Flash Attention

Balakrishnan Nagaraj

November 2025

1 Introduction

The Transformer architecture changed the way sequence data is modeled by replacing recurrence with self-attention, where every token can directly interact with every other token. This design captures long-range relationships effectively but comes with a high computational and memory cost that grows rapidly with sequence length. Earlier methods such as Sparse Attention, Linformer, Performer, and Longformer tried to reduce this cost by using *approximations* like sparsity or low-rank projections. While these approaches improved efficiency, they often sacrificed exactness or generality.

FlashAttention addresses this challenge without compromising accuracy. It is an IO-aware algorithm that accelerates the computation of self-attention by focusing on efficient data movement rather than mathematical shortcuts. By reorganizing how attention is computed and how data flows through GPU memory, FlashAttention achieves the same results as standard attention but with significantly improved speed and scalability on long sequences.

2 Background

2.1 The Self-Attention Mechanism

In the self-attention mechanism, the output for each token is computed as

$$O = \mathrm{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V,$$

where Q, K, and V are the query, key, and value matrices, and d is the head dimension. This operation allows each token to attend to all others in the sequence, capturing contextual dependencies effectively.

2.2 Computational and Memory Costs

The computational and memory requirements of self-attention increase rapidly with sequence length N. The matrix multiplication QK^T involves $O(N^2d)$ operations, and the resulting attention matrix of size $N \times N$ must be stored temporarily during computation, requiring $O(N^2)$ space. Although this quadratic scaling is inherent to the self-attention mechanism, in practice the major bottleneck arises not from the arithmetic itself but from the repeated reading and writing of large intermediate matrices between different levels of memory.

2.3 Modern GPU Architecture

GPUs employ a hierarchical memory structure with components that vary widely in size and speed. At the top level, high-bandwidth memory (HBM) provides tens of gigabytes of storage but is relatively slow to access. Closer to the processing cores are much smaller but significantly faster on-chip memories such as shared memory (SRAM) and registers. For example, an NVIDIA A100 GPU offers 40–80 GB of HBM with around 1.5–2 TB/s bandwidth, compared to only about 192 KB of on-chip SRAM per streaming multiprocessor with nearly an order of magnitude higher bandwidth. As GPU compute speed has grown faster than memory bandwidth, performance for many workloads is now limited by how quickly data can be moved between these memory levels rather than by arithmetic capability itself.

GPU computations are executed through *kernels*, each representing a parallel operation launched across thousands of lightweight threads. A kernel typically loads data from HBM into fast on-chip memory, performs the required arithmetic, and writes the results back to HBM. The performance of a kernel depends on its *arithmetic intensity*—the ratio of computation to data movement. If computation dominates, the operation is said to be *compute-bound*; examples include large matrix multiplications or deep convolutions with many channels. Conversely, when data transfer dominates, the operation is *memory-bound*; this is the case for many elementwise functions, reductions (such as softmax or normalization), and attention mechanisms.

2.4 The Need for IO-Aware Computation

In practice, many operations within the self-attention computation are bottlenecked by memory access rather than arithmetic cost, as data must be frequently read from and written to different levels of GPU memory. FlashAttention addresses this inefficiency by reorganizing the computation to minimize memory traffic while preserving the exact mathematical result of standard self-attention. The algorithm introduces the principle of *IO-aware* computation, which explicitly accounts for the cost of reading and writing data between different levels of GPU memory. On modern hardware, computational throughput has grown much faster than memory bandwidth, making memory access the dominant factor in the runtime of attention operations. FlashAttention optimizes this by reorganizing the attention computation into small blocks that fit in fast on-chip memory, avoiding the need to store or repeatedly access the large intermediate attention matrix. It performs the softmax operation incrementally over these blocks (a technique known as tiling) and recomputes the necessary quantities during the backward pass instead of reading them from memory. Despite performing a comparable number of arithmetic operations, this IO-efficient design drastically reduces memory traffic and leads to significant speedups over standard attention while maintaining exact results.

3 Flash Attention Algorithm

3.1 Forward Pass

There are three ingredients that go into constructing the forward pass of flash attention:

1. Numerically stable softmax
2. Online softmax
3. Block computation of matrix multiplication

3.1.1 Numerically Stable Softmax

The softmax operation converts raw attention scores into normalized probabilities, ensuring that the weights assigned to all key positions for each query sum to one. However, directly computing

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

can lead to numerical overflow or underflow when the values of x_i are large in magnitude. To improve stability, the softmax is implemented in its numerically stable form by subtracting the maximum value within each row:

$$\text{softmax}(x_i) = \frac{e^{x_i - \max_j(x_j)}}{\sum_k e^{x_k - \max_j(x_j)}}.$$

Notice that the new softmax(x_i) still computes the same value as $e^{-\max_j(x_j)}$ in the numerator and the denominator cancel each other out. But now $x_i - \max_j(x_j)$ is always less than or equal to zero and hence $e^{-\max_j(x_j)}$ do not explode. This formulation ensures that all exponential arguments are non-positive, preventing overflow while preserving the exact mathematical result.

3.1.2 Online Softmax

In the standard implementation, the softmax for each query vector requires access to all its attention scores before normalization. This means that the entire set of scores must be computed and stored in memory, which is not feasible when working with long sequences. To avoid this, FlashAttention computes the softmax in an *online* manner, processing one segment of the scores at a time while maintaining running statistics that allow exact normalization.

For a single query, let the attention scores be represented as a sequence of values $s_1, s_2, \dots, s_N$. Instead of computing the maximum and sum over all scores at once, the algorithm updates them incrementally. After processing the first i scores, it keeps track of the running maximum m_i and normalization factor l_i:

$$m_i = \max(m_{i-1}, s_i), \qquad l_i = e^{m_{i-1}-m_i} l_{i-1} + e^{s_i - m_i}.$$

The first term, $e^{m_{i-1}-m_i} l_{i-1}$, rescales the previously accumulated normalization factor l_{i-1} to account for any change in the running maximum from m_{i-1} to m_i, ensuring that the exponents remain numerically stable even when new scores exceed the previous maximum. The second term, $e^{s_i - m_i}$, adds the contribution from the newly encountered score. Together, these updates allow the algorithm to maintain the exact normalization constant as if all scores had been processed simultaneously, but using only constant memory.

Once all N scores have been processed, the final maximum m_N and normalization factor l_N are used to compute the softmax output for each element:

$$p_i = \frac{e^{s_i - m_N}}{l_N}.$$

This yields the exact same result as the standard softmax, while requiring only a single pass through the data and constant memory overhead.

3.1.3 Block Computation

Large matrix multiplications are often too big to fit entirely in fast on-chip memory, requiring data to be repeatedly loaded from and written back to slower main memory. To address this, modern high-performance algorithms use a technique known as *tiling* or *block computation*. The main idea is to divide the matrices into smaller submatrices (or tiles) that fit within fast memory, allowing computation to proceed on one tile at a time while reusing loaded data efficiently.

Consider the matrix multiplication $C = AB$, where $A \in \mathbb{R}^{M \times K}$, $B \in \mathbb{R}^{K \times N}$, and $C \in \mathbb{R}^{M \times N}$. Instead of computing the entire result at once, we partition the matrices into blocks of size $B_M \times B_K$ and $B_K \times B_N$, such that

$$A = \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1p} \\ A_{21} & A_{22} & \cdots & A_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m1} & A_{m2} & \cdots & A_{mp} \end{bmatrix}, \qquad B = \begin{bmatrix} B_{11} & B_{12} & \cdots & B_{1n} \\ B_{21} & B_{22} & \cdots & B_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ B_{p1} & B_{p2} & \cdots & B_{pn} \end{bmatrix}.$$

Each block A_{ij} or B_{ij} represents a small submatrix, for example:

$$A_{11} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1B_K} \\ a_{21} & a_{22} & \cdots & a_{2B_K} \\ \vdots & \vdots & \ddots & \vdots \\ a_{B_M1} & a_{B_M2} & \cdots & a_{B_MB_K} \end{bmatrix}$$

Each output block C_{ij} is computed as a sum of products between corresponding tiles:

$$C_{ij} = \sum_{k=1}^{p} A_{ik} B_{kj}$$

This means that one pair of blocks A_{ik} and B_{kj} is loaded into fast memory, multiplied to produce a partial result, and then accumulated into C_{ij}. By performing the computation block by block, the algorithm minimizes memory movement while maximizing data reuse. This blockwise strategy forms the basis of most efficient matrix multiplication kernels, as it balances the limited capacity of fast memory with the need to minimize data transfer.

3.1.4 Putting It All Together

We now combine the ideas of numerically stable softmax, online softmax, and block computation to describe the complete FlashAttention forward pass.

High-Level Idea The FlashAttention forward pass can be summarized as follows:

- **Tiling:** Split the query (Q), key (K), and value (V) matrices into smaller tiles that fit in fast on-chip memory. Each query tile $Q_i \in \mathbb{R}^{B_Q \times d}$ is processed sequentially against all key/value tiles $K_j, V_j \in \mathbb{R}^{B_K \times d}$.
- **Streaming computation:** Keep the query tile Q_i resident in fast memory while streaming the key/value tiles one at a time from slower global memory.
- **Online updates:** For each pair of tiles (Q_i, K_j, V_j), compute partial attention scores and update running per-row quantities:
  - m_{ij}: running maximum (for numerical stability)
  - l_{ij}: normalization factor (for softmax scaling)
  - O_i: accumulated output
- **Numerical stability:** Apply rescaling using the factor $\exp(m_{ij-1} - m_{ij})$ to keep all quantities consistent under changing maxima.
- **Final normalization:** After iterating through all key/value tiles, normalize O_i by the final l_{iN} to obtain the exact softmax output for that query tile.

Algorithm For each pair of tiles (Q_i, K_j, V_j):

$$S_{ij} = Q_i K_j^T,$$

$$m_{ij} = \max(\text{rowmax}(S_{ij}), m_{ij-1}),$$

$$P_{ij} = \exp(S_{ij} - m_{ij}),$$

$$l_{ij} = \text{rowsum}(P_{ij}) + l_{ij-1} \exp(m_{ij-1} - m_{ij}),$$

$$O_i = \text{diag}(\exp(m_{ij-1} - m_{ij}))O_i + P_{ij} V_j.$$

After all key/value tiles are processed, the final normalization is applied:

$$O_i = (\text{diag}(l_{iN}))^{-1} O_i,$$

where N is the total number of key/value tiles.

Here, m_{ij} represents the updated running maximum for each query row, and l_{ij} is the corresponding normalization factor accumulated so far. The matrix exponential and maximum operations are applied elementwise across rows of the tile. The rescaling factor $\exp(m_{ij-1} - m_{ij})$ ensures that the previously accumulated quantities are adjusted to the new numerical maximum, maintaining numerical stability across iterations.

Each query tile Q_i remains in on-chip memory throughout computation, while key/value tiles (K_j, V_j) are streamed from global memory. At every step, only small per-row quantities (m_i, l_i, and O_i) are updated, avoiding the need to materialize the large $N \times N$ attention matrix. This fusion of block computation and online softmax enables FlashAttention to compute exact attention efficiently while drastically reducing memory traffic.

3.1.5 Pseudocode

```
# FlashAttention - Forward pass (tile-wise pseudocode)

Inputs:
  Q: (N, d)           # full query matrix (per head)
  K: (N, d)           # full key matrix (per head)
  V: (N, d)           # full value matrix (per head)
  B_Q, B_K            # query-tile and key/value-tile sizes
  scale = 1 / sqrt(d)

For each query_tile_index i = 0 .. ceil(N / B_Q)-1:
  Q_i = load_tile(Q, i, B_Q)          # shape (B_Q, d)
  initialize:
    m    = -inf * ones(B_Q)           # running max per query row
    l    = zeros(B_Q)                 # running normalizer per query row
    O_acc = zeros(B_Q, d)             # output accumulator for tile

  For each kv_tile_index j = 0 .. ceil(N / B_K)-1:
    K_j = load_tile(K, j, B_K)        # shape (B_K, d)
    V_j = load_tile(V, j, B_K)        # shape (B_K, d)

    # Compute partial logits for this tile
    S = (Q_i @ K_j^T) * scale         # shape (B_Q, B_K)

    # (Optional) apply causal mask to S here if autoregressive

    # Row-wise block maximum
    s_max = row_max(S)                # shape (B_Q)

    # Update running maximum (elementwise per row)
    m_new = max(m, s_max)             # shape (B_Q)

    # Compute exponentials in numerically stable frame
    P = exp(S - m_new[:, None])       # shape (B_Q, B_K)

    # Update running normalizer l with rescaling
    l = exp(m - m_new) * l + row_sum(P)   # shape (B_Q)

    # Rescale previous accumulator and add contribution from this block
    O_acc = exp(m - m_new)[:, None] * O_acc + P @ V_j   # shape (B_Q, d)

    # Commit updated running max
    m = m_new

  # End kv-tiles loop

  # Final normalization for this query tile
  O_i = O_acc / l[:, None]            # shape (B_Q, d)

  # Store output tile into O
  store_tile(O, i, O_i)

# End query-tiles loop
```

```
Output:
  O: (N, d)  # attention outputs per head (reconstructed from tiles)
```

3.2 Backward Pass

We now describe the backward pass of FlashAttention, which computes gradients with respect to the query, key, and value matrices without storing the full $N \times N$ attention matrix. Like the forward pass, it reuses the principles of tiling, online recomputation, and numerical stability.

3.2.1 Gradient Calculations

Here I will provide a detailed derivation of the gradients. The derivation given in the flash attention paper is a little confusing, and hence I will provide my own derivation.

We have the following equations:

$$S = \frac{QK^T}{\sqrt{d}}, \qquad P = \text{softmax}(S), \qquad O = PV.$$

We can write these equations explicitly with all the indices as follows:

$$S_{ij} = \sum_k \frac{Q_{ik}K_{jk}}{\sqrt{d}}, \qquad P_{ij} = \text{softmax}(S_{ij}) = \frac{\exp\left(\sum_k Q_{ik}K_{jk}/\sqrt{d}\right)}{\sum_l \exp\left(\sum_m Q_{im}K_{lm}/\sqrt{d}\right)}, \qquad O_{ij} = \sum_k P_{ik}V_{kj}.$$

Our problem is to find the gradients

$$dQ_{ij} = \frac{\partial \mathcal{L}}{\partial Q_{ij}}, \qquad dK_{ij} = \frac{\partial \mathcal{L}}{\partial K_{ij}}, \qquad dV_{ij} = \frac{\partial \mathcal{L}}{\partial V_{ij}},$$

given the gradient with respect to the output $dO_{ij} = \partial\mathcal{L}/\partial O_{ij}$.

Now, let us get the gradients one by one. We will start with the easiest.

$$\begin{aligned}
dV_{ij} &= \frac{\partial \mathcal{L}}{\partial V_{ij}} \\
&= \sum_{mn} \frac{\partial \mathcal{L}}{\partial O_{mn}} \frac{\partial O_{mn}}{\partial V_{ij}} \\
&= \sum_{mn} \frac{\partial \mathcal{L}}{\partial O_{mn}} \frac{\partial(\sum_l P_{ml}V_{ln})}{\partial V_{ij}} \\
&= \sum_{mnl} \frac{\partial \mathcal{L}}{\partial O_{mn}} P_{ml}\delta_{il}\delta_{nj} \\
&= \sum_{m} \frac{\partial \mathcal{L}}{\partial O_{mj}} P_{mi} \\
&= \sum_{m} (P^T)_{im} \frac{\partial \mathcal{L}}{\partial O_{mj}}.
\end{aligned}$$

This can be written as

$$dV_{ij} = \sum_m P^T_{im} dO_{mj}.$$

Now let us now on to dQ and dK. In the above derivation, we used Kronecker delta function, which is defined as

$$\delta_{ij} = \begin{cases} 1, & \text{if } i = j, \\ 0, & \text{if } i \neq j. \end{cases}$$

Using chain rule, we have

$$\begin{aligned} dQ_{ij} &= \sum_{mnklrt} \frac{\partial \mathcal{L}}{\partial O_{mn}} \frac{\partial O_{mn}}{\partial P_{kl}} \frac{\partial P_{kl}}{\partial S_{rt}} \frac{\partial S_{rt}}{\partial Q_{ij}} \\ &= \sum_{klrt} \frac{\partial \mathcal{L}}{\partial P_{kl}} \frac{\partial P_{kl}}{\partial S_{rt}} \frac{\partial S_{rt}}{\partial Q_{ij}} \end{aligned}$$

Similarly

$$\begin{aligned} dK_{ij} &= \sum_{mnklrt} \frac{\partial \mathcal{L}}{\partial O_{mn}} \frac{\partial O_{mn}}{\partial P_{kl}} \frac{\partial P_{kl}}{\partial S_{rt}} \frac{\partial S_{rt}}{\partial K_{ij}} \\ &= \sum_{klrt} \frac{\partial \mathcal{L}}{\partial P_{kl}} \frac{\partial P_{kl}}{\partial S_{rt}} \frac{\partial S_{rt}}{\partial K_{ij}} \end{aligned}$$

So for both dQ and dK we need to find $\partial P_{kl}/\partial S_{rt}$. We have

$$P_{kl} = \frac{e^{S_{kl}}}{\sum_m e^{S_{km}}}$$

Therefore

$$\begin{aligned} \frac{\partial P_{kl}}{\partial S_{rt}} &= \frac{\partial}{\partial S_{rt}} \left(\frac{e^{S_{kl}}}{\sum_m e^{S_{km}}} \right) \\ &= \frac{e^{S_{kl}} \delta_{kr} \delta_{lt} (\sum_m e^{S_{km}}) - e^{S_{kl}} (\sum_m e^{S_{km}} \delta_{kr} \delta_{mt})}{(\sum_m e^{S_{km}})^2} \\ &= \frac{e^{S_{kl}} \delta_{kr} \delta_{lt} (\sum_m e^{S_{km}}) - e^{S_{kl}} e^{S_{kt}} \delta_{kr}}{(\sum_m e^{S_{km}})^2} \\ &= \frac{e^{S_{kl}}}{\sum_m e^{S_{km}}} \delta_{kr} \left(\delta_{lt} - \frac{e^{S_{kt}}}{\sum_m e^{S_{km}}} \right) \\ &= P_{kl} \delta_{kr} (\delta_{lt} - P_{kt}). \end{aligned}$$

Using the above result,

$$\begin{aligned} \sum_{kl} \frac{\partial \mathcal{L}}{\partial P_{kl}} \frac{\partial P_{kl}}{\partial S_{rt}} &= \sum_{kl} \frac{\partial \mathcal{L}}{\partial P_{kl}} P_{kl} \delta_{kr} (\delta_{lt} - P_{kt}) \\ &= P_{rt} \left(\frac{\partial \mathcal{L}}{\partial P_{rt}} - \sum_l P_{rl} \frac{\partial \mathcal{L}}{\partial P_{rl}} \right) \end{aligned}$$

Now, the second term

$$\begin{aligned} \sum_l P_{rl} \frac{\partial \mathcal{L}}{\partial P_{rl}} &= \sum_{mnl} P_{rl} \frac{\partial \mathcal{L}}{\partial O_{mn}} \frac{\partial O_{mn}}{\partial P_{rl}} \\ &= \sum_{mnlt} P_{rl} \frac{\partial \mathcal{L}}{\partial O_{mn}} \delta_{mr} \delta_{tl} V_{tn} \\ &= \sum_{nl} P_{rl} \frac{\partial \mathcal{L}}{\partial O_{rn}} V_{ln} \\ &= \sum_n O_{rn} \frac{\partial \mathcal{L}}{\partial O_{rn}} \end{aligned}$$

The above result allows us to trade the summation over the sequence length (l) for the summation over the head dimension (d). Therefore,

$$\sum_{kl} \frac{\partial \mathcal{L}}{\partial P_{kl}} \frac{\partial P_{kl}}{\partial S_{rt}} = P_{rt} \frac{\partial \mathcal{L}}{\partial P_{rt}} - P_{rt} \sum_n O_{rn} \frac{\partial \mathcal{L}}{\partial O_{rn}}.$$

Notice that

$$dS_{rt} = \frac{\partial \mathcal{L}}{\partial S_{rt}} = \sum_{kl} \frac{\partial \mathcal{L}}{\partial P_{kl}} \frac{\partial P_{kl}}{\partial S_{rt}}$$

So we have,

$$dS_{rt} = P_{rt} \left(dP_{rt} - \sum_n O_{rn} dO_{rn} \right).$$

We will define $D_r = \sum_n O_{rn} dO_{rn}$. Therefore, we finally have

$$dS_{rt} = P_{rt}(dP_{rt} - D_r).$$

We still have to find dP_{rt} to make use of the above formula in computations. We have

$$\begin{aligned} dP_{rt} = \frac{\partial \mathcal{L}}{\partial P_{rt}} &= \sum_{mn} \frac{\partial \mathcal{L}}{\partial O_{mn}} \frac{\partial O_{mn}}{\partial P_{rt}} \\ &= \sum_{mnl} dO_{mn} \delta_{mr} \delta_{lt} V_{ln} \\ &= \sum_{n} dO_{rn} V_{tn} \\ &= \sum_{n} dO_{rn} V^T_{nt} \end{aligned}$$

Now we can use dS_{rt} to find the gradients with respect to Q and K,

$$\begin{aligned} dQ_{ij} &= \sum_{rt} \frac{\partial \mathcal{L}}{\partial S_{rt}} \frac{\partial S_{rt}}{\partial Q_{ij}} \\ &= \frac{1}{\sqrt{d}} \sum_{rtm} dS_{rt} \delta_{ri} \delta_{mj} K_{tm} \\ &= \frac{1}{\sqrt{d}} \sum_{t} dS_{it} K_{tj}. \end{aligned}$$

Similarly,

$$\begin{aligned} dK_{ij} &= \sum_{rt} \frac{\partial \mathcal{L}}{\partial S_{rt}} \frac{\partial S_{rt}}{\partial K_{ij}} \\ &= \frac{1}{\sqrt{d}} \sum_{rtm} dS_{rt} Q_{rm} \delta_{ti} \delta_{mj} \\ &= \frac{1}{\sqrt{d}} \sum_{r} dS_{ri} Q_{rj} \\ &= \frac{1}{\sqrt{d}} \sum_{r} dS^T_{ir} Q_{rj}. \end{aligned}$$

The final set of equations are

$$\begin{aligned}
dV_{ij} &= \sum_k P^T_{ik} dO_{kj}, \\
dP_{ij} &= \sum_k dO_{ik} V^T_{kj} \\
D_i &= \sum_j O_{ij} dO_{ij}, \\
dS_{ij} &= P_{ij}(dP_{ij} - D_i), \\
dQ_{ij} &= \frac{1}{\sqrt{d}} \sum_k dS_{ik} K_{kj}, \\
dK_{ij} &= \frac{1}{\sqrt{d}} \sum_k dS^T_{ik} Q_{kj}.
\end{aligned}$$

We can use all the above formulae to get the gradients.

3.2.2 The Log-Sum-Exp Trick and Forward-Pass Statistics

A standard implementation of attention would store the full probability matrix $P \in \mathbb{R}^{N \times N}$ during the forward pass so that it can be reused in the backward pass. However, this is infeasible for long sequences: the matrix P requires $O(N^2)$ memory, which quickly exceeds GPU capacity even for moderate values of N. FlashAttention avoids this cost entirely by *never storing* P. Instead, the backward pass simply recomputes the relevant blocks of P on the fly.

To make this recomputation possible, the forward pass stores only a single scalar per query row: the *log-sum-exp* value

$$M_i \;=\; m_i + \log(\ell_i) \;=\; \log\left(\sum_j e^{S_{ij}}\right),$$

where $m_i = \max_j S_{ij}$ is the running maximum accumulated across key tiles, and $\ell_i = \sum_j e^{S_{ij}-m_i}$ is the corresponding normalization factor computed in a numerically stable way. These two quantities are maintained incrementally during the tiled forward computation, and combined into M_i at the end of the forward pass.

During the backward pass, when a block of scores S_{ij} is recomputed, the corresponding block of softmax probabilities is recovered using the log-sum-exp identity:

$$P_{ij} \;=\; \exp\big(S_{ij} - M_i\big).$$

This works because

$$\exp(S_{ij} - M_i) = \frac{e^{S_{ij}-m_i}}{\sum_k e^{S_{ik}-m_i}} = \mathrm{softmax}(S_{ij}).$$

Thus, storing M_i is sufficient to compute the correct softmax of$S$ exactly, without ever computing maximum and normalization factor again.

3.2.3 Blockwise Form of the Backward Pass

To connect the index-level gradients with the FlashAttention implementation, we now express all quantities in terms of *blocks* (tiles). Let the sequence dimension be partitioned into query tiles $Q_i \in \mathbb{R}^{B_Q \times d}$ and key/value tiles $K_j, V_j \in \mathbb{R}^{B_K \times d}$. For each pair of tiles (i, j), define the block matrices

$$S_{ij} \in \mathbb{R}^{B_Q \times B_K}, \qquad P_{ij} \in \mathbb{R}^{B_Q \times B_K}, \qquad dP_{ij}, dS_{ij} \in \mathbb{R}^{B_Q \times B_K}.$$

Blockwise equations. The global elementwise formulas translate blockwise into

$$S_{ij} = \frac{Q_i K_j^T}{\sqrt{d}}, \qquad P_{ij} = \exp(S_{ij} - M_i[:, \text{None}])\,,$$

$$dP_{ij} = dO_i V_j^T, \qquad D_i = \text{rowsum}(O_i \odot dO_i),$$

$$dS_{ij} = P_{ij} \odot (dP_{ij} - D_i[:, \text{None}])\,,$$

$$dV_j \mathrel{+}= P_{ij}^T dO_i, \qquad dK_j \mathrel{+}= \frac{1}{\sqrt{d}}\, dS_{ij}^T Q_i, \qquad dQ_i \mathrel{+}= \frac{1}{\sqrt{d}}\, dS_{ij} K_j.$$

Here, M_i is the log-sum-exp normalization vector stored from the forward pass (None in $M_i[:, \text{None}]$ is to broadcast the values of M_i across the columns), and D_i ($\odot$ is elementwise multiplication) is a per-row scalar defined by

$$D_i = \sum_{\alpha} O_{i\alpha} dO_{i\alpha}.$$

The notation rowsum$(\cdot)$ sums across columns.

Blockwise backward algorithm. Using these quantities, the backward pass proceeds as follows.

- **Preprocess:** For each query tile i, load O_i and dO_i and compute

  $$D_i = \text{rowsum}(O_i \odot dO_i).$$

  This is a small per-row vector reused throughout the backward pass.

- **Gradients for K and V:** For each key/value tile j:

  1. Load K_j, V_j.
  2. For each query tile i:

  $$S_{ij} = \frac{Q_i K_j^T}{\sqrt{d}}, \qquad P_{ij} = \exp(S_{ij} - M_i[:, \text{None}])\,,$$

  $$dV_j \mathrel{+}= P_{ij}^T dO_i, \qquad dP_{ij} = dO_i V_j^T,$$

  $$dS_{ij} = P_{ij} \odot (dP_{ij} - D_i[:, \text{None}])\,, \qquad dK_j \mathrel{+}= \frac{1}{\sqrt{d}}\, dS_{ij}^T Q_i.$$

- **Gradients for Q:** For each query tile i:

  1. Initialize $dQ_i = 0$.
  2. For each key/value tile j:

  $$S_{ij} = \frac{Q_i K_j^T}{\sqrt{d}}, \qquad P_{ij} = \exp(S_{ij} - M_i[:, \text{None}])\,,$$

  $$dP_{ij} = dO_i V_j^T, \qquad dS_{ij} = P_{ij} \odot (dP_{ij} - D_i[:, \text{None}])\,,$$

  $$dQ_i \mathrel{+}= \frac{1}{\sqrt{d}}\, dS_{ij} K_j.$$

This blockwise formulation reproduces the exact gradients of standard attention while avoiding materialization of the full $N \times N$ matrices P and dP, enabling the memory-efficient backward pass used in FlashAttention.

3.2.4 Pseudocode (sketch)

```
# Preprocess: compute D[i] = sum_j O[i,j] * dO[i,j]
for each query_tile Q_t:
    load O_t, dO_t
    D_t = rowsum(O_t * dO_t)
    store D_t

# Backward for dK, dV (fix KV block, iterate Q tiles)
for each KV_block (K_b, V_b):
    for each query_tile Q_t:
        load Q_t, K_b, V_b, dO_t, M_t, D_t
        S = Q_t @ K_b^T / sqrt(d)
        P = exp(S - M_t)            # recomputed unnormalized block
        dV_b += P^T @ dO_t
        dP = dO_t @ V_b^T
        dS = P * (dP - rowsum(P * dP)[:,None])
        dK_b += (dS^T @ Q_t) / sqrt(d)
# write back dK, dV

# Backward for dQ (fix Q tile, iterate KV blocks)
for each query_tile Q_t:
    initialize dQ_t = 0
    for each KV_block (K_b, V_b):
        load Q_t, K_b, V_b, dO_t, M_t, D_t
        S = Q_t @ K_b^T / sqrt(d)
        P = exp(S - M_t)
        dP = dO_t @ V_b^T
        dS = P * (dP - rowsum(P * dP)[:,None])
        dQ_t += (dS @ K_b) / sqrt(d)
    store dQ_t
```

4 Triton

Triton is a domain-specific language for writing high-performance GPU kernels in Python. It provides a higher-level abstraction compared to CUDA by allowing developers to express computations in terms of *blocks* of data rather than individual threads. This block-based model simplifies kernel development for tensor operations such as matrix multiplication or attention, while Triton's compiler automatically optimizes memory access, register usage, and synchronization. In contrast to CUDA, where the programmer must explicitly manage threads and shared memory, Triton offers a concise and flexible syntax that achieves near-CUDA performance with significantly less complexity.

4.1 Implementation Notes

Top-level idea One Triton forward kernel (`_attn_fwd`) computes attention per query-tile while streaming K/V tiles; it never materializes the full $N \times N$ attention matrix.

Key kernels (what to look for)

- `_attn_fwd` — forward entry point (sets up pointers, grid, stages).
- `_attn_fwd_inner` — inner loop that iterates over K/V tiles, performs $Q \cdot K^T$, updates the online softmax, and accumulates `O_block`.
- `_attn_bwd_preprocess` — computes $D = \sum(O \cdot dO)$ per query row (small, cheap preprocessing).

- `_attn_bwd_dk_dv` — computes `dK` and `dV` (fixes Q blocks, iterates KV blocks).
- `_attn_bwd_dq` — computes `dQ` (fixes KV blocks, iterates Q blocks).

Tile shapes (inside the kernel)

- `Q_block`: (BLOCK_SIZE_Q, HEAD_DIM).
- `K_block` (loaded transposed): (HEAD_DIM, BLOCK_SIZE_KV).
- `V_block`: (BLOCK_SIZE_KV, HEAD_DIM).
- `QK_block` has shape (BLOCK_SIZE_Q, BLOCK_SIZE_KV); `P_block` has the same shape.

Backward strategy (no $N \times N$ stored) The forward pass stores compact per-query statistics `M` (log-sum-exp), allowing the backward kernels to recompute partial `P_block` values on the fly rather than loading a full attention matrix. This trades a small amount of extra compute for large memory savings.

Pointer / stride API The code uses `tl.make_block_ptr`, `tl.advance`, and explicit strides passed from PyTorch, so the kernels support arbitrary tensor layouts and avoid data copies.

References

[1] Tri Dao et al., *FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness*, arXiv:2205.14135, 2022. Available: https://arxiv.org/abs/2205.14135

[2] Umar Jamil (presentation), *Flash Attention derived and coded from first principles with Triton (Python)*, YouTube, 2022. Available: https://www.youtube.com/watch?v=zy8ChVd_oTM&t=14458s

[3] Dao AILab, *FlashAttention — implementation repository*, GitHub, 2022. Available: https://github.com/Dao-AILab/flash-attention

[4] Triton contributors, *Triton: an open-source language and compiler for writing optimized GPU code*, Triton documentation. Available: https://triton-lang.org