From Online Softmax to FlashAttention

BY ZIHAO YE

Email: zhye@cs.washington.edu

May 11, 2023

UW CSE 599M Spring 2023: ML for ML Systems

The key innovation of FlashAttention [1] is using an idea similar to Online Softmax [3] to tile the self-attention computation, so that we can fuse the entire multi-head attention layer without accessing GPU global memory for intermediate logits and attention scores. In this note I'll briefly explain why tiling self-attention computation is non-trivial, and how to derive FlashAttention computation from online softmax trick.

We thank Andrew Gu for proofreading this note.

1 The Self-Attention

The computation of Self-Attention can be summarized as (we ignore heads and batches because computation on these dimensions are fully parallel, we also omit details such as attention masks and scale factor $\frac{1}{\sqrt{D}}$ for simplicity):

$$O = \text{softmax}(Q\,K^T)\,V \tag{1}$$

where Q, K, V, O are both 2D matrix with shape (L, D), where L is the sequence length and D is the dimension per head (a.k.a. head dimension), the softmax applies to the last dimension (columns).

The standard approach to computing self-attention is to factorize the computation into several stages:

$$X = Q\,K^T \tag{2}$$
$$A = \text{softmax}(X) \tag{3}$$
$$O = A\,V \tag{4}$$

We call X matrix the pre-softmax logits, A matrix the attention score and O matrix the output.

One amazing fact about FlashAttention is that we don't need to materialize X and A matrices on global memory, instead we fuse the entire computation in formula 1 in a single CUDA kernel. This requires us to design an algorithm that carefully manages on-chip memory (like stream algorithms) because NVIDIA GPU's shared memory is small.

For classical algorithms such as matrix multiplication, tiling is used to ensure that on-chip memory does not exceed hardware limits. Figure 1 provides an example of this. During kernel execution, only $3T^2$ elements are stored on-chip, regardless of the matrix shape. This tiling approach is valid because addition is associative, allowing for the decomposition of the entire matrix multiplication into the sum of many tile-wise matrix multiplications.

However, Self-Attention includes a softmax operator that is not directly associative, making it hard to simply tile Self-Attention like in Figure 1. Is there a way to make softmax associative?

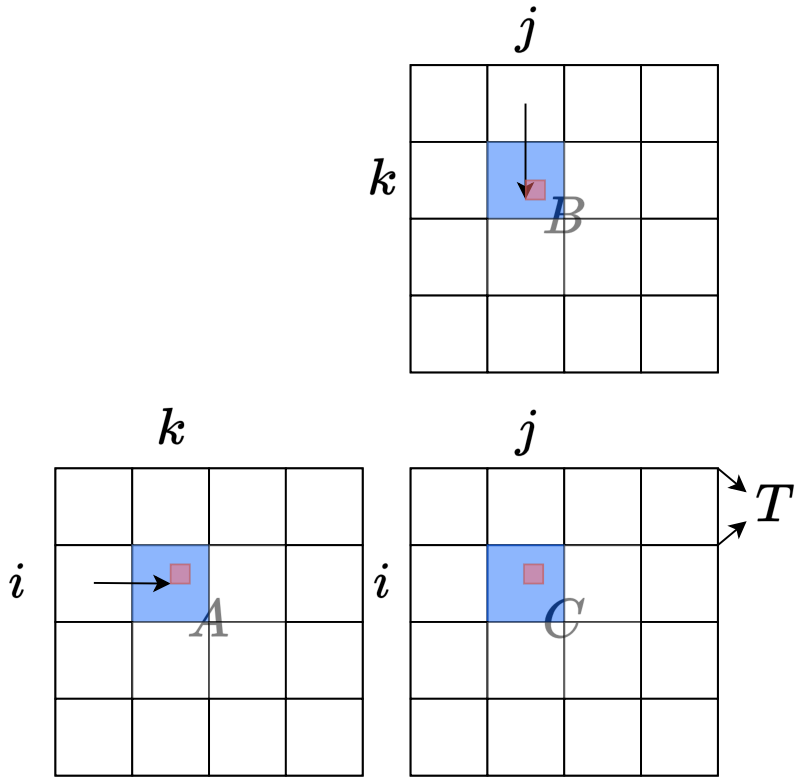

Figure 1. The figure about briefly explains how to tile input and output matrices for matrix multiplication $C = A \times B$, the matrices are partitioned to $T \times T$ tiles. For each output tile, we sweep the related tiles in A for left to right, and related tiles in B for top to down, and load the values from global memory to on-chip memory (colored in blue, the overall on-chip memory footprint is $O(T^2)$). For tile-wise partial matrix multiplication, for position (i, j), we load $A[i, k]$ and $B[k, j]$ (colored in red) for all k inside the tile from on-chip memory, then aggregate $A[i, k] \times B[k, j]$ to $C[i, j]$ in on-chip memory. After the commputation of a tile is complete, we write the on-chip C tile back to main memory and move on to the processing the next tile.

Tiling in real world application is much more complicated, you can refer to Cutlass implementation of matrix multiplication on A100 [2].

2 (Safe) Softmax

Let's recall the softmax operator first, below is the generic formula of softmax computation:

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

Note that x_i might be very large and e^{x_i} can easily overflow. For instance, the maximum number that float16 can support is 65536, which means that for $x \geqslant 11$, e^x would exceed the effective range of float16.

To mitigate this issue, mathematical software often employs a trick known as the "safe" softmax:

$$\frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}} = \frac{e^{x_i - m}}{\sum_{j=1}^{N} e^{x_j - m}} \tag{6}$$

where $m = \max_{j=1}^{N}(x_j)$, so that we can confirm each $x_i - m \leqslant 0$, which is safe because the exponential operator is accurate for negative inputs.

Then we can summarize the computation of safe softmax as the following 3-pass algorithm:

Algorithm 3-pass safe softmax

Notations
$\{m_i\}$: $\max_{j=1}^{i}\{x_j\}$, with initial value $m_0 = -\infty$.
$\{d_i\}$: $\sum_{j=1}^{i} e^{x_j - m_N}$, with initial value $d_0 = 0$, d_N is the denominator of safe softmax.
$\{a_i\}$: the final softmax value.

Body
for $i \leftarrow 1, N$ **do**

$$m_i \leftarrow \max\left(m_{i-1}, x_i\right) \tag{7}$$

end
for $i \leftarrow 1, N$ **do**

$$d_i \leftarrow d_{i-1} + e^{x_i - m_N} \tag{8}$$

end
for $i \leftarrow 1, N$ **do**

$$a_i \leftarrow \frac{e^{x_i - m_N}}{d_N} \tag{9}$$

end

This algorthm requires us to iterate over $[1, N]$ for 3 times. In the context of self-attention in Transformer, the $\{x_i\}$ are pre-softmax logits computed by QK^T. This means if we don't all logits $\{x_i\}_{i=1}^{N}$ (we don't have large enough SRAM to fit all of them), we need to access Q and K three times (to re-compute logits on-the-fly), which is not I/O efficient.

3 Online Softmax

If we fuse the equation 7, 8 and 9 in a single loop, we can reduce the global memory access time from 3 to 1. Unfortunately, we cannot fuse equation 7 and 8 in the same loop because 8 depends on m_N, which cannot be determined until the first loop completes.

We can create another sequence $d'_i := \sum_{j=1}^{i} e^{x_j - m_i}$ as a surrogate for original sequence $d_i := \sum_{j=1}^{i} e^{x_j - m_N}$ to remove the dependency on N, and the N-th term of these two sequence are identical: $d_N = d'_N$, thus we can safely replace d_N in equation 9 with d'_N. We can also find a recurrence relation between d'_i and d'_{i-1}:

$$\begin{aligned}
d'_i &= \sum_{j=1}^{i} e^{x_j - m_i} \\
&= \left(\sum_{j=1}^{i-1} e^{x_j - m_i}\right) + e^{x_i - m_i} \\
&= \left(\sum_{j=1}^{i-1} e^{x_j - m_{i-1}}\right) e^{m_{i-1} - m_i} + e^{x_i - m_i} \\
&= d'_{i-1}\, e^{m_{i-1} - m_i} + e^{x_i - m_i}
\end{aligned} \tag{10}$$

This recurrent form only relies on m_i and m_{i-1}, and we can compute m_j and d'_j together in the same loop:

Algorithm 2-pass online softmax

for $i \leftarrow 1, N$ **do**

$$\begin{aligned} m_i &\leftarrow \max\left(m_{i-1}, x_i\right) \\ d'_i &\leftarrow d'_{i-1}\, e^{m_{i-1}-m_i} + e^{x_i - m_i} \end{aligned}$$

end
for $i \leftarrow 1, N$ **do**

$$a_i \leftarrow \frac{e^{x_i - m_N}}{d'_N}$$

end

This is the algorithm proposed in Online Softmax paper [3]. However, it still requires two passes to complete the softmax calculation, can we reduce the number of passes to 1 to minimize global I/O?

4 FlashAttention

Unfortunately, the answer is "no" for softmax, but in Self-Attention, our final target is not the attention score matrix A, but the O matrix which equals $A \times V$. Can we find a one-pass recurrence form for O instead?

Let's try to formulate the k-th row (the computation of all rows are independent, and we explain the computation of one row for simplicity) of Self-Attention computation as recurrence algorithm:

Algorithm Multi-pass Self-Attention

NOTATIONS
$Q[k,:]$: the k-th row vector of Q matrix.
$K^T[:,i]$: the i-th column vector of K^T matrix.
$O[k,:]$: the k-th row of output O matrix.
$V[i,:]$: the i-th row of V matrix.
$\{\boldsymbol{o}_i\}$: $\sum_{j=1}^{i} a_j V[j,:]$, a row vector storing partial aggregation result $A[k,:i] \times V[:i,:]$

BODY
for $i \leftarrow 1, N$ **do**

$$\begin{aligned} x_i &\leftarrow Q[k,:]\, K^T[:,i] \\ m_i &\leftarrow \max\left(m_{i-1}, x_i\right) \\ d'_i &\leftarrow d'_{i-1}\, e^{m_{i-1}-m_i} + e^{x_i - m_i} \end{aligned}$$

end
for $i \leftarrow 1, N$ **do**

$$a_i \leftarrow \frac{e^{x_i - m_N}}{d'_N} \tag{11}$$

$$\boldsymbol{o}_i \leftarrow \boldsymbol{o}_{i-1} + a_i V[i,:] \tag{12}$$

end

$$O[k,:] \leftarrow \boldsymbol{o}_N$$

Let's replace the a_i in equation 12 by its definition in equation 11:

$$\boldsymbol{o}_i := \sum_{j=1}^{i} \left(\frac{e^{x_j - m_N}}{d'_N} V[j,:] \right) \tag{13}$$

This still depends on m_N and d_N which cannot be determined until the previous loop completes. But we can play the "surrogate" trick in section 3 again, by creating a surrogate sequence $\boldsymbol{o}'$:

$$\boldsymbol{o}'_i := \left(\sum_{j=1}^{i} \frac{e^{x_j - m_i}}{d'_i} V[j,:] \right)$$

The n-th element of $\boldsymbol{o}$ and $\boldsymbol{o}'$ are the identical: $\boldsymbol{o}'_N = \boldsymbol{o}_N$, and we can find a recurrence relation between $\boldsymbol{o}'_i$ and $\boldsymbol{o}'_{i-1}$:

$$\begin{aligned}
\boldsymbol{o}'_i &= \sum_{j=1}^{i} \frac{e^{x_j - m_i}}{d'_i} V[j,:] \\
&= \left(\sum_{j=1}^{i-1} \frac{e^{x_j - m_i}}{d'_i} V[j,:] \right) + \frac{e^{x_i - m_i}}{d'_i} V[i,:] \\
&= \left(\sum_{j=1}^{i-1} \frac{e^{x_j - m_{i-1}}}{d'_{i-1}} \frac{e^{x_j - m_i}}{e^{x_j - m_{i-1}}} \frac{d'_{i-1}}{d'_i} V[j,:] \right) + \frac{e^{x_i - m_i}}{d'_i} V[i,:] \\
&= \left(\sum_{j=1}^{i-1} \frac{e^{x_j - m_{i-1}}}{d'_{i-1}} V[j,:] \right) \frac{d'_{i-1}}{d'_i} e^{m_{i-1} - m_i} + \frac{e^{x_i - m_i}}{d'_i} V[i,:] \\
&= \boldsymbol{o}'_{i-1} \frac{d'_{i-1} e^{m_{i-1} - m_i}}{d'_i} + \frac{e^{x_i - m_i}}{d'_i} V[i,:]
\end{aligned} \tag{14}$$

which only depends on d'_i, d'_{i-1}, m_i, m_{i-1} and x_i, thus we can fuse all computations in Self-Attention in a single loop:

Algorithm FlashAttention

for $i \leftarrow 1, N$ **do**

$$\begin{aligned}
x_i &\leftarrow Q[k,:]\, K^T[:,i] \\
m_i &\leftarrow \max(m_{i-1}, x_i) \\
d'_i &\leftarrow d'_{i-1} e^{m_{i-1} - m_i} + e^{x_i - m_i} \\
\boldsymbol{o}'_i &\leftarrow \boldsymbol{o}'_{i-1} \frac{d'_{i-1} e^{m_{i-1} - m_i}}{d'_i} + \frac{e^{x_i - m_i}}{d'_i} V[i,:]
\end{aligned}$$

end

$$O[k,:] \leftarrow \boldsymbol{o}'_N$$

The states x_i, m_i, d'_i, and $\boldsymbol{o}'_i$ have small footprints that can easily fit into GPU shared memory. Because all operations in this algorithm are associative, it is compatible with tiling. If we compute the states tile-by-tile, the algorithm can be expressed as follows:

Algorithm FlashAttention (Tiling)

New Notations
b: the block size of the tile
#tiles: number of tiles in the row, $N = b \times$ #tiles.
$\boldsymbol{x}_i$: a vector storing the $Q[k]\, K^T$ value of the i-th tile $[(i-1)\, b{:}\, i\, b]$.
$m_i^{(\text{local})}$: the local maximum value inside $\boldsymbol{x}_i$.

BODY
for $i \leftarrow 1, \#\text{tiles}$ **do**

$$
\begin{aligned}
\boldsymbol{x}_i &\leftarrow Q[k,:]\, K^T[:, (i-1)\,b{:}\, i\,b] \\
m_i^{(\text{local})} &= \max_{j=1}^{b} (\boldsymbol{x}_i[j]) \\
m_i &\leftarrow \max\left(m_{i-1}, m_i^{(\text{local})}\right) \\
d'_i &\leftarrow d'_{i-1}\, e^{m_{i-1}-m_i} + \sum_{j=1}^{b} e^{\boldsymbol{x}_i[j]-m_i} \\
\boldsymbol{o}'_i &\leftarrow \boldsymbol{o}'_{i-1}\, \frac{d'_{i-1}\, e^{m_{i-1}-m_i}}{d'_i} + \sum_{j=1}^{b} \frac{e^{\boldsymbol{x}_i[j]-m_i}}{d'_i}\, V[j+(i-1)\,b,:]
\end{aligned}
$$

end

$$O[k,:] \leftarrow \boldsymbol{o}'_{N/b}$$

The figure 2 illustrate how to map this algorithm to hardware.

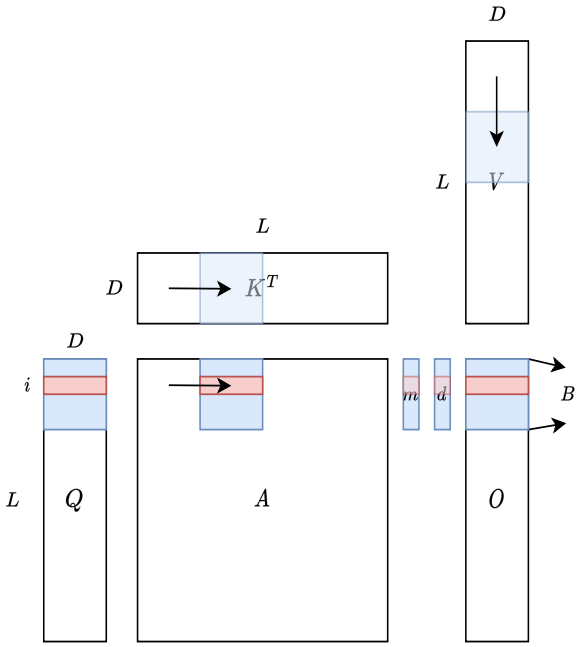

Figure 2. The diagram above illustrates how FlashAttention is computed on hardware. The blue-colored blocks represent the tiles that reside in SRAM, while the red-colored blocks correspond to the i-th row. L denotes the sequence length, which can be quite large (e.g., 16k), D denotes the head dimension, which is usually small in Transformers (e.g., 128 for GPT3), and B is the block size that can be controlled.

Notably, the overall SRAM memory footprint depends only on B and D and is not related to L. As a result, this algorithm can scale to long context without encountering memory issues (GPU shared memory is small, 228kb/SM for H100 architecture). During the computation, we sweep the tiles from left to right for K^T and A, from top to bottom for V, and update the state of m, d, and O accordingly.

Bibliography

[1] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: fast and memory-efficient exact attention with io-awareness. *CoRR*, abs/2205.14135, 2022.

[2] Andrew Kerr. Gtc 2020: developing cuda kernels to push tensor cores to the absolute limit on nvidia a100. May 2020.

[3] Maxim Milakov and Natalia Gimelshein. Online normalizer calculation for softmax. *CoRR*, abs/1805.02867, 2018.