Efficient Attention

Karl Stratos

1 Overview

In decoder-only transformers, attention computes[1]

$$\underbrace{O}_{T\times d} = (\underbrace{Q}_{T\times d}\underbrace{K^\top}_{d\times T}).\mathbf{softmax}(2)\underbrace{V}_{T\times d} \tag{1}$$

How to improve the efficiency of (1) has been a central research topic in recent years.

1.1 Memory Scalability

Naively computing $QK^\top \in \mathbb{R}^{T\times T}$ requires $O(T^2)$ memory. Many works have noted that (1) can be computed incrementally [8, 5, 9]. Partition $K = (K_1 \dots K_C)$ and $V = (V_1 \dots V_C)$ into chunks of shape $\Lambda \times d$. Initialize $\pi \leftarrow 0^{T\times 1}$ and $O \leftarrow 0^{T\times d}$. For $j = 1 \dots C$ compute ($\oslash, \circledast$ are broadcasted division and multiplication, Appendix A)

$$\begin{aligned} A_j &\leftarrow \underbrace{Q}_{T\times d}\underbrace{K_j^\top}_{d\times\Lambda} & \pi^{\text{new}} &\leftarrow \pi + \underbrace{\exp(A_j)}_{T\times\Lambda}.\mathbf{sum}(2) \\ & & O^{\text{new}} &\leftarrow \underbrace{(\pi \oslash \pi^{\text{new}})}_{T\times 1} \circledast \underbrace{O}_{T\times d} + \underbrace{(\exp(A_j) \oslash \pi^{\text{new}})}_{T\times\Lambda}\underbrace{V_j}_{\Lambda\times d} \end{aligned} \tag{2}$$

and overwrite $\pi \leftarrow \pi^{\text{new}}$ and $O \leftarrow O^{\text{new}}$ at each step. It is easy to verify that the final O is equal to (1). We now need $O(T\Lambda)$ additional memory (i.e., excluding the $O(Td)$ memory for storing the input/output $Q, K, V, O \in \mathbb{R}^{T\times d}$) instead of $O(T^2)$.

1.2 Runtime Latency

How can we compute (1) *faster*? Insidiously, attention is **memory-bound**, not **compute-bound** [10, 4]: it is slow not because the operations (matmul, softmax) are slow but because it moves data around in memory. Prior to an operation, an accelerator moves data from large but slow global memory (aka. **HBM**) to small but fast on-chip memory (aka. **SRAM**) shared by execution units (e.g., threads on GPUs). Accessing HBM takes time, thus a "good" accelerator program (aka. kernel) minimizes HBM access, e.g., by using a small intermediate cache on SRAM. For example, consider the following algorithms for square matrix multiplication $C = AB$ (A, B are already in HBM):

MatMul I

- $C \leftarrow 0^{n\times n}$
- For $i = 1 \dots n$:
  - For $j = 1 \dots n$:
    - $c_{i,j} \leftarrow 0$
    - For $k = 1 \dots n$:
      - $a_{i,k} \leftarrow \mathbf{READ}(A_{i,k})$
      - $b_{k,j} \leftarrow \mathbf{READ}(B_{k,j})$
      - $C_{i,j} \leftarrow C_{i,j} + a_{i,k} \times b_{k,j}$

MatMul II (cache $u, v \in \mathbb{R}^n$ on SRAM)

- $C \leftarrow 0^{n\times n}$
- For $k = 1 \dots n$:
  - For $i = 1 \dots n$:
    - $u_i \leftarrow \mathbf{READ}(A_{i,k})$
    - $v_i \leftarrow \mathbf{READ}(B_{k,i})$
  - For $i = 1 \dots n$:
    - For $j = 1 \dots n$:
      - $C_{i,j} \leftarrow C_{i,j} + u_i \times v_j$

While they both perform $2n^3$ FLOPs (floating-point operations), **MatMul I** performs $2n^3$ global reads, whereas **MatMul II** performs $2n^2$ global reads by using the shared on-chip memory $u, v \in \mathbb{R}^n$ to avoid re-reading.[2] In a similar manner, **FlashAttention** (Section 2) is an attention kernel that minimizes HBM access.

[1] For simplicity, we omit the causal masking which rewrites $A = QK^\top$ to contain $A_{t,t'} = -\infty$ for $t' > t$.

[2] It also uses the observation that $AB = \sum_{k=1}^n A.\mathbf{col}(k)A.\mathbf{row}(k)$.

1.3 Inference Latency

At inference time, (1) becomes a matrix-*vector* product:[3]

$$\underbrace{o}_{1\times d} = (\underbrace{q}_{1\times d}\ \underbrace{K^\top}_{d\times(T+1)}).\mathbf{softmax}(2)\ \underbrace{V}_{(T+1)\times d} \tag{3}$$

where $K, V \in \mathbb{R}^{(T+1)\times d}$ include T previous key-value embeddings. To avoid recomputation, these embeddings are loaded from a cache called the **KV cache**. Loading from the KV cache becomes the latency bottleneck as T grows. Various works propose modifications of the transformer architecture to make the KV cache smaller (Section 3), e.g., by reducing the number of key-value heads (GQA [10, 1]) or down-projecting the hidden states (MLA [7]).

1.4 Inference Throughput

Throughput is the key efficiency metric when we process multiple inference requests. Throughput is again memory-bound; it is low because we lack the memory to process more requests concurrently. A main issue is *memory fragmentation*. Consider two simultaneous requests A, B with maximum target lengths $T_B \gg T_A$. A standard memory management scheme allocates a continguous sequence of bytes (Appendix B) for the batch tensor. This causes two types of memory waste:

- **External fragmentation**: For request A, $O(T_B - T_A)$ bytes are never used.
- **Internal fragmentation**: Assuming the generations for A, B end early (i.e., the model generates the end-of-sequence tokens) with lengths T'_A, T'_B, $O((T_A - T'_A) + (T_B - T'_B))$ bytes are never used.

Even the reserved bytes for future tokens that *do* get used stay unused most of the time; it is more efficient to use them for processing other requests until needed. **PagedAttention** [6], implemented in the popular vLLM library, solves this problem by applying the concept of paged virtual memory. It non-contiguously stores the KV cache in *small, fixed-size* blocks, then uses a centralized scheduler to coordinate GPUs on how to use them. It reduces memory fragmentation and also enables features like cache sharing and request prioritization, dramatically increasing the throughput when serving a large number of streaming requests.

2 FlashAttention

2.1 Review of Standard Attention

The forward function of standard attention takes $Q, K, V \in \mathbb{R}^{T\times d}$ as input and outputs $O \in \mathbb{R}^{T\times d}$ by

$$O = \underbrace{\exp\left(\frac{QK^\top}{\sqrt{d}} \ominus \left(\frac{QK^\top}{\sqrt{d}}\right).\mathbf{max}(2)\right)}_{T\times T} \oslash \underbrace{\exp\left(\frac{QK^\top}{\sqrt{d}} \ominus \left(\frac{QK^\top}{\sqrt{d}}\right).\mathbf{max}(2)\right).\mathbf{sum}(2)}_{T\times 1} \underbrace{V}_{T\times d} \tag{4}$$

This is a real-world version of (1) which uses numerically stable softmax and dot product scaling. The backward function computes $z_Q, z_K, z_V \in \mathbb{R}^{T\times d}$ given $z_O \in \mathbb{R}^{T\times d}$, where $z_X = \frac{\partial L}{\partial X} \in \mathbb{R}^{m\times n}$ denotes the gradient of a loss $L \in \mathbb{R}$ with respect to tensor $X \in \mathbb{R}^{m\times n}$. Let $A = \frac{QK^\top}{\sqrt{d}}$ and $P = A.\mathbf{softmax}(2)$ denote the intermediate variables in (4) whereby $O = PV$ (we assume that A, P are not used outside this operation). From the backward functions of matmul and softmax (Appendix C.1), we have $z_P = z_O V^\top \in \mathbb{R}^{T\times T}$ and $z_A = P \circledast z_P - (P \circledast z_P).\mathbf{sum}(2) \circledast P \in \mathbb{R}^{T\times T}$. By the chain rule,

$$z_V \leftarrow z_V + P^\top z_O \qquad z_K \leftarrow z_K + \frac{z_A^\top Q}{\sqrt{d}} \qquad z_Q \leftarrow z_Q + \frac{z_A K}{\sqrt{d}} \tag{5}$$

Clearly, the memory overhead of both (4) and (5) is $O(T^2)$ since we explicitly compute $A, P \in \mathbb{R}^{T\times T}$. It makes $O(T^2 + Td)$ HBM reads (for loading and writing P as well as Q, K, V).

[3] This is preceded by a so-called "prefill" stage that computes (1) for all conditioning tokens to obtain the initial KV cache and predict the first token, which becomes the decoding input in the first step.

2.2 Forward

FlashAttention [3] first notes incremental attention (2) has implications not only on memory usage but also HBM access. Let Λ denote a chunk size, grouping $K, V \in \mathbb{R}^{T\times d}$ into $C = \frac{T}{\Lambda}$ chunks $(K_1 \dots K_C), (V_1 \dots V_C)$. Let Γ denote a batch size, grouping $Q \in \mathbb{R}^{T\times d}$ into $B = \frac{T}{\Gamma}$ batches $(Q_1 \dots Q_B)$. The following is a minor variant of (2) that uses stable softmax and treats each query batch Q_i independently (assuming a batch-wise causal mask); this style of computation is called "tiling" because we process a $\Gamma \times \Lambda$ tile of the attention matrix at each step. We assume that the inputs $Q, K, V \in \mathbb{R}^{T\times d}$ are in HBM. We assume that $c \leftarrow -\infty^{T\times 1}$, $\pi \leftarrow 0^{T\times 1}$, $O \leftarrow 0^{T\times d}$ are initialized and in HBM.

FlashForward

1. **Load** $c = (c_1 \dots c_B), \pi = (\pi_1 \dots \pi_B) \in \mathbb{R}^{T\times 1}$ **to SRAM.**
2. For $j = 1 \dots C$:
   (a) **Load** $K_j, V_j \in \mathbb{R}^{\Lambda\times d}$ **to SRAM.**
   (b) For $i = 1 \dots B$: (embarrassingly parallel)
      i. **Load** $Q_i, O_i \in \mathbb{R}^{\Gamma\times d}$ **to SRAM.**
      ii. Compute

$$A_{i,j} \leftarrow \frac{Q_i K_j^\top}{\sqrt{d}} \in \mathbb{R}^{\Gamma\times\Lambda} \qquad c_i^{\text{new}} \leftarrow \max\left(c_i, A_{i,j}.\mathbf{max}(2)\right) \in \mathbb{R}^{\Gamma\times 1}$$

$$\pi_i^{\text{new}} \leftarrow \exp(c_i - c_i^{\text{new}}) \circledast \pi_i + \exp(A_{i,j} \ominus c_i^{\text{new}}).\mathbf{sum}(2) \in \mathbb{R}^{\Gamma\times 1}$$

$$O_i^{\text{new}} \leftarrow (\pi_i \oslash \pi_i^{\text{new}}) \circledast \exp(c_i - c_i^{\text{new}}) \circledast O_i + (\exp(A_{i,j} \ominus c_i^{\text{new}}) \oslash \pi_i^{\text{new}}) V_j \in \mathbb{R}^{\Gamma\times d}$$

      iii. Update $c_i \leftarrow c_i^{\text{new}}$, $\pi_i \leftarrow \pi_i^{\text{new}}$, $O_i \leftarrow O_i^{\text{new}}$.
      iv. **Write** $O_i \in \mathbb{R}^{\Gamma\times d}$ **to HBM**.
3. Return $O \in \mathbb{R}^{T\times d}$ in HBM.

It has $O(\max(T, \Gamma\Lambda))$ additional memory overhead (to store $c, \pi \in \mathbb{R}^T$ and $A_{i,j} \in \mathbb{R}^{\Gamma\times\Lambda}$). It makes $O(CTd) = O(\frac{T^2 d}{\Lambda})$ HBM accesses since it accesses Q, O for C times in the inner loop. For best latency, we want to use the largest possible Λ, Γ under the on-chip memory constraint.[4] Let M denote the number of bytes in SRAM. The constraints are:

$$\Lambda d = O(M) \qquad \text{(to fit } K_j, V_j \in \mathbb{R}^{\Lambda\times d}\text{)}$$
$$\Gamma d = O(M) \qquad \text{(to fit } Q_i, O_i \in \mathbb{R}^{\Gamma\times d}\text{)}$$
$$\Lambda\Gamma = O(M) \qquad \text{(to fit } A_{i,j} \in \mathbb{R}^{\Gamma\times\Lambda}\text{)}$$

Choosing $\Lambda = \Theta(\frac{M}{d})$ and $\Gamma = \Theta(\min(\frac{M}{d}, \frac{M}{\Lambda})) = \Theta(\min(\frac{M}{d}, d))$ satisfies these constraints.[5] This implies $O(\frac{T^2 d^2}{M})$ HBM accesses, which is fewer than the naive $O(T^2)$ HBM accesses assuming $M > d^2$. The assumption is benign given that d corresponds to the head dimension (e.g., $d = 64 = \frac{8{,}192}{128}$ for 70B Llama 3) and modern chips have many kilobytes of on-chip memory (e.g., A100s have 192KB or $M = 196{,}608$ bytes).

2.3 Backward

Recall the formulas in the standard backward function (5) where we express $P = A.\mathbf{softmax}(2)$ using $c, \pi \in \mathbb{R}^{T\times 1}$ from the forward function:

$$A = \frac{QK^\top}{\sqrt{d}} \in \mathbb{R}^{T\times T} \qquad P = \exp(A \ominus c) \oslash \pi \in \mathbb{R}^{T\times T} \qquad O = PV \in \mathbb{R}^{T\times d}$$

$$z_A = P \circledast (z_P \ominus (P \circledast z_P).\mathbf{sum}(2)) \in \mathbb{R}^{T\times T} \qquad z_P = z_O V^\top \in \mathbb{R}^{T\times T} \qquad z_O \in \mathbb{R}^{T\times d} \text{ (input)}$$

$$z_Q \leftarrow z_Q + \frac{z_A K}{\sqrt{d}} \in \mathbb{R}^{T\times d} \qquad z_K \leftarrow z_K + \frac{z_A^\top Q}{\sqrt{d}} \in \mathbb{R}^{T\times d} \qquad z_V \leftarrow z_V + P^\top z_O \in \mathbb{R}^{T\times d}$$

[4] But note that there is a tradeoff between memory usage and HBM access. If we set $\Lambda = T$ (i.e., 1 chunk), hypothetically assuming we have that much SRAM, we make only $O(Td)$ accesses but the memory overhead is back to standard attention (i.e., $O(T^2)$ for batch size $\Gamma = T$). If we set $\Lambda = 1$ (i.e., T chunks), memory usage is truly $O(T)$ but we make $O(T^2 d)$ HBM accesses.

[5] In the paper, the authors specify "$\frac{M}{4d}$" to account for the fact that each 32-bit float takes up 4 bytes of memory. However, Λ, Γ are just fixed to some reasonable size (e.g., 512) in practice.

Since we never explicitly store the whole $P \in \mathbb{R}^{T\times T}$ during forward, we need to recompute it in tiles (gradient checkpointing). This is easy since $P_{i,j} = \exp(\frac{Q_i K_j^\top}{\sqrt{d}} \ominus c_i) \oslash \pi_i \in \mathbb{R}^{\Gamma\times\Lambda}$ where $c_i, \pi_i \in \mathbb{R}^{\Gamma\times 1}$ denote the i-th segments of c, π. The gradients must be computed in tiles as well. Let $z_O = (z_{O,1} \dots z_{O,B})$ where $z_{O,i} \in \mathbb{R}^{\Gamma\times d}$. The (i,j)-th tile of z_P is $z_{P,i,j} = z_{O,i} V_j^\top \in \mathbb{R}^{\Gamma\times\Lambda}$. The case for z_A is trickier. If we have the vector $u := (P \circledast z_P).\mathbf{sum}(2) \in \mathbb{R}^{T\times 1}$, then (i,j)-th tile of z_A is $z_{A,i,j} = P_{i,j} \circledast (z_{P,i,j} \ominus u_i) \in \mathbb{R}^{\Gamma\times\Lambda}$ where $u_i \in \mathbb{R}^{\Gamma\times 1}$ denotes the i-th segment of u. However, computing u requires summing over all T elements of P, which we never explicitly materialize. To avoid this difficulty, FlashAttention uses the following reparameterization:

Lemma 2.1.

$$u := \underbrace{(P \circledast z_P)}_{T\times T}.\mathbf{sum}(2) = \underbrace{(O \circledast z_O)}_{T\times d}.\mathbf{sum}(2) \in \mathbb{R}^{T\times 1} \tag{6}$$

The intuition is that $O = PV$ has "already done" the sum over T elements, and we should be able to use this fact given $Pz_P^\top = PVz_O^\top = Oz_O^\top$. The paper proves (6) by expanding each element and showing equivalence. Here we give a simpler proof using a property of the Hadamard product, $(M \odot M')v = \operatorname{diag}\left(MD_v(M')^\top\right)$ where D_v constructs a diagonal matrix from vector v and $\operatorname{diag}(G)$ extracts the diagonal entries of matrix G.

Proof. In this case, the broadcasted multiplication $\circledast$ coincides with the Hadamard product $\odot$. Thus we show

$$(P \odot z_P)1_T^\top = \operatorname{diag}\left(Pz_P^\top\right) = \operatorname{diag}\left(PVz_O^\top\right) = \operatorname{diag}\left(Oz_O^\top\right) = (O \odot z_O)1_d^\top$$

□

Given Lemma 2.1, we can now compute $u_i = (O_i \circledast z_{O,i}).\mathbf{sum}(2) \in \mathbb{R}^{\Gamma\times 1}$ without P. Finally, we can accumulate the gradients z_Q, z_K, z_V in tiles since $M(M_1 \dots M_m) = \sum_i MM_i$ for any matrix $M \in \mathbb{R}^{d\times d}$ and $M_i \in \mathbb{R}^{d\times d/m}$ (e.g., $z_{Q,i}$ accumulates $\sum_j (z_{A,i,j}K_j)/\sqrt{d}$). Put together, the backward function is summarized below. We assume that Q, K, V, the computed quantities from the forward function c, π, O, and the gradient slots $z_O, z_Q, z_K, z_V \in \mathbb{R}^{T\times d}$ are in HBM.

FlashBackward

1. **Load** $c = (c_1 \dots c_B), \pi = (\pi_1 \dots \pi_B) \in \mathbb{R}^{T\times 1}$ **to SRAM.**
2. For $j = 1 \dots C$:
   (a) **Load** $K_j, V_j, z_{K,j}, z_{V,j} \in \mathbb{R}^{\Lambda\times d}$ **to SRAM.**
   (b) For $i = 1 \dots B$:
      i. **Load** $Q_i, O_i, z_{O,i}, z_{Q,i} \in \mathbb{R}^{\Gamma\times d}$ **to SRAM.**
      ii. Compute

$$P_{i,j} \leftarrow \exp\left(\frac{Q_i K_j^\top}{\sqrt{d}} \ominus c_i\right) \oslash \pi_i \in \mathbb{R}^{\Gamma\times\Lambda} \qquad z_{V,j} \leftarrow z_{V,j} + (P_{i,j})^\top z_{O,i} \in \mathbb{R}^{\Lambda\times d}$$

$$z_{P,i,j} \leftarrow z_{O,i}(V_j)^\top \in \mathbb{R}^{\Gamma\times\Lambda} \qquad z_{K,j} \leftarrow z_{K,j} + \frac{(z_{A,i,j})^\top Q_j}{\sqrt{d}} \in \mathbb{R}^{\Lambda\times d}$$

$$u_i \leftarrow (O_i \circledast z_{O,i}).\mathbf{sum}(2) \in \mathbb{R}^{\Gamma\times 1} \qquad z_{Q,i} \leftarrow z_{Q,i} + \frac{z_{A,i,j}K_j}{\sqrt{d}} \in \mathbb{R}^{\Gamma\times d}$$

$$z_{A,i,j} \leftarrow P_{i,j} \circledast (z_{P,i,j} \ominus u_i) \in \mathbb{R}^{\Gamma\times\Lambda}$$

      iii. **Write** $z_{Q,i} \in \mathbb{R}^{\Gamma\times d}$ **to HBM.**
   (c) **Write** $z_{K,j}, z_{V,j} \in \mathbb{R}^{\Lambda\times d}$ **to HBM.**
3. Return $z_Q, z_K, z_V \in \mathbb{R}^{T\times d}$.

The additional memory overhead is $O(\max(T, \Gamma\Lambda, \Gamma d, \Lambda d))$. It again makes $O(CTd) = O(\frac{T^2 d}{\Lambda})$ HBM accesses since it accesses Q, O, z_O, z_Q for C times in the inner loop. Thus choosing the chunk size $\Lambda = \Theta(\frac{M}{d})$ and the batch size $\Gamma = \Theta(\min(\frac{M}{d}, d))$ where M is the SRAM size again implies $O(\frac{T^2 d^2}{M})$ HBM accesses. See Appendix D for further refinement of FlashAttention.

3 KV Cache Reduction

In practice, we always use multi-head attention (MHA). MHA has trainable parameters $W_q, W_k, W_v, W_f \in \mathbb{R}^{d\times d}$. It assumes a number of heads H that divides d, yielding the head dimension $d_H := \frac{d}{H}$. Given input hidden states $X \in \mathbb{R}^{T\times d}$, it outputs $\widetilde{O} \in \mathbb{R}^{T\times d}$ by

$$\begin{aligned} [Q_1 \dots Q_H] &= XW_q & O_h &= \left(\frac{Q_h K_h^\top}{\sqrt{d_H}}\right).\mathbf{softmax}(2)V_h \quad \forall h \in \{1\dots H\} \\ K = [K_1 \dots K_H] &= XW_k & \widetilde{O} &= [O_1 \dots O_H]W_f \\ V = [V_1 \dots V_H] &= XW_v \end{aligned} \tag{7}$$

where $Q_h, K_h, V_h \in \mathbb{R}^{T\times d_H}$ are the H heads extracted from X. By exploiting memory contiguity (Appendix B.3), we can batch the heads to perform the operations efficiently and (almost) free of additional memory.[6] The cache size is $2Td$ (for storing $K, V \in \mathbb{R}^{T\times d}$).

3.1 Grouped Query Attention (GQA)

GQA reduces the number of heads in the KV cache. It assumes $H' = \frac{H}{R}$ heads for key and value which are then shared between a group of R query heads [10, 1]. The trainable parameters of GQA are $W_q, W_f \in \mathbb{R}^{d\times d}$ and $U_k, U_v \in \mathbb{R}^{d\times \frac{d}{R}}$ ($\frac{1}{2}(\frac{R+1}{R})$ of MHA). Given $X \in \mathbb{R}^{T\times d}$, it outputs $\widetilde{O} \in \mathbb{R}^{T\times d}$ by

$$\begin{aligned} [Q_1 \dots Q_H] &= XW_q & O_h &= \left(\frac{Q_h K_{\lceil \frac{h}{R} \rceil}^\top}{\sqrt{d_H}}\right).\mathbf{softmax}(2)V_{\lceil \frac{h}{R} \rceil} \quad \forall h \in \{1\dots H\} \\ K = [K_1 \dots K_{H'}] &= XU_k & \widetilde{O} &= [O_1 \dots O_H]W_f \\ V = [V_1 \dots V_{H'}] &= XU_v \end{aligned}$$

Since $K, V \in \mathbb{R}^{T\times \frac{d}{R}}$, the cache size is $\frac{2Td}{R}$ ($\frac{1}{R}$ of MHA). The contiguous query matching $h \mapsto \lceil \frac{h}{R} \rceil$ can be done efficiently.[7] GQA was initially developed as a post-training scheme to speed up decoding (i.e., replace MHA with GQA and finetune). Now, it is standard to just use GQA directly during training.

3.2 Multi-Head Latent Attention (MLA)

MLA reduces the dimension of the KV cache by imposing a low rank $m \ll d$ on the key and value embeddings [7]. The training parameters of MLA are $D_q, D_{kv} \in \mathbb{R}^{d\times m}$, $U_q, U_k, U_v \in \mathbb{R}^{m\times d}$, and $W_f \in \mathbb{R}^{d\times d}$ ($\frac{m}{2d} + \frac{1}{4}$ of MHA). Given $X \in \mathbb{R}^{T\times d}$, it computes MHA (7) with the rank-m heads

$$[Q_1 \dots Q_H] = XD_qU_q \qquad [K_1 \dots K_H] = \underbrace{XD_{kv}}_{Z}U_k \qquad [V_1 \dots V_H] = \underbrace{XD_{kv}}_{Z}U_v \tag{8}$$

Since $Z \in \mathbb{R}^{T\times m}$ is shared by key and value, the cache size is Tm ($\frac{m}{2d}$ of MHA). At test time, the additional cost of up-projecting Z can be avoided by absorbing U_k, U_v. This can be seen easily in the case $H = 1$. We precompute $E = U_qU_k^\top \in \mathbb{R}^{m\times m}$ and $F = U_vW_f \in \mathbb{R}^{m\times d}$. Then given query X, we load cache Z and compute

$$\widetilde{O} = \left(\frac{XD_qEZ^\top}{\sqrt{d}}\right).\mathbf{softmax}(2)ZF$$

The case $H > 1$ is similar.

[6] For instance, $\mathbf{Q} = (XW_q).\mathbf{view}(T, H, d_H).\mathbf{transpose}(1, 2)$ creates a batch $\mathbf{Q} \in \mathbb{R}^{H\times T\times d_H}$ of H queries with dimension d_H without making any copy of the tensor. Then we can compute vanilla attention (4) by broadcasting to obtain the batched output $\mathbf{O} \in \mathbb{R}^{H\times T\times d_H}$. For the final step of matrix multiplication with $W_f \in \mathbb{R}^{d\times d}$, we can compute $\widetilde{O} \leftarrow \mathbf{O}.\mathbf{transpose}(1, 2).\mathbf{reshape}(T, d)W_f$ where we need to **reshape** the tensor instead of **view** since **transpose** creates a non-contiguous tensor. This creates a copy, incurring $O(Td)$ additional memory.

[7] If $\mathbf{K} \in \mathbb{R}^{H'\times T\times d_H}$ denotes the H' key heads in batch form, then $\mathbf{K'} = \mathbf{K}.\mathbf{expand}(H', R, T, d_H).\mathbf{reshape}(H, T, d_H)$ repeats each key embedding for R times (**expand** creates a non-contiguous tensor). After this step, computation is the same as MHA.

3.2.1 MLA with RoPE

We cannot absorb the up-projections to each other when we use the RoPE embedding [11]. This is because RoPE applies some transformation $R : \mathbb{R}^{T\times d_H} \to \mathbb{R}^{T\times d_H}$ to the already up-projected heads Q_h, K_h. Thus MLA computes separate embeddings for RoPE. Specifically, it uses additional parameters $V_q \in \mathbb{R}^{m\times Hp}$ and $D_k \in \mathbb{R}^{d\times p}$ to compute the p-dimensional decoupled query-key embeddings: [8]

$$[Q_{\text{RoPE},1} \dots Q_{\text{RoPE},H}] = XD_qV_q$$
$$K_{\text{RoPE}} = XD_k$$

The choice to share the key embedding between heads is deliberate (to reduce the cache size). The final query and key heads are redefined to be $\bar{Q}_h = [Q_h \; R(Q_{\text{RoPE},h})], \bar{K}_h = [K_h \; R(K_{\text{RoPE}})] \in \mathbb{R}^{T\times(d+p)}$ where $Q_h, K_h \in \mathbb{R}^{T\times d}$ are the same up-projections in (8). (Attention only cares about the query-key scores; their dimensions are irrelevant.) With this definition, the query-key scores take the form

$$\bar{Q}_h\bar{K}_h^\top = Q_hK_h^\top + R(Q_{\text{RoPE},h})R(K_{\text{RoPE}})^\top$$

so we can compute the first term using the absorption trick as before. The second term takes care of relative positional encoding. The cache size is now $T(m + p)$ to additionally store $K_{\text{RoPE}} \in \mathbb{R}^p$.

References

[1] Ainslie, J., Lee-Thorp, J., de Jong, M., Zemlyanskiy, Y., Lebron, F., and Sanghai, S. (2023). Gqa: Training generalized multi-query transformer models from multi-head checkpoints. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 4895–4901.

[2] Dao, T. (2024). Flashattention-2: Faster attention with better parallelism and work partitioning. In *The Twelfth International Conference on Learning Representations*.

[3] Dao, T., Fu, D., Ermon, S., Rudra, A., and Ré, C. (2022). Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, **35**, 16344–16359.

[4] Ivanov, A., Dryden, N., Ben-Nun, T., Li, S., and Hoefler, T. (2021). Data movement is all you need: A case study on optimizing transformers. *Proceedings of Machine Learning and Systems*, **3**, 711–732.

[5] Jang, H., Kim, J., Jo, J.-E., Lee, J., and Kim, J. (2019). Mnnfast: A fast and scalable system architecture for memory-augmented neural networks. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 250–263.

[6] Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. (2023). Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 611–626, New York, NY, USA. Association for Computing Machinery.

[7] Liu, A., Feng, B., Wang, B., Wang, B., Liu, B., Zhao, C., Dengr, C., Ruan, C., Dai, D., Guo, D., *et al.* (2024). Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434*.

[8] Milakov, M. and Gimelshein, N. (2018). Online normalizer calculation for softmax. *arXiv preprint arXiv:1805.02867*.

[9] Rabe, M. N. and Staats, C. (2021). Self-attention does not need $o(n^2)$ memory. *arXiv preprint arXiv:2112.05682*.

[10] Shazeer, N. (2019). Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150*.

[11] Su, J., Ahmed, M., Lu, Y., Pan, S., Bo, W., and Liu, Y. (2024). Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, **568**, 127063.

[8] In practice, we combine this with computing $Q = [Q_1 \dots Q_H]$ and Z by $[Q \; Q_{\text{RoPE}}] = XD_q\bar{U}_q$ and $[Z \; K_{\text{RoPE}}] = X\bar{D}_{kv}$ using the parameters $\bar{U}_q \in \mathbb{R}^{m\times(d+Hp)}$ (i.e., $[U_q \; V_q]$) and $\bar{D}_{kv} \in \mathbb{R}^{m\times(d+p)}$ (i.e., $[D_{kv} \; D_k]$).

A Broadcasting Notation

We use the following binary operators for broadcasting: elementwise addition $\oplus$, elementwise subtraction $\ominus$, elementwise division $\oslash$, elementwise multiplication $\circledast$, elementwise division $\oslash$, pairwise maximum **max**, matrix multiplication **matmul**. If broadcasting is not needed, we use the standard mathematical notation (e.g., a matrix $A \in \mathbb{R}^{m \times n}$ divided by a constant $c \in \mathbb{R}$ is written as $\frac{A}{c} \in \mathbb{R}^{m \times n}$, standard matrix multiplication between $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{n \times p}$ is written as $AB \in \mathbb{R}^{m \times p}$). Remember the rules of broadcasting (poet: Sasha Rush):

There is a rule you should learn at long last,
combination of tensors the task.
Dims right-aligned,
extra left 1s assigned,
match paired dimensions: Broadcast!

9 x 1 x 3
1 x 8 x 1
---
9 x 8 x 3

For example, if $A \in \mathbb{R}^{9 \times 1 \times 3}$ and $B \in \mathbb{R}^{8 \times 1}$, the result $C \leftarrow A \oplus B \in \mathbb{R}^{9 \times 8 \times 3}$ contains $C_{i,j,k} = A_{i,1,k} + B_{j,1}$; $C \leftarrow \textbf{max}(A, B) \in \mathbb{R}^{9 \times 2 \times 3}$ contains $C_{i,j,k} = \max(A_{i,1,k}, B_{i,j,1})$. Matrix multiplication broadcasts over dimensions excluding the last two: if $A \in \mathbb{R}^{9 \times 1 \times 3 \times 2}$ and $B \in \mathbb{R}^{8 \times 2 \times 4}$, the result $C \leftarrow \textbf{matmul}(A, B) \in \mathbb{R}^{9 \times 8 \times 3 \times 4}$ contains matrices $C_{i,j} = A_{i,1} B_j \in \mathbb{R}^{3 \times 4}$ where tensors are sliced left to right. We write boldfaced $A.\textbf{max}(i)$ and $A.\textbf{sum}(i)$ to denote the maximum and summation along the i-th axis of A. When no axis is provided, all axes are used. We keep the collapsed dimension to preserve the number of dimensions. For example, if $A \in \mathbb{R}^{3 \times 5 \times 2}$, then $A.\textbf{max}(3) \in \mathbb{R}^{3 \times 5 \times 1}$. See Appendix B for an overview of how tensors are stored in memory.

B How Tensors Are Stored in Memory

B.1 Memory Tape

Conceptually, memory (of either CPU or GPU) can be viewed as a long sequence ("tape") of bytes. The CPU memory (RAM) is managed by the operating system; the GPU memory is managed by the GPU itself. Modern addressing systems are 64-bit, yielding 2^{64} addressable bytes (i.e., 16 billion GBs of theoretically possible memory size). A **pointer** is the index on the memory tape, representing the unique address of the referred byte (e.g., the pointer value `0xffe2ac6c` refers to the 4,293,045,356-th byte in hexadecimal notation).

In this section only, we assume tensor indexing from 0 for convenience. We use the standard range notation $\text{range}(n) = [0, 1, \ldots, n-1]$. We can stride by the number of bytes needed for the considered data type to address values (stored contiguously in memory) in the data type. For instance, if $u = (u_0, u_1, \ldots, u_7)$ is a vector of 8 `float32` numbers, and u_{ptr} is a pointer to u, the address of the i-th element u_i is

$$\text{address}(u_i) = u_{\text{ptr}} + \texttt{sizeof(float32)}\, i \qquad \forall i \in \text{range}(8)$$

where `sizeof(float32)` $= 4$. In languages like Triton, the data loading functions automatically take care of index scaling for the considered data type (e.g., $u_i = \texttt{load}(u_{\text{ptr}} + i)$ performs scaling under the hood). Thus for our purposes, we will view memory as a tape of floats, even though each float takes up 4 bytes.

B.2 Flattening

To store multi-dimensional tensors in memory, they must be flattened. Two popular schemes are (i) row-major (NumPy, PyTorch) and (ii) column-major (Matlab, Fortran). For instance, the matrix $A = [[a_{1,1}, a_{1,2}]; [a_{2,1}, a_{2,2}]] \in \mathbb{R}^{2 \times 2}$ is flattened as $[a_{1,1}, a_{1,2}, a_{2,1}, a_{2,2}] \in \mathbb{R}^4$ under row-major ordering and $[a_{1,1}, a_{2,1}, a_{1,2}, a_{2,2}] \in \mathbb{R}^4$ under column-major ordering. They have different implications in efficiency, but we will assume row-major. If we have a 3-dimensional tensor like $A \in \mathbb{R}^{m \times n \times p}$ at A_{ptr}, the value $A_{i,j,k}$ can be accessed as[9]

$$A_{i,j,k} = \texttt{load}(A_{\text{ptr}} + (np)i + (p)j + k) \qquad \forall i \in \text{range}(m),\ j \in \text{range}(n),\ k \in \text{range}(p)$$

[9] Using broadcasting, we can load the entire tensor as

$$A = \texttt{load}(A_{\text{ptr}} \oplus (n \times p \times \text{range}(m)).\textbf{view}(m, 1, 1) \oplus (p \times \text{range}(n)).\textbf{view}(1, n, 1) \oplus (\text{range}(p)).\textbf{view}(1, 1, p))$$

The multipliers of the axes (here, $(np, p, 1)$) are called **strides**. The stride of an axis specifies how many values in the memory tape to skip over to reach the next element along that axis (e.g., we need to skip over np floats to get to the next matrix slice in A). We can check the strides of a tensor in PyTorch like this:

```
>>> x = torch.tensor([[1, 2, 3], [4, 5, 6]])
>>> x.stride()  # (3, 1)
```

B.3 Contiguity and Memory-Free Tensor Operations

A tensor is called **contiguous** if its elements are accessed sequentially in memory without gaps when we traverse it in the order of increasing axis. More formally, a tensor $A \in \mathbb{R}^{d_1 \times \cdots \times d_m}$ with memory strides $(s_1 \dots s_m)$ is contiguous iff $s_m = 1$ and $s_i = d_{i+1} \times s_{i+1}$.[10] Note that this implies $s_1 \geq \cdots \geq s_m$. Any tensor is contiguous when first created. Contiguity is important for performance, e.g., CPUs and GPUs are optimized for sequential memory accesses (e.g., cache locality, prefetching mechanisms).

If the tensor is contiguous, we can freely change its strides by the **view** operator, thereby merging or spliting neighboring dimensions without making a copy or losing contiguity (by setting the strides as $s_m = 1$ and $s_i = d_{i+1} \times s_{i+1}$). No copy of x is made below and y is contiguous.

```
>>> x = torch.arange(12) + 1  # x.stride() => (1,)
>>> x
tensor([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
>>> y = x.view(2, 2, 3)  # y.stride() => (6, 3, 1)
>>> y
tensor([[[ 1,  2,  3],
         [ 4,  5,  6]],
        [[ 7,  8,  9],
         [10, 11, 12]]])
```

There are memory-free tensor operations that yield non-contiguous tensors. **Transpose** is a classical example. It is easy to see that transposing two axes corresponds to switching two strides. Thus no copy of x is made below, but y is not contiguous. In particular, we cannot view y in different shapes.[11]

```
>>> x = (torch.arange(12) + 1).view(4, 3)  # x.stride() => (3, 1)
>>> x
tensor([[ 1,  2,  3],
        [ 4,  5,  6],
        [ 7,  8,  9],
        [10, 11, 12]])
>>> y = x.transpose(0, 1)  # y.stride() => (1, 3), non-contiguous
>>> y
tensor([[ 1,  4,  7, 10],
        [ 2,  5,  8, 11],
        [ 3,  6,  9, 12]])
>>> x.view(12)  # Fine
tensor([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
>>> y.view(12)  # Error!
>>> y.reshape(12)  # Returns a copy in contiguous memory
tensor([ 1,  4,  7, 10,  2,  5,  8, 11,  3,  6,  9, 12])
```

Another example is **expand**, which expands singleton dimensions to larger sizes without making a copy of the tensor. This is achieved by setting their strides to 0, thereby "staying put" while striding (i.e., repeat). This also makes the tensor non-contiguous.

[10] This way, the last axis is contiguous; the axis i striding the entire contiguous axis $i + 1$ in a row-major fashion is contiguous.

[11] If we wish to view a non-contiguous tensor as a different shape in PyTorch, we can use the shortcut **reshape** which creates a copy if non-contiguous.

```
>>> x = (torch.arange(12) + 1).view(1, 12)  # x.stride() => (12, 1)
>>> x
tensor([[ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12]])
>>> y = x.expand(3, 12)  # y.stride() => (0, 1)
>>> y  # Non-contiguous
tensor([[ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12],
        [ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12],
        [ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12]])
```

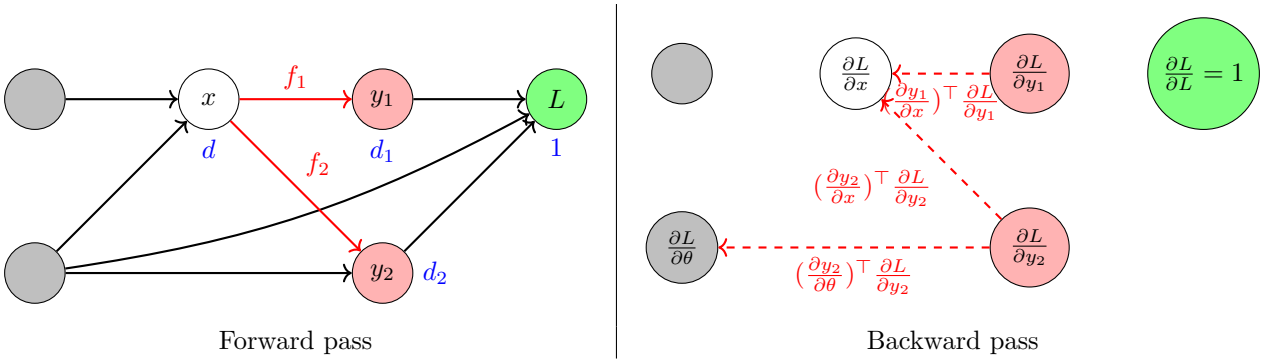
Finally, some operations must make a copy of the input. For instance, concatenating two input tensors A, B along a specified dimension i (i.e., $\mathbf{cat}(A, B, i)$) must make a copy of A and B, since they may reside in completely different locations in memory, making it impossible to stride over them.

C A Quick Review of Backprop

A loss $L \in \mathbb{R}$ on a batch is the final node of a DAG whose root nodes are inputs and parameters; internal nodes are hidden states. The goal is to compute the gradient of L with respect to all nodes. A node $x \in \mathbb{R}^d$ affects L only through its children $y_k = f_k(x) \in \mathbb{R}^{d_k}$ where $f_k : \mathbb{R}^d \to \mathbb{R}^{d_k}$ is some differentiable function. By the chain rule

$$\frac{\partial L}{\partial x} = \sum_{k=1}^{K} \left(\frac{\partial y_k}{\partial x}\right)^\top \frac{\partial L}{\partial y_k} \tag{9}$$

where $(\frac{\partial y_k}{\partial x})^\top \in \mathbb{R}^{d \times d_k}$ is the (transposed) Jacobian of f_k with respect to x (i.e., $(\frac{\partial y_k}{\partial x})^\top_{i,j} = \frac{\partial y_{k,j}}{\partial x_i}$).[12] **Backprop** computes (9) for all nodes by traversing the DAG from L in a *reverse* topological order and at each node *accumulating* the Jacobian-gradient product to all its parents' gradient slots (initialized to zeros). This works because of the DAG structure. In the example, before reaching the node x, we will have finished accumulating $\frac{\partial L}{\partial x}$ (thus the value will be correct):



A DAG is built by predefined operators that specify (1) **forward**: how to map parent tensors to an output tensor, and (2) **backward**: how to compute the Jacobian-gradient product for each parent. For example, if $w = f(u, v)$ is a node created by the operator $f(x, y) = \mathrm{ReLU}(x \circledast y) \in \mathbb{R}^d$ with $z = \frac{\partial L}{\partial w} \in \mathbb{R}^d$, the backward function accumulates $(\frac{\partial w}{\partial u})^\top z$ and to the gradient slot of u and $(\frac{\partial w}{\partial v})^\top z$ and to the gradient slot of v. In PyTorch,

[12] The sum over children is consistent with the "normal" chain rule without the sum if we view x as affecting L through a single node $y = (y_1 \dots y_K) \in \mathbb{R}^{d_1 + \dots + d_K}$ so that $\left(\frac{\partial y}{\partial x}\right)^\top \frac{\partial L}{\partial y} = \sum_{k=1}^{K} \left(\frac{\partial y_k}{\partial x}\right)^\top \frac{\partial L}{\partial y_k}$.

```python
class Op(torch.autograd.Function):

    @staticmethod
    def forward(ctx, x, y):
        ctx.save_for_backward(x, y, x * y > 0)
        return (x * y > 0) * x * y

    @staticmethod
    def backward(ctx, z):  # z is the grad wrt. the node
        x, y, inds = ctx.saved_tensors
        return inds * y * z, inds * x * z


u = torch.randn((8,), requires_grad=True)
L = (Op.apply(u, u) + 3 * u + u.exp()).sum()  # Loss
L.backward()  # Computes u.grad
```

```python
class Layer(torch.nn.Module):
    """Computes ReLU(w * x) with a learnable w"""

    def __init__(self, d):
        super().__init__()
        self.w = torch.nn.Parameter(torch.empty(d))
        torch.nn.init.uniform_(self.w, -0.1, 0.1)

    def forward(self, x):
        return Op.apply(self.w, x)


layer = Layer(8)
u = torch.randn((8,), requires_grad=True)
L = layer(layer(u)).sum()  # Loss
L.backward()  # Computes u.grad, layer.w.grad
```

C.1 Tips and Examples

Matrix multiplication. Treat a matrix of shape $m \times n$ as a vector of length mn. We can then derive

$$\underbrace{C}_{m\times p} = \underbrace{A}_{m\times n}\underbrace{B}_{n\times p}: \qquad \underbrace{z_A}_{m\times n} \leftarrow z_A + \underbrace{z_C}_{m\times p}\underbrace{B^\top}_{p\times n} \qquad \underbrace{z_B}_{n\times p} \leftarrow z_B + \underbrace{A^\top}_{n\times m}\underbrace{z_C}_{m\times p}$$

Independent dimensions. If the dimensions along an axis are independent, the Jacobian is diagonal and the backward function can treat the dimensions as independent transformations. In the previous example, we had

$$t = \mathrm{ReLU}(x \circledast y) \;\Leftrightarrow\; t_i = \max(0, x_i y_i): \qquad z_{x,i} \leftarrow z_{x,i} + \mathbb{1}(x_i y_i > 0) y_i z_{t,i} \qquad z_{y,i} \leftarrow z_{y,i} + \mathbb{1}(x_i y_i > 0) x_i z_{t,i}$$

The independent transformations can be multi-dimensional, e.g., batched matrix multiplication

$$\underbrace{C}_{N\times m\times p} = \mathbf{matmul}(\underbrace{A}_{N\times m\times n}, \underbrace{B}_{N\times n\times p}): \qquad \underbrace{z_A}_{N\times m\times n} \leftarrow z_A + \mathbf{matmul}(\underbrace{z_C}_{N\times m\times p}, \underbrace{B.\mathbf{transpose}(2,3)}_{N\times p\times n})$$

Softmax. If $p = \mathrm{softmax}\,(x) \in \mathbb{R}^d$, we can verify $\frac{\partial p_j}{\partial x_i} = p_i(\mathbb{1}(i=j) - p_j)$ which implies

$$z_{x,i} \leftarrow z_{x,i} + p_i z_{p,i} - \left(\sum_{j=1}^{d} p_j z_{p,j}\right) p_i \qquad \Leftrightarrow \qquad z_x \leftarrow z_x + p \circledast z_p - (p \circledast z_p).\mathbf{sum}()\; p \tag{10}$$

The shifted softmax $p = \mathrm{softmax}\,(x - x.\mathbf{max}()) \in \mathbb{R}^d$ has the same backward function (10) (hint: the gradient wrt. $x.\mathbf{max}()$ vanishes by the property of softmax). The row-wise softmax $P = X.\mathbf{softmax}(2) \in \mathbb{R}^{N\times d}$ where we apply (shifted) softmax over the rows of $X \in \mathbb{R}^{N\times d}$ independently is then

$$z_X \leftarrow z_X + P \circledast z_P - \underbrace{(P \circledast z_P).\mathbf{sum}(2)}_{N\times 1} \circledast \underbrace{P}_{N\times d} = z_X + P \circledast (z_P \ominus (P \circledast z_P).\mathbf{sum}(2)) \tag{11}$$

D Further Optimization on FlashAttention

FlashAttention-2 [2] first notes that the incremental updates in **FlashForward** telescope and allow us to simplify the calculation. Assume a single batch $B = 1$ for simplicity so that we update $O^{(0)} \leftarrow 0^{T\times d}, O^{(1)}, \ldots, O^{(C)}$ for C chunks. The incremental update is of the form (by v^{-1} we mean elementwise inversion $v_i^{-1} = 1/v_i$):

$$O^{(j)} = (\pi^{(j)})^{-1} \circledast \left(\pi^{(j-1)} \circledast u^{(j)} \circledast O^{(j-1)} + B_j V_j\right)$$

for some $u^{(j)} \in \mathbb{R}^{T\times 1}$ and $B_j \in \mathbb{R}^{T\times\Lambda}$. We see that

$$\begin{aligned}
O^{(1)} &= (\pi^{(1)})^{-1} \circledast B_1 V_1 \\
O^{(2)} &= (\pi^{(2)})^{-1} \circledast \left(u^{(2)} \circledast B_1 V_1 + B_2 V_2\right) \\
O^{(3)} &= (\pi^{(3)})^{-1} \circledast \left(u^{(3)} \circledast u^{(2)} \circledast B_1 V_1 + u^{(3)} \circledast B_2 V_2 + B_3 V_3\right) \\
&\vdots
\end{aligned}$$

Thus we can instead maintain $G^{(0)} \leftarrow 0^{T\times d}$, $G^{(j)} = u^{(j)}G^{(j-1)} + B_jV_j$, finally setting $O^{(C)} = (\pi^{(C)})^{-1}G^{(C)}$. This saves us $O(CT)$ FLOPs. Next, we note that we do not need to save both $c, \pi \in \mathbb{R}^{T\times 1}$ in the forward pass, since in the backward pass we only compute

$$P_j = \exp\left(A_j \ominus c\right) \oslash \pi = \exp(A_j \ominus (c + \log \pi))$$

So we can only save $\xi := c + \log \pi \in \mathbb{R}^{T\times 1}$, saving HBM access by $O(T)$. Finally, we move the embarrassingly parallel batch axis (i.e., $i = 1 \dots B$) from the inner loop to the outer loop for better parallelization. This gives the forward function of FlashAttention-2. Note that writing to O_i has moved out of the inner loop.

FlashForward2

1. For $i = 1 \dots B$: (embarrassingly parallel)
   (a) **Load $Q_i, O_i \in \mathbb{R}^{\Gamma\times d}$ to SRAM.**
   (b) **Initialize $c_i \leftarrow -\infty^{\Gamma\times 1}, \pi_i \leftarrow 0^{\Gamma\times 1}, G_i \leftarrow 0^{\Gamma\times d}$ in SRAM.**
   (c) For $j = 1 \dots C$:
      i. **Load $K_j, V_j \in \mathbb{R}^{\Lambda\times d}$ to SRAM.**
      ii. Compute

$$A_{i,j} \leftarrow \frac{Q_iK_j^\top}{\sqrt{d}} \in \mathbb{R}^{\Gamma\times\Lambda} \qquad c_i^{\text{new}} \leftarrow \max\left(c_i, A_{i,j}.\textbf{max}(2)\right) \in \mathbb{R}^{\Gamma\times 1}$$

$$\pi_i^{\text{new}} \leftarrow \exp(c_i - c_i^{\text{new}}) \circledast \pi_i + \exp(A_{i,j} \ominus c_i^{\text{new}}).\textbf{sum}(2) \in \mathbb{R}^{\Gamma\times 1}$$

$$G_i^{\text{new}} \leftarrow \exp(c_i - c_i^{\text{new}}) \circledast G_i + \exp(A_{i,j} \ominus c_i^{\text{new}})V_j \in \mathbb{R}^{\Gamma\times d}$$

      iii. Update $c_i \leftarrow c_i^{\text{new}}$, $\pi_i \leftarrow \pi_i^{\text{new}}$, $G_i \leftarrow G_i^{\text{new}}$.
   (d) **Write $O_i \leftarrow G_i \oslash \pi_i \in \mathbb{R}^{\Gamma\times d}$ and $\xi_i \leftarrow c_i + \log \pi_i \in \mathbb{R}^{\Gamma\times 1}$ to HBM.**
2. Return $O \in \mathbb{R}^{T\times d}$ in HBM.

The backward function remains the same except that the attention matrix uses ξ_i (i.e., $P_{i,j} \leftarrow \exp(\frac{Q_iK_j^\top}{\sqrt{d}} - \xi_i)$). While the outer loop is not embarrassingly parallel, atomic adds (hardware-level locking mechanism) are used to parallelize it. FlashAttention-2 includes many other optimization tricks. One is exploiting causal masking by realizing that we can skip a half of the tiles (furthermore, for the computed batch, only the last chunk needs masking). Another is work partitioning between "warps" (groups of 32 threads within a thread block); Q is split across warps while K, V are shared to reduce inter-warp communication. FlashAttention-3 further refines the algorithm by using GPU-architecture-specific techniques for asynchronous execution and quantization.