log(\exp(x_1) + \cdots + \exp(x_n))$$

$$= c + \log \sum_n \exp(x_i - c)$$

$$c = \max(x_1, \ldots, x_n)$$

lse_i, m_i and acc_o are accumulators.

$lse_i =$ ***-inf***

$m_i =$ ***-inf***: maximum up to the previous block. m_{ij} maximum up to the current block.

$acc\_o = \vec{\mathbf{0}}$.

Iterate (k, v) in K, V :

$$qk = \text{dot}(q, k) \tag{1}$$

$$m_{ij} = \max(\max(qk), lse_i) \tag{2}$$

$$p = \exp(qk - m_{ij}) \tag{3}$$

$$l_{ij} = \text{sum}(p) \tag{4}$$

// renormalize o

$$acc\_o\_scale = \exp(m_i - m_{ij}) \tag{5}$$

$$acc\_o = acc\_o\_scale * acc\_o \tag{6}$$

$$acc\_o = acc\_o + \text{dot}(p, v) \tag{7}$$

// update statistics

$$m_i = m_{ij} \tag{8}$$

$$l_{i\_new} = \exp(lse_i - m_{ij}) + l_{ij} \tag{9}$$

$$lse_i = m_{ij} + \log(l_{i\_new}) \tag{10}$$

// o_scale is the denominator of the softmax function

$$o\_scale = \exp(m_i - lse_i) \tag{11}$$

$$acc\_o = acc\_o * o\_scale \tag{12}$$

In eq. (2) lse_i is used as the approximation of max.

The first component of the right hand side of eq. (9):

$$\begin{aligned}
\exp\left(lse_i - m_{ij}\right) &= \exp\left(c_{old} + \log\sum_n \exp(x_i - c_{old}) - c_{new}\right) + l_{ij} \\
&= \exp\left((c_{old} - c_{new}) + \log\sum_n \exp(x_i - c_{old})\right) \\
&= \exp\left(c_{old} - c_{new}\right)\sum_n \exp(x_i - c_{old})
\end{aligned}$$

o_scale in equation eq. (11) is the denominator of the softmax function.

$$\begin{aligned}
m_i - lse_i &= \exp\left(c - c - \log\sum_n \exp(x_i - c)\right) \\
&= \exp\left(-\log\sum_n \exp(x_i - c)\right) \\
&= \exp\left(\log\frac{1}{\sum_n \exp(x_i - c)}\right) \\
&= \frac{1}{\sum_n \exp(x_i - c)}
\end{aligned}$$

1.2 How was Flash Attention's Formula Derived

$$\text{Attention}(Q, K, V) = \text{softmax}(QK^T)V \tag{13}$$

where $Q, K, V \in \mathbb{R}^{N \times d}$ are sequences of vectors.

We expect that the **entire** computational process of equation (13) can proceed block by block, with each block computing a portion of the final result. If the block size is chosen appropriately, the entire computational process can be kept in high-speed memory. We expect the computational process to proceed as follows:

```
// stage1: local reducer computes partial results on individual blocks
o1 = Attention(q, ks1, vs1)      o2 = Attention(q, ks2, vs2)

// stage2: combine partial results
o_new = Combiner(o1, o2)
```

During the first stage, each local reducer computes a partial result on an individual block; during the second stage, these partial results are combined to obtain the correct final result. Thus, the problems are:

1. Under what conditions can the entire computational process be decomposed into these two stages, and is there a combiner that can compute the correct final result?
2. If such a combiner exists, how can it be constructed

Let's see how was Flash Attention's formula derived.

$q \in \mathbb{R}^d$ is a vector (the smallest block of Q). $ks_1, ks_2, vs_1, vs_2 \in \mathbb{R}^{B \times d}$ are sequences of vectors. They are blocks of K and V. B are block size.

$$\begin{aligned}
a_1 &= \text{dot}(q, ks_1) & a_2 &= \text{dot}(q, ks_2) \\
b_1 &= \max(-\inf, a_1) & b_2 &= \max(-\inf, a_2) \\
c_1 &= a_1 - b_1 & c_2 &= a_2 - b_2 \\
d_1 &= \exp(c_1) & d_2 &= \exp(c_2) \\
e_1 &= \text{sum}(0, d_1) & e_2 &= \text{sum}(0, d_2) \\
f_1 &= \frac{d_1}{e_1} & f_2 &= \frac{d_1}{e_1} \\
g_1 &= f_1 * vs_1 & g_2 &= f_2 * vs_2 \\
o_1 &= \text{sum}(0, g_1) & o_2 &= \text{sum}(0, g_2)
\end{aligned}$$

$b = \max(b_1, b_2)$, $\triangle c_1 := b_1 - b$ $\triangle c_2 := b_2 - b$

$$\begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ o \end{bmatrix} = \begin{bmatrix} a_1 \\ b_1 \\ c_1 \\ d_1 \\ e_1 \\ f_1 \\ g_1 \\ o_1 \end{bmatrix} \oplus \begin{bmatrix} a_2 \\ b_2 \\ c_2 \\ d_2 \\ e_2 \\ f_2 \\ g_2 \\ o_2 \end{bmatrix} = \begin{bmatrix} [a_1 : a_2] \\ \max(b_1, b_2) \\ [c_1 + \triangle c_1 : c_2 + \triangle c_2] \\ [d_1 \exp(\triangle c_1) : d_2 \exp(\triangle c_1)] \\ e_1 \exp(\triangle c_1) + e_2 \exp(\triangle c_2) \\ \left[\frac{\exp(c_1 + \triangle c_1)}{e_1 \exp(\triangle c_1) + e_2 \exp(\triangle c_2)} : \frac{\exp(c_2 + \triangle c_2)}{e_1 \exp(\triangle c_1) + e_2 \exp(\triangle c_2)}\right] \\ \left[\frac{\exp(c_1 + \triangle c_1) * vs_1}{e_1 \exp(\triangle c_1) + e_2 \exp(\triangle c_2)} : \frac{\exp(c_2 + \triangle c_2) * vs_2}{e_1 \exp(\triangle c_1) + e_2 \exp(\triangle c_2)}\right] \\ \frac{\exp(c_1 + \triangle c_1) * vs_1}{e_1 \exp(\triangle c_1) + e_2 \exp(\triangle c_2)} + \frac{\exp(c_2 + \triangle c_2) * vs_2}{e_1 \exp(\triangle c_1) + e_2 \exp(\triangle c_2)} \end{bmatrix}$$

$$
\begin{aligned}
e_1' &= \text{sum}(0, d_1') \\
&= \text{sum}(0, d_1 \exp(\triangle c_1)) \\
&= \text{sum}(0, d_1) \exp(\triangle c_1) \\
&= e_1 \exp(\triangle c_1)
\end{aligned}
$$

$$
\begin{aligned}
o =& \frac{\exp(c_1 + \triangle c_1) * vs_1}{e_1 \exp(\triangle c_1) + e_2 \exp(\triangle c_2)} + \frac{\exp(c_2 + \triangle c_2) * vs_2}{e_1 \exp(\triangle c_1) + e_2 \exp(\triangle c_2)} \\
=& \frac{\exp(\triangle c_1)}{e_1 \exp(\triangle c_1) + e_2 \exp(\triangle c_2)} \exp(c_1) * vs_1 + \frac{\exp(\triangle c_2)}{e_1 \exp(\triangle c_1) + e_2 \exp(\triangle c_2)} \exp(c_2) * vs_2
\end{aligned}
$$

$$
\begin{aligned}
o_1 &= f_1 * vs_1 \\
o_1 * e_1 &= f_1 * e_1 * vs_1 \\
o_1 * e_1 &= d_1 * vs_1 \\
o_1 * e_1 &= \exp(c_1) * vs_1
\end{aligned}
$$

Therefore, we have:

$$
o = \frac{\exp(\triangle c_1)}{e_1 \exp(\triangle c_1) + e_2 \exp(\triangle c_2)} o_1 * e_1 + \frac{\exp(\triangle c_2)}{e_1 \exp(\triangle c_1) + e_2 \exp(\triangle c_2)} o_2 * e_2 \tag{14}
$$

Equation (14) specifies how to combine the partial results of a local reducer. Suppose b_1, e_1, and o_1 are the outputs of another local reducer. Given a new block, the current local reducer first computes b_2, o_2, and e_2. The combiner then computes $b = \max(b_1, b_2)$, we set $\triangle c_1 := b_1 - b$, and $\triangle c_2 := b_2 - b$. Finally, use equation (14) to obtain a combined result.

1.3 Replace softmax with logsoftmax

$\mathbf{x} \in \mathbb{R}^n$ is a vector. $m(\mathbf{x}) := \max_i(\mathbf{x})$

$$
\begin{aligned}
\operatorname{logsoftmax}(\mathbf{x}) &= \log(\operatorname{softmax}(\mathbf{x})) \\
&= \log \frac{\exp(x_i - m(\mathbf{x}))}{\sum_i \exp(x_i - m(\mathbf{x}))} \\
&= x_i - m(\mathbf{x}) - \log\left(\sum_i \exp(x_i - m(\mathbf{x}))\right)
\end{aligned}
$$

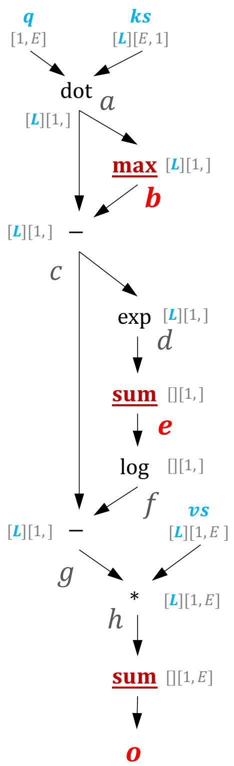


Figure 1: The expression tree of logsoftmax.

$$
\begin{aligned}
a_1 &= \text{dot}(q, ks_1) & a_2 &= \text{dot}(q, ks_2) \\
b_1 &= \text{max}(-\inf, a_1) & b_2 &= \text{max}(-\inf, a_2) \\
c_1 &= a_1 - b_1 & c_2 &= a_2 - b_2 \\
d_1 &= \exp(c_1) & d_2 &= \exp(c_2) \\
e_1 &= \text{sum}(0, d_1) & e_2 &= \text{sum}(0, d_2) \\
f_1 &= \log e_1 & f_2 &= \log e_2 \\
g_1 &= c_1 - f_1 & g_2 &= c_2 - f_2 \\
h_1 &= g_1 * vs_1 & h_2 &= g_2 * vs_2 \\
o_1 &= \text{sum}(0, h_1) & o_2 &= \text{sum}(0, h_2)
\end{aligned}
$$

Equatioins in red are partial results that require a further aggregation. Equations that consume results of these partial results requires updates.

$$
\begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \\ o \end{bmatrix} = \begin{bmatrix} a_1 \\ b_1 \\ c_1 \\ d_1 \\ e_1 \\ f_1 \\ g_1 \\ h_1 \\ o_1 \end{bmatrix} \bigoplus \begin{bmatrix} a_2 \\ b_2 \\ c_2 \\ d_2 \\ e_2 \\ f_2 \\ g_2 \\ h_2 \\ o_2 \end{bmatrix} =?
$$

$[x_1 : x_2]$ stands for concatenate x_1 and x_2 into a longer sequence.

$$
\begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \\ o \end{bmatrix} = \begin{bmatrix} a_1 \\ b_1 \\ c_1 \\ d_1 \\ e_1 \\ f_1 \\ g_1 \\ h_1 \\ o_1 \end{bmatrix} \bigoplus \begin{bmatrix} a_2 \\ b_2 \\ c_2 \\ d_2 \\ e_2 \\ f_2 \\ g_2 \\ h_2 \\ o_2 \end{bmatrix} = \begin{bmatrix} [a_1 : a_2] \\ \text{max}(b_1, b_2) \\ [c_1 + (b_1 - b) : c_2 + (b_2 - b)] \\ [d_1 \exp(b_1 - b) : d_2 \exp(b_2 - b)] \\ e_1 \exp(b - b_1) + e_2 \exp(b_2 - b) \\ \log\left(e_1 \exp(b - b_1) + e_2 \exp(b_2 - b)\right) \\ [c_1 + (b_1 - b) - f : c_2 + (b_2 - b) - f] \\ [(c_1 + (b_1 - b) - f) * vs_1 : (c_2 + (b_2 - b) - f) * vs_2] \\ (c_1 + (b_1 - b) - f) * vs_1 + (c_2 + (b_2 - b) - f) * vs_2 \end{bmatrix}
$$

$$
\begin{aligned}
\triangle c_1 &:= b_1 - b \\
\triangle c_2 &:= b_2 - b \\
f &= \log(e_1 \exp(\triangle c_1) + e_2 \exp(\triangle c_1))
\end{aligned}
$$

$$
\begin{aligned}
o &= o_1' + o_2' \\
&= (c_1 + \triangle c_1 - f) * vs_1 + (c_2 + \triangle c_2 - f) * vs_2 \\
&= (c_1 + \triangle c_1 - \log(e_1 \exp(\triangle c_1) + \exp(\triangle c_2))) * vs_1 + (c_2 + \triangle c_2 - \log(e_1 \exp(\triangle c_1) + \exp(\triangle c_2))) * vs_2
\end{aligned}
$$

2 Online Normalized Softmax

```
Input:  X : [M × N], m_0 : [M], d_0[M]
Output:  Y : [M × N]

m_0 ← −inf   // accumulator
d_0 ← 0   // accumulator
for  i ← 1, M  do   // parallel for, maps to blocks
  for  j ← 1, N  do   // reduce, maps for threads in a thread block
      m_j[i] ← max(m_{j−1}[i], X[i, j])
      d_j[i] ← d_{j−1}[i] × e^{m_{j−1}[i] − m_j[i]} + e^{X[i,j] − m_j[i]}
  endfor

  for  j ← 1, N  do   // parallel for
    Y[i, j] ← e^{X[i,j] − m[i]} / d[i]
  endfor
endfor
```

2.1 Fused expression: version 1

m,L,o are accumulators.

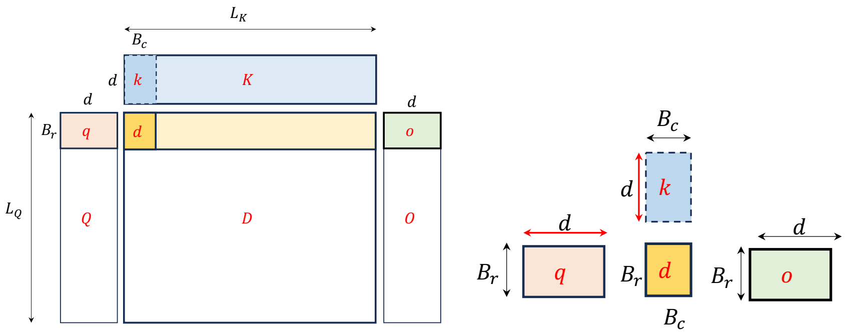
```
for q in Q: // for loops outside the compute kernel
  for k,v in K,V: // for loops outside the compute kernel
    m,L,o = compute_kernel(q, k, v, m, L, o)
```

Input: $q, k, v, m_{old}, L_{old}, o_{old}$ (q,k,v are small tiles of Q,K,V);

$$
\begin{aligned}
x_1 &= \text{dot}(q, k) \\
x_2 &= \text{max}(x_1) \\
x_3 &= \text{exp}(x_1 - x_2) \\
x_4 &= \text{sum}(x_3) \\
x_5 &= \text{dot}(x_3, v) \\
m_{new} &= \text{max}(m_{old}, x_2) \quad \text{// new partial max value} \\
x_6 &= \text{exp}(m_{old} - m_{new}) \\
x_7 &= \text{exp}(x_2 - m_{new}) \\
L_{new} &= x_6 * o_{old} + x_7 * x_4 \quad \text{// new partial sum} \\
o_{new} &= \frac{o_{old} * L_{old} * x_6 + x_7 * x_3}{L_{new}} \quad \text{// new partial o}
\end{aligned}
$$

return $m_{new}, L_{new}, o_{new}$

3 CTA Offset



(a) Data accessed by a CTA.

(b) Data accessed by one iteration inside the kernel.

```
// ------   Step 1: each CTA gets its input data.   ------
Q_offset = blockIdx.x * d * B_r   // offset
D_offset = blockIdx.x/N * B_r * L_K   // offset
K_offset = blockIdx.x/T_r * d * L_K   // offset

Q = Qs[Q_offset]   // move the pointer
O = Os[Q_offset]   // move the pointer

K = Ks[K_offset] // move the pointer
V = Vs[K_offset] // move the pointer
D = Ds[D_offset] // move the pointer
for(int i = 0; i < T_c; ++i) {
  k_offset = i * B_c * d // offset
  k = K[k_offset]   // move the pointer

  d_offset = i * B_r * B_c // offset
  d = D[d_offset]   // move the pointer



  // ------   thread in the CTA offset gets its input   ------
}
```

Notations:

- N stands for batch size, H stands for head number, L_Q, L_K and L_V stands for Q, K, V lengths respectively.
- $\mathbf{Qs}$, $\mathbf{Ks}$, $\mathbf{Vs}$ stands for the whole input matrix, that has a shape of: $[N * H * L_Q, d]$, $[N * H * L_K, d]$, $[N * H * L_Q, d]$ respectively.
- $T_r = \frac{L_Q}{B_r}$: how many blocks along the row dimension.
- $T_c = \frac{L_K}{B_c}$: how many blocks along the column dimension.
- Q,K,V stands for inputs to a CTA.
- q,k,v stands for a tile.

A single thread block (CTA) reads an orange block of Q; A blue block of K and V (K and V are repeatedly load from global memory T_r times). Inside a CTA, a sequential *for* loop iterates over dark blue blocks of K and V. green block of V.

Launch configuration: $L_Q/B_r * H * N$ blocks are required.

Every T_r blocks repeatedly load K and V.

Input: $\mathbf{Qs}, \mathbf{Ks}, \mathbf{Vs}$

4 I/O Complexity Analysis

4.1 self-implemented softmax

- $\mathbf{G}_{\mathrm{MN}}$ stands for access global memory $M \times N$ times.
- $\mathbf{S}_{\mathrm{MN}}$ stands for access shared memory $M \times N$ times.

Step No.	Pattern	Expression	Input	Output
1	Reduction	$t_0 = \max(\mathbf{V})$	$\mathbf{G}_{MN}$	$\mathbf{S}_{\mathbf{M}}$
2	Broadcast	$t_1 = \exp(\mathbf{V} - t_0)$	$\mathbf{G}_{MN}+S_M$	$\mathbf{G}_{MN}$
3	Reduction	$t_2 = \text{sum}(t_1)$	$\mathbf{G}_{MN}$	S_M
4	Broadcast	$\mathbf{O} = \frac{t_1}{t_2}$	$\mathbf{G}_{MN}+S_M$	$\mathbf{G}_{MN}$
		Softmax	$4\mathbf{G}_{MN}+2S_M$	$2\mathbf{G}_{MN}+2S_M$

leads to $6\mathbf{G}_{MN}$ global memory access in total.

4.2 self-implemented online-softmax

- $\mathbf{G}_{\mathrm{MN}}$ stands for access global memory $M \times N$ times.
- $\mathbf{S}_{\mathrm{MN}}$ stands for access shared memory $M \times N$ times.

Step No.	Pattern	Expression	Input	Output
1	Reduction	compute m and do	$\mathbf{G}_{MN}$	$2\mathbf{S}_M$
2	Broadcast	rescale	$\mathbf{G}_{MN}+2S_M$	$\mathbf{G}_{MN}$
		Online-Softmax	$2\mathbf{G}_{MN}+2S_M$	$\mathbf{G}_{MN}+2S_M$

leads to $3\mathbf{G}_{MN}$ global memory access in total.

4.3 Cub softmax

- $\mathbf{G}_{\mathrm{MN}}$ stands for access global memory $M \times N$ times.
- $\mathbf{S}_{\mathrm{MN}}$ stands for access shared memory $M \times N$ times.

Step No.	Pattern	Expression	Input	Output
1	Reduction	$t_0 = \max(\mathbf{V})$	$\mathbf{G}_{MN}$	$\mathbf{S}_{\mathbf{M}}$
2	Elementwise	$t_1 = \exp(\mathbf{V} - t_0)$	$\mathbf{G}_{MN}+S_M$	$\mathbf{G}_{MN}$
3	Reduction	$t_2 = \text{sum}(t_1)$	$\mathbf{G}_{MN}+S_M$	$\mathbf{S}_M$
3	Broadcast	$\mathbf{O} = \frac{\exp(\mathbf{V}-t_0)}{t_2}$	$\mathbf{G}_{MN}+2S_M$	$\mathbf{G}_{MN}$
		Softmax	$4\mathbf{G}_{MN}+4S_M$	$2\mathbf{G}_{MN}+2S_M$

leads to $6\mathbf{G}_{MN}$ global memory access in total.

4.4 Cub online softmax

- $\mathbf{G_{MN}}$ stands for access global memory $M \times N$ times.
- $\mathbf{S_{MN}}$ stands for access shared memory $M \times N$ times.

Step No.	Pattern	Expression	Input	Output
1	Reduction	compute m and do	$\mathbf{G}_{MN}$	$2\mathbf{S}_M$
2	Broadcast	rescale	$\mathbf{G}_{MN}$+$2S_M$	$\mathbf{G}_{MN}$
		Online-Softmax	$2\mathbf{G}_{MN}$+$2S_M$	$\mathbf{G}_{MN}$+$2S_M$

leads to $3\mathbf{G}_{MN}$ global memory access in total.

5 Fuse Consecutive Aggregations

5.1 Background

A reduce function processes data from a list. It has the following standard form:

```
function R(I : T_1, X : List[T_2]) → T_1
  s = I
  foreach(x_i in X)
    s = A(I, x_i)
  return F(s)
```

where I is the initializer , $A(s, x)$ is the user-defined accumulator, and $F(s)$ is the finalizer. s is a set of solution variables that store a partial solution, and are initialized to the initializer I.

Let's consider a computational pattern shown below. X, Y and Z are lists that have the same length. R_1 and R_2 are user-defined reduce functions. G is user-defined element-wise (broadcast is also regarded as an element-wise function) function. There are data dependence (the producer-consumer relation) between R_1, G, and R_2. They have to be executed sequentially.

```
function R_1(I_1 : T_1, X : List[T_2]) → T_1
  s_1 = I_1
  foreach(x_i in X):
    s_1 = A_1(I_1, x_i)
  return F_1(s_1)

function G(Z : List[T_1]) → List[T_2]  // element-wise operations
  foreach(z_i in Z):
    t_i = g(I_2, z_i)
  return T  // T = [..., t_i, ...]

function R_2(I_2 : T_1, Y : List[T_2]) → T_1
  s_2 = I_2
  foreach(y_i in Y):
    s_1 = A_2(I_2, y_i)
  return F(s_2)

s_1 = R_1(I_1, X)  // reducer 1
T = G(s_1, Y, ...)  // mapper, some element-wise function
s_2 = R_2(I_2, T)  // reducer 2
```

Listing 1: Consecutive reduce functions.

5.2 Motivation and Goal

In the computational pattern above, reducer R_1's output s_1 has data dependence on each element of list X. Mapper G consume s_1 and prodce a new list T.

reducer R_2's output s_2 has data dependence on each element of list T.

The storage of T might be very large.

The goal is to get a fused reduce function $s_1, s_2 = \bar{R}(I_1, I_2, s_1', s_2', X, Y)$ from R_1, R_2 and G.

```
function R̄(p_1 : T_1, p_2 : T_2, I_1 : T_1, I_2 : T_2, X : List[T_3], Y : List[T_4])
  s_1 = I_1
  s_2 = I_2
  foreach(x_i, y_1 in zip(X, Y)):
    s'_1, s'_2 = Ā(s_1, s_2, x_i, y_i,)  // local aggregate
    s_1, s_2 = C(s'_1, s'_2, p_1, p_2)  // combine
  return s_1, s_2
```

$\bar{A}$ is straightforward. It is constructed as follows:

$$
\begin{aligned}
s'_1, s'_2 &= \bar{A}(I_1, I_2, s_1, s_2, x_i, y_i) \\
s'_1 &= R_1(I_1, s_1, x_i) \\
s'_2 &= R_2(I_2, s_2, y_i)
\end{aligned}
$$

The problem is how to get C.

References

[1] Gregory Gundersen. The log-sum-exp trick. https://gregorygundersen.com/blog/2020/02/09/log-sum-exp/, 2020. [Online; accessed 17-April-2023].

[2] Markus N. Rabe and Charles Staats. Self-attention Does Not Need O(n^2) Memory. *CoRR*, abs/2112.05682, 2021.