Flash Attention

July 23, 2024

Contents

:: is read as ”have a type of”.

$\rightarrow$ is read as ”maps to”.

1 Background: The Computational Process of Reduce and Map

A *list* is a linearly ordered collection of values. All the elements of a given list must have the same type. If the elements of a list all have the type α, then the list itself has the type of $[\alpha]$.

1.1 The *reduce* operator

The ***reduce*** operator is a high-order list processing function where a list is aggregated and reduced down to a single value. It is defined as follows:

$$\begin{aligned} &\textit{reduce} :: (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow ([\alpha] \rightarrow \beta) \\ &\textit{reduce} \ \oplus \ v \ [] = v \\ &\textit{reduce} \ \oplus \ v \ (x : xs) = \oplus \ x \ (\textit{reduce} \ \oplus \ v \ xs) \end{aligned}$$

The reduce operator takes a binary operator $\oplus$ of type $\alpha \rightarrow \beta \rightarrow \beta$, an initial value v of type β, and a list of values xs of type $[\alpha]$ as input, and returns a single value of β obtained by applying $\oplus$ to the initial value and the first element of the list, then to the result of that operation and the second element of the list, and so on, until all elements have been processed.

1.2 The *map* operator

$$\begin{aligned} \textit{map} &:: (\alpha \rightarrow \beta) \rightarrow ([\alpha] \rightarrow [\beta]) \\ \textit{map} \ f \ xs &= [f \ x_1 \ldots f \ x_n] \end{aligned}$$

The *map* operation applies a unary function f of type $\alpha \rightarrow \beta$ to each element of a list. Because there is no way to communicate among each evaluation of f, the underlying implementation of the *map* operation can execute the evaluations of f over list elements in any order it chooses.

2 A Generalized *Broadcast-and-then-Aggregate* Operation

2.1 The *broadcast-and-then-aggregate* operation

We consider a generalized *broadcast-and-then-aggregate* operation that has the following standard form:

```
function BroadcastAndThenAggregate ($f :: (\alpha \rightarrow \gamma \rightarrow \beta), c :: \gamma, \oplus :: (\beta \rightarrow \theta \rightarrow \theta)$, $s_0 :: \theta$, $xs :: [\alpha]$)
  $s = s_0$  // initialize the accumulator
  foreach($x$ in $xs$)
    $s = \oplus(s, f(x, c))$  // broadcast c and accumulate s
  return s
```

The *broadcast-and-then-aggregate* operation is defined by the quadruple $R =< f, c, \oplus, s_0 >$. f is a function of type $\alpha \rightarrow \gamma \rightarrow \beta$, c is a constant value of type γ, $\oplus$ is a binary operator of type $\beta \rightarrow \theta \rightarrow \theta$. s is an accumulator of type θ that store the aggregated value, and are initialized to s_0.

The computational process proceeds by enumerating each input value x from the input list xs, broadcasting c to x by applying $y = f(x, c)$, and then aggregating s according to $s = s \oplus y$. It is important to note that *there is no communication among the evaluations of* $f(x, c)$.

We can use the diagram below to visualize this operation.

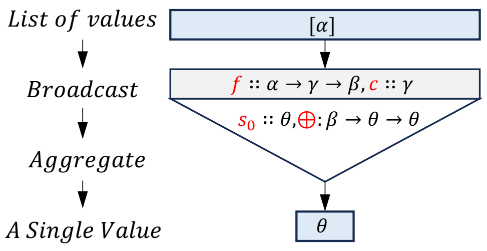


Figure 1: The diagram to visualize the *broadcast-and-then-aggregate* operation.

2.2 The running example: attention in *Broadcast-and-then-Aggregate*

Neural network computations can be expressed uniformly as a dataflow graph of broadcast-and-then-aggregate operations. As a concrete example, we can express attention using a set of broadcast-and-then-aggregate operatiosn. Figure 2 shows the corresponding dataflow graph.

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

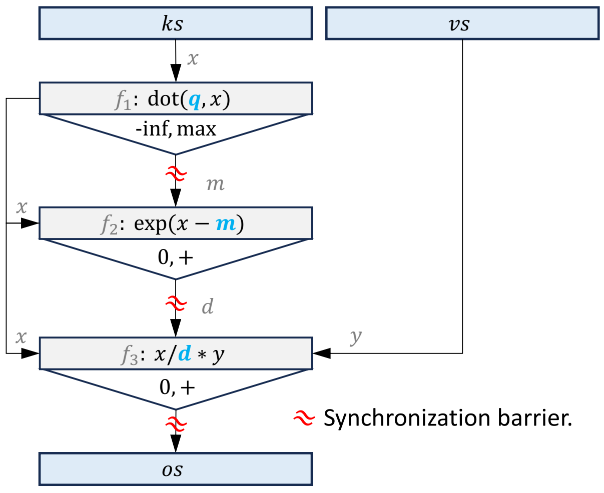


Figure 2: Express attention function using *broadcast-and-then-aggregate*.

3 Block Execution of a Chain of *Broadcast-and-then-Aggregate*

We use $\bar{x} = [x_1, \ldots, x_n]$ to represent a list of values, and $\bar{x}_1 : \bar{x}_2$ to represent the concatenation of $\bar{x}_1$ and $\bar{x}_2$. $f(\bar{x}) = [f(x_1, c), \ldots, f(x_n, c)]$. $s_0 \oplus \bar{x} =$ *reduce* $s_0\ \bar{x}$ where s_0 and $\oplus$ are as defined in section 1.

3.1 Motivations

Blocked execution of a single *broadcast-and-then-aggregate*. When the input list $\bar{x}$ is too large to be processed within hardware constraints, We need to split it into portions that can be handled by the hardware. By definition, the *broadcast-and-then-aggregate* operation can be executed portion by portion. This means that the following equation holds:

$$s_0 \oplus f(\bar{x}, c) = (s_0 \oplus f(\bar{x}_1, c)) \oplus (s_0 \oplus f(\bar{x}_2, c)) \quad \text{where} \;\; \bar{x} = \bar{x}_1 : \bar{x}_2 \tag{2}$$

Assuming that the input list $\bar{x}$ is divided into two segments, $\bar{x}_1$ and $\bar{x}_2$, the broadcast stage can independently evaluate the function f on each segment. This results in $\bar{y}_1 = f(\bar{x}_1, c)$ and $\bar{y}_2 = f(\bar{x}_2, c)$, which can be executed in parallel. The reduce operation is recursively defined and aggregates each segment independently to obtain partial aggregated results. For example, $s_1 = s_0 \oplus \bar{y}_1$ and $s_2 = s_0 \oplus \bar{y}_2$. These partial results are then recursively combined until a single aggregated value is obtained, such as $s = s_1 \oplus s_2$. ***If the operator $\oplus$ is both associative and commutative, the partial reduction can be computed in parallel.***

Regarding a single *broadcast-and-then-aggregate* operation, the final aggregated result depends on completing the scan of all the list inputs. This makes the reduce operation a barrier to proceeding further. Furthermore, if the output of the broadcast stage is required for subsequent computations, it must be stored in memory. This requires storage proportional to the length of the input sequence.

Motivations for blocked execution for a chain of *map-and-then-aggregate* operations. If the input list is very long, the time it takes to scan the input and write the results of the broadcast stage (if needed) will be the dominant factor in the overall latency of the *map-and-then-aggregate* operation.

If the entire computational process can be executed in parallel, with each portion computing a partial result, an efficient implementation can exploit the memory hierarchy by caching segments of input list in high-speed memory. Computation can then continue in high-speed memory until a synchronization barrier is met, at which point partial results must be stored in slow memory.

In the *broadcast-and-then-aggregate* operation, the *broadcast* stage requires storage proportional to the length of the input to store its output, while the *reduce* operation aggregates input to a smaller value that consumes less storage for its output. If a chain of *broadcast-and-then-aggregate* can proceed portion by portion in high-speed memory until the results of the broastcast stage are no longer needed by subsequent computation (and therefore do not need to be stored), both I/O complexity and memory footprint can be reduced.

We use the attention example shown in Figure 2 as our running example. There are three *broadcast-and-then-aggregate* operations involved. Each can execute internally in parallel, block by block. However, the three *broadcast-and-then-aggregate* operations must execute sequentially since there are data dependences among them. Additionally, the result of the first broadcast operation is required by the second broadcast operation, and the results of the second broadcast operation are required by the third broadcast operation. Sequentially execute these three *broadcast-and-then-aggregate* requires two storages that are proportional to the input list.

```
// stage1: local reducer computes partial results on individual blocks
$o_1$ = Attention($q, ks_1, vs_1$)      $o_2$ = Attention($q, ks_2, vs_2$)

// stage2: combine partial results
$o_{\text{new}}$ = Combiner($o_1, o_2$)
```

After the last *broadcast-and-then-aggregate* operation, a noticeable decrease in storage size was observed. Therefore, we anticipate that the entire computational process of equation (1) be executed in two stages. In the first stage, the input will be split into portions and partial results that require storage much smaller than the original input list will be computed without regard to original data dependencies among dependent *broadcast-and-then-reduce* operations. By selecting an appropriate block size, the entire computational process can be stored in high-speed memory. In the second stage, a combiner will merge these partial results to obtain the correct final aggregated value.

The problems are:

1. Under what conditions (necessary and sufficient conditions) can dependent *broadcast-and-then-aggregate* operations be decomposed into these two stages?
2. The existence of a combiner that can compute the correct final result.
3. If such a combiner exists, how can it be constructed?

3.2 A (informal) Problem Statement

Given two dependent *broadcast-and-then-aggregate* operations:

$$
\begin{aligned}
r &= s_0 \oplus_1 f_1(\bar{x} - c) \\
o &= s_1 \oplus_2 f_2(\bar{y} - r)
\end{aligned}
$$

Design a combiner, denoted by C, that can produce an output $\bar{\bar{o}}$ equivalent to o when computed using the following formula:

$$
\begin{aligned}
r_1 &= s_0 \oplus_1 f_1(\bar{x}_1, c) & \qquad r_2 &= s_0 \oplus_1 f_1(\bar{x}_2, c) \\
o_1 &= s_1 \oplus_2 f_2(\bar{y}_1, r_1) & \qquad o_2 &= s_1 \oplus_2 f_2(\bar{y}_2, r_2) \\
\bar{\bar{o}} &= C(r_1, r_2, o_1, o_2)
\end{aligned}
$$

Note that a combiner $C(r_1, \bar{y}_1, r_2, \bar{y}_2)$ always exists. This is because r can be computed as $r = r_1 \oplus r_2$, and then o can be re-computed using the formula $o = s_1 \oplus_2 f_2(\bar{y}_1 : \bar{y}_2, r)$. However, implementing such a combiner would require storing the entire $\bar{y}$ and re-computing the second *broadcast-and-aggregate*, which is not practical.

In order to reduce both IO complexity and the storage requirements, we must ensure that the combiners only depend on the outputs of the *reduce* operations, namely r_1, r_2, o_1, and o_2.

3.3 Preliminary Thinking to the Problem

Decomposable Broadcast-and-then-Aggregate Operation. We say that a *broadcast-and-then-aggregate* operation $R =< F, c, s_0, \oplus >$ is decomposable under $\circ$ if there exists functions h and $\odot$ such that the following equation holds:

$$
s_0 \oplus F(\bar{x}, c \circ \triangle c) = (s_0 \oplus F(\bar{x}, c)) \odot h(c, \triangle c) \tag{3}
$$

In other words, a decomposable *broadcast-and-then-aggregate* allows to broadcast a value $c \circ \triangle c$ in two steps. First, broadcast c to the input list using F, and then aggregate the broadcast result using $\oplus$. Second, adjust the result of the first step with an adjusting function $h(c, \triangle c)$ using $\odot$.

If a *broadcast-and-then-aggregate* operation can be decomposed, then a combiner C exists by the definition. Since this definitive equation(3) is a strict necessary condition, and the problem is then reduced

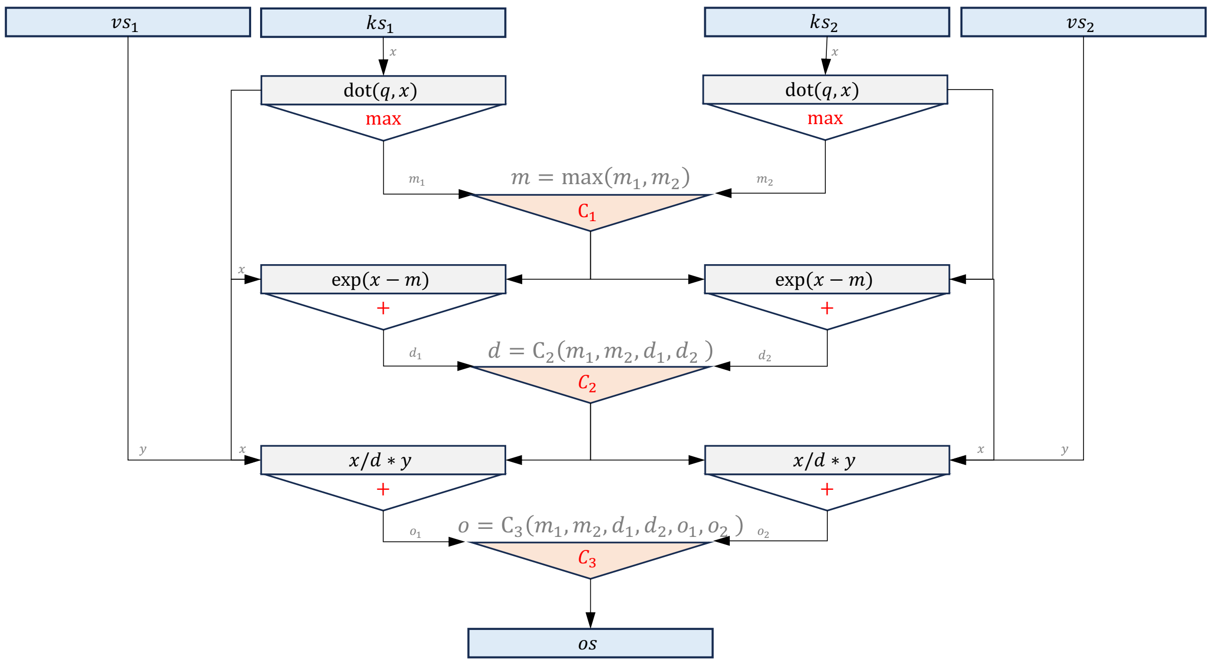


Figure 3: The expected blocked execution of a chain of *Broadcast-and-then-aggregates*.

to determining the existence of the functions $\circ$ and h. Unfortunately, it is easy to find counterexamples that demonstrate the non-existence of $\circ$ and h for arbitrary F in general case.

In the particular instance of flash attention, a combiner exists.

For the first *BroadcastAndThenAggregate* , C_1 is trivial that is equal to the original reduce function: $m = C_1(m_1, m_2) = \max(m_1, m_2)$.

For the second instance of *BroadcastAndThenAggregate* , consider the following sub-problem: $\bar{y}_1 = \exp(\bar{x} - m_1)$ has been computed for a single block, but a new constant factor m needs to be broadcast for the entire list. This requires re-computing $\bar{y}_1' = \exp(\bar{x} - m)$. The solution is to recover $\bar{x}$ from $\bar{y}_1$ and m_1 by computing:

$$\bar{x} = \log(\bar{y}_1) + m_1$$

then substitue $\bar{x}$ into the equation that computes $\bar{y}_1'$. That is:

$$\begin{aligned} \bar{y}_1' &= \exp\left(\log\left(\bar{y}_1\right) + m_1 - m\right) \\ &= \bar{y}_1 \exp(m_1 - m) \end{aligned}$$

Therefore:

1. the updating euqation for the broadcast step is: $\bar{y}_{\text{new}} = \bar{y}_{\text{old}} \exp(m_{\text{old}} - m_{\text{new}})$
2. combine partial results:

$$\begin{aligned} d =& C_2(d_1, d_2) = d_1 * \exp(\triangle m_1) + d_2 * \exp(\triangle m_2) \\ \triangle m_1 :=& m_1 - \max(m_1, m_2) \\ \triangle m_2 :=& m_2 - \max(m_1, m_2) \end{aligned}$$

For the third instance of *BroadcastAndThenAggregate* , consider the following sub-problem: $\bar{o}_1 = \frac{\bar{x}_1}{d_1} * \bar{y}$ has been computed for a single block, but a new constant factor d_2 needs to be broadcast for the entire list. This requires re-computing $\bar{o}_1' = \frac{\bar{x}_1'}{d} * \bar{y}$. The solution is to recover $\bar{x}_1 * \bar{y}$ from $\bar{o}_1$ and d_1 by computing:

$$\bar{x}_1' * \bar{y} = \exp(\triangle m_1) * \bar{o}_1 * d_1$$

Then the adjusted $o_1' = \frac{\exp(\triangle m_1) * o_1 * d_1}{d} = o_1 * \frac{d_1 \exp(\triangle m_1)}{d_1 * \exp(\triangle m_1) + d_2 * \exp(\triangle m_2)}$, and $o = C_3(o_1, o_2) = o_1 * \frac{d_1 * \exp(\triangle m_1)}{d_1 * \exp(\triangle m_1) + d_2 * \exp(\triangle m_2)} + o_2 * \frac{d_2 * \exp(\triangle m_2)}{d_1 * \exp(\triangle m_1) + d_2 * \exp(\triangle m_2)}$.

3.4 A Counter-example: Logsoftmax instead of softmax

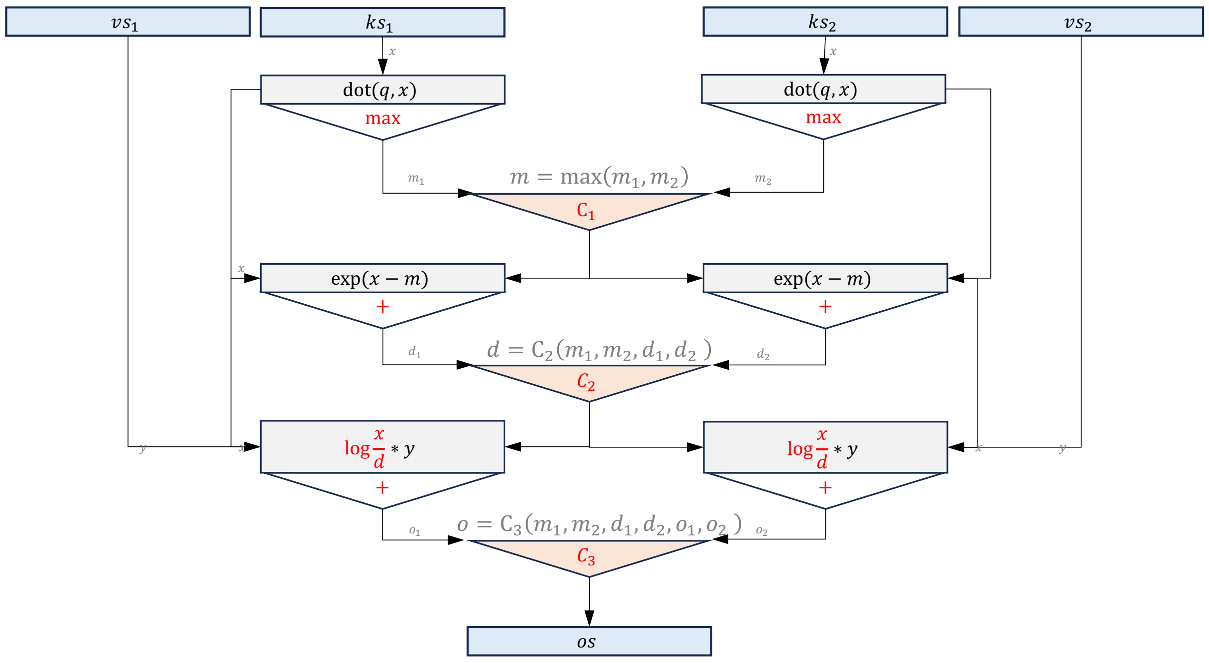


Figure 4: An imaginary example: use logsoftmax instead of softmax.

Let's think of an imaginary example, that compute:

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

C_1 and C_2 doest not change, thus: how C_3 looks like.

Suppose $\bar{z}_1$ and o_1 are results for the broadcast stage, and reduce stage computed on input blocks $\bar{x}_1$ and $\bar{y}$ respectively:

$$\bar{z}_1 = \log\left(\frac{\bar{x}_1}{d_1}\right) * \bar{y} = \bar{y} * \log(\bar{x}_1) - \bar{y} \log d_1$$

$$o_1 = \sum(0, \bar{z}_1)$$

Then it is required to compute an updated $\bar{z}_1'$ and o_1' since $\bar{x}_1$ and d_1 are updated.

We have:

$$\bar{z}'_1 = \log\left(\frac{\bar{x}'_1}{d}\right) * \bar{y} = \bar{y}\log\bar{x}'_1 - \bar{y}\log d$$

$$\begin{aligned}\bar{x}'_1 &= \exp(m_1 - m)\bar{x}_1 \\ d &= \exp(m_1 - m)d_1 + \exp(m_2 - m)d_2\end{aligned}$$

Thus we have:

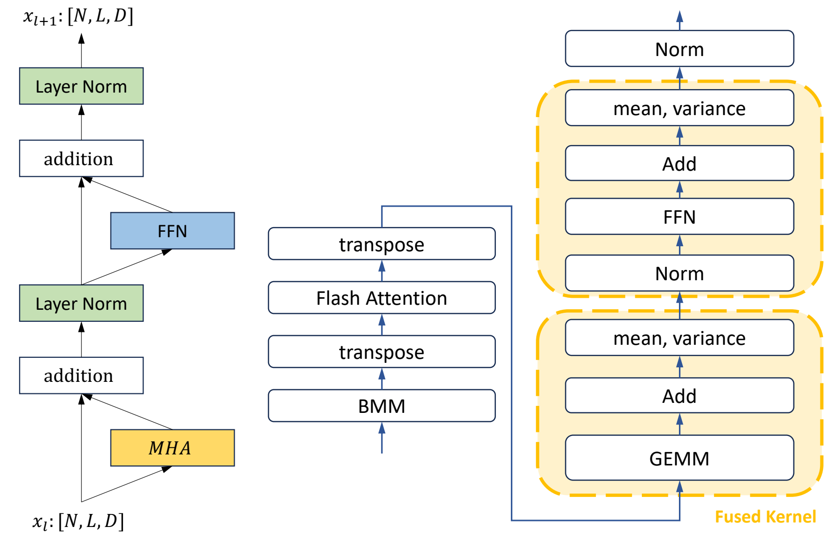
$$\begin{aligned}\bar{z}'_1 &= \bar{y}\log\left(\exp(m_1 - m)\bar{x}_1\right) - \bar{y}\log d \\ &= \bar{y}\left((m_1 - m) + \log(\bar{x}_1)\right) - \bar{y}\log d \\ &= \bar{y}(m_1 - m - \log d) + \bar{y}\log(\bar{x}) \\ &= \bar{y}(m_1 - m - \log d) + \bar{z}_1 + \bar{y}\log d_1 \\ &= \bar{y}(m_1 - m - \log d + \log d_1) + \bar{z}_1\end{aligned}$$

Given $\bar{z}'_1$:

$$\begin{aligned}o'_1 &= \text{sum}(0, \bar{z}'_1) \\ &= (m_1 - m - \log d + \log d_1)\text{sum}(0, \bar{y}) + o_1\end{aligned}$$

Update o_1 has to access its input $\bar{y}$ and compute an extra reduction sum on it.

4 The Transformer Block



(a) The Transformer block (b) Fused transformer block

Figure 5: Transformer block and a possible fusion plan.

The transformer model consists of a series of transformer blocks that are stacked on top of each other. A transformer block is as shown above.

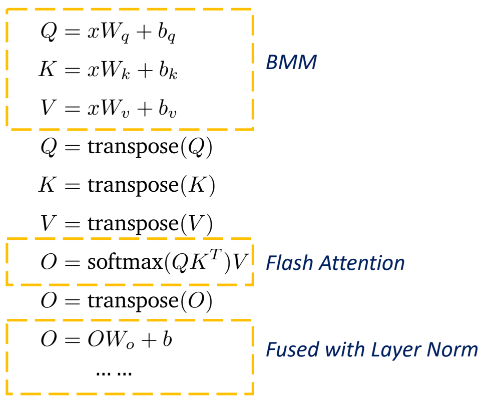

$Q = xW_q + b_q$
$K = xW_k + b_k$
$V = xW_v + b_v$
BMM
$Q = \text{transpose}(Q)$
$K = \text{transpose}(K)$
$V = \text{transpose}(V)$
$O = \text{softmax}(QK^T)V$
Flash Attention
$O = \text{transpose}(O)$
$O = OW_o + b$
… …
Fused with Layer Norm

Figure 6: The fusion plan for MHA.

4.1 MHA

4.2 Layer Normalization

$$y = \text{layernorm}(oW_o + b + x)$$

Input tensor x has a shape of $N \times L \times D$ where N stands for batch size, L stands for sequence length, and D stands for hidden size. In the original Transformer, $D = 512$.

$$y_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}} * \gamma_i + \beta_i \quad i = [0, \ldots D - 1]$$

Be definition:

$$\mu_N = \frac{1}{N}\sum_{i=1}^{N} x_i$$

$$\sigma_N^2 = \frac{1}{N-1}\sum_{i=1}^{N}(x_i - \mu_N)^2$$

To compute layer noramlization blockwise, we must use welford's online algorithm [2] to compute the variance σ^2. Define a quantity called $s_N = \sum_{i=1}^{N}(x_i - \mu_N)^2$:

$$s_N = s_{N-1} + (x_N - \mu_{N-1})(x_N - \mu_N)$$

4.3 FFN

W_1 has a shape of $D \times 4D$ and W_2 has a shape of $4D * D$.

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

4.4 Multi-scale Attention

$$v_1 = QK^T * \text{mask}$$
$$v_2 = |v_1|.\text{sum}(\text{dim} = -1).\text{clamp}(\text{min} = 1)$$
$$O = \frac{v_1}{v_2} V$$

Table 1: Input tensors and their shapes.

Input	**Shape**
Q	$[N, H, L, E]$
K	$[N, H, E, L]$
V	$[N, H, L, E]$
M(mask)	$[H, L, L]$

N stands for batch size, H stands for head number, L stands for sequence length and E stands for head dimensionality.

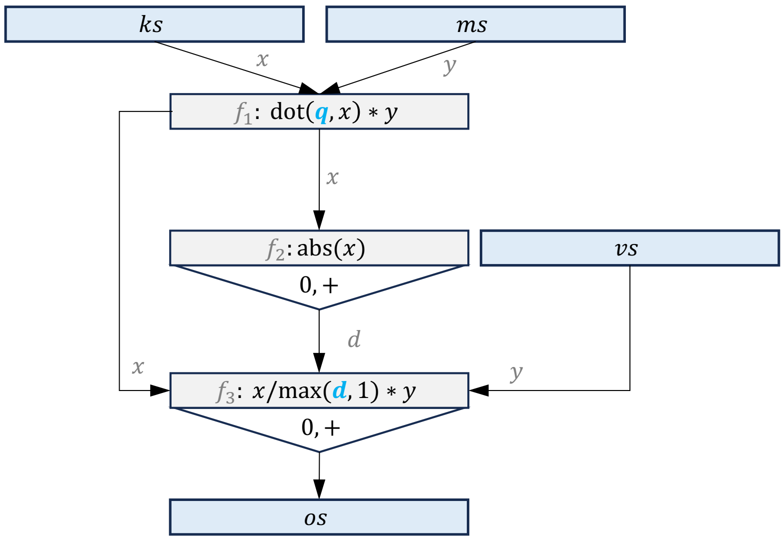


Figure 7: The multi-scale attention: the original computional flow.

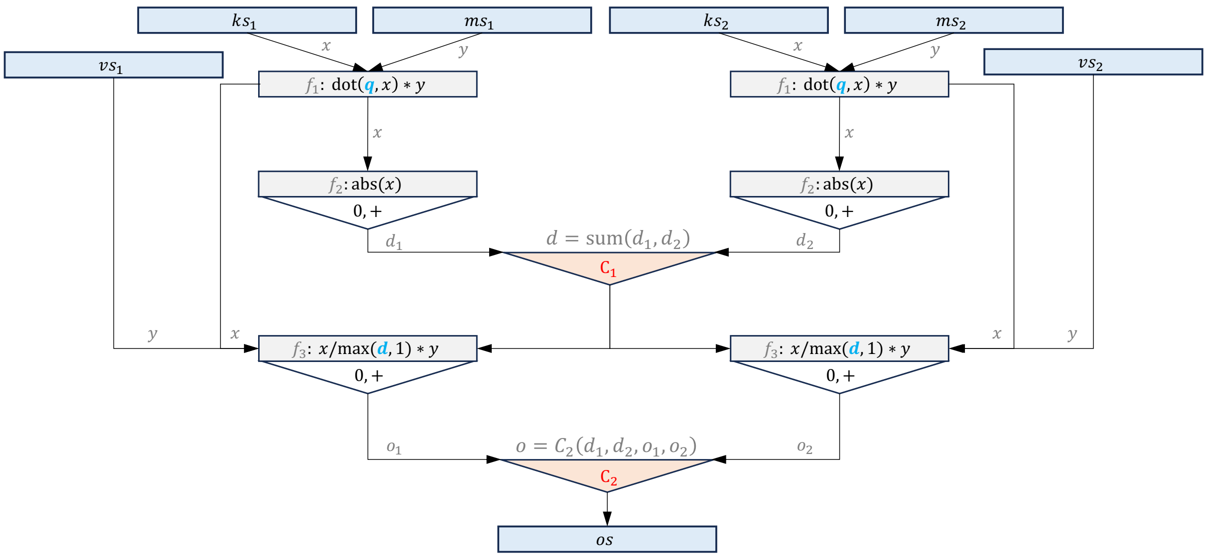
C_1 is trivial:

$$C_1(d_1, d_2) = d_1 + d_2$$

Consider $\bar{o}_1 = \frac{\bar{x}_1 * \bar{y}_1}{d_1}$ has been computed for a single block, but a new constant $d' = \max(d_1, 1)$ has to be broadcast to the entire list $\bar{x}_1$ to compute a new output $\bar{o}'_1 = \frac{\bar{x}_1 * \bar{y}_1}{d'_1}$.

$$\bar{x}_1 * \bar{y}_1 = d_1 * \bar{o}_1$$

$$\bar{o}'_1 = \frac{\bar{x}_1 * \bar{y}_1}{d'} = \frac{d_1 * \bar{o}_1}{d'}$$



ks_1
ms_1
ks_2
ms_2
vs_1
vs_2
f_1: dot(q, x) * y
f_2: abs(x)
0, +
$d = \mathrm{sum}(d_1, d_2)$
C_1
f_3: x/max(d, 1) * y
$o = C_2(d_1, d_2, o_1, o_2)$
C_2
os

Figure 8: The multi-scale attention: the expected computational flow.

Similarly: $\bar{o}'_2 = \frac{\bar{x}_2 * \bar{y}_2}{d'} = \frac{d_2 * \bar{o}_2}{d'}$, therefore:

$$
\begin{aligned}
o &= \bar{o}'_1 + \bar{o}'_2 \\
&= \frac{1}{d'}(d_1 * \bar{o}_1 + d_2\bar{o}_2)
\end{aligned}
$$

$$
\begin{aligned}
\bar{v} &= [\bar{v}_1 : \bar{v}_2] & & \\
r_1 &= \mathrm{sum}(\bar{v}_1) & r_2 &= \mathrm{sum}(\bar{v}_2) \\
d_1 &= \max(r_1, 1) & d_2 &= \max(r_2, 1)
\end{aligned}
$$

$\max(\mathrm{sum}(r_1, r_2), 1) \neq \max(\mathrm{sum}(\bar{v}), 1)$.

Inputs: $r_{i-1} : [B_r, 1]$,$d_{i-1} : [B_r, 1]$, $o_i : [B_r, d]$, $q_i : [B_r, d]$, $k_i : [d, B_c]$, $v_i : [d, B_c]$, $m_i : [B_r, B_c]$

Outputs: $r_i : [B_r, 1]$, $d_i : [B_r, 1]$, $o_i : [B_r, d]$

```
// o and d are accumulators
// q, k, v, m are input tiles
for $r_{i-1}, d_{i-1}, o_{i-1}, q_i, k_i, v_i, m_i \leftarrow i = [0, \frac{L}{B_c})$
  // compute partial results for the current tile
  $t_i = q_i @ k_i * m_i$
  $\bar{r}_i = \mathrm{sum}(\mathrm{abs}(t_i), \mathrm{dim=-1}))$
  $o_{prev} = t_i @ v_i$

  // update accumulators
  $\bar{o}_i = d_{i-1} * o_{i-1} + o_{prev}$
  $r_i = r_{i-1} + \bar{r}_i$
  $d_i = \max(r_i, \mathbf{1.})$
  $o_i = \bar{o}_i / d_i$
```

Listing 1: Fused multi-scale attention kernel.

Appendices

A Welford Algorithm for Calculating Variance

We want to calculate the mean and unbiased variance of a numeric list $X = [x_1 \dots x_N]$ that has a length N:

$$\mu_N = \frac{1}{N}\sum_{i=1}^{N} x_i$$

$$\sigma_N^2 = \frac{1}{N-1}\sum_{i-1}^{N} (x_i - \mu_N)^2$$

The two computations mentioned above scan the input list elements twice, each with a complexity of $O(N)$. Now, suppose we add a new element x_N to the list and want to update the mean μ_{N+1} and variance σ_{N+1}. Instead of repeating the two $O(N)$ computations, can we update μ_{N+1} and σ_{N+1}^2 using the existing values of N, μ_N, and σ_N^2? If so, we can compute μ_N and σ_N^2 for the entire list portion by portion for extremely long lists.

Let's start from the new mean μ_{N+1} from x_{N+1} and μ_N. This is rather straightforward:

$$\begin{aligned}\mu_{N+1} &= \frac{x_{N+1} + N\mu_N}{N+1} \\ &= \mu_N + \frac{x_{N+1} - \mu_N}{N+1} \qquad (1)\end{aligned}$$

Subsitute $N + 1$ with N in equation (1), we can also get:

$$\mu_N = \mu_{N-1} + \frac{x_N - \mu_{N-1}}{N} \qquad (2)$$

Let's define a quantity called s_N:

$$\begin{aligned}s_N &= \sum_{i=1}^{N}(x_i - \mu_N)^2 \\ &= \sum_{i=1}^{N-1} (x_i - \mu_N)^2 + (x_N - \mu_N)^2 \qquad (3)\end{aligned}$$

Substitue equation (2) into equation (3):

$$s_N = \sum_{i=1}^{N-1} \left(x_i - \mu_{N-1} - \frac{x_N - \mu_{N-1}}{N} \right)^2 + \left(x_N - \mu_{N-1} - \frac{x_N - \mu_{N-1}}{N} \right)^2$$

$$= \sum_{i=1}^{N-1} \left((x_i - \mu_{N-1}) - \frac{1}{N} (x_N - \mu_{N-1}) \right)^2 + \left(\frac{N-1}{N} (x_N - \mu_{N-1}) \right)^2 \tag{4}$$

Let's first look into the first component of equation (4):

$$\sum_{i=1}^{N-1} \left((x_i - \mu_{N-1}) - \frac{1}{N} (x_N - \mu_{N-1}) \right)^2$$

$$= \sum_{i=1}^{N-1} \left((x_i - \mu_{N-1})^2 - \frac{2}{N}(x_i - \mu_{N-1})(x_N - \mu_{N-1}) + (x_N - \mu_{N-1})^2 \right)$$

The equation in red is actually equal to zero.

$$\sum_{i=1}^{N-1} \frac{2}{N}(x_i - \mu_{N-1})(x_N - \mu_{N-1})$$

$$= \frac{2}{N}(N-1)(x_N - \mu_{N-1}) \left(\sum_{i=0}^{N-1} x_i - (N-1)\mu_{N-1} \right)$$

$$= \frac{2}{N}(N-1)(x_N - \mu_{N-1}) \left(\sum_{i=0}^{N-1} x_i - \sum_{i=0}^{N-1} x_i \right)$$

$$= 0$$

Substitue these results into equation (4):

$$s_N = \sum_{i=1}^{N-1} \left((x_i - \mu_{N-1})^2 + \frac{1}{N}(x_N - \mu_{N-1})^2 \right) + \left(\frac{N-1}{N}(x_N - \mu_{N-1}) \right)^2$$

$$= \frac{N-1}{N^2}(x_N - \mu_{N-1})^2 + \frac{(N-1)^2}{N^2}(x_N - \mu_{N-1})^2 + \sum_{i=1}^{N-1}(x_i - \mu_{N-1})^2$$

$$= \left[\frac{N-1}{N^2} + \frac{(N-1)^2}{N^2} \right] (x_N - \mu_{N-1})^2 + \sum_{i=1}^{N-1} (x_i - \mu_{N-1}^2)$$

$$s_N = s_{N-1} + \frac{N-1}{N}(x_N - \mu_{N-1})^2 \tag{5}$$

We can manipulate equation (5) further by given:

$$(N-1)\mu_{N-1} = N\mu_N - x_N$$

$$\mu_{N-1} = \frac{N\mu_N - x_N}{N-1}$$

substitute it into equation (5):

$$s_N = s_{N-1} + (x_N - \mu_{N-1})\frac{N-1}{N}(x_N - \frac{N\mu_N - x_N}{N-1})$$

$$s_N = s_{N-1} + (x_N - \mu_{N-1})(x_N - \mu_N)$$

Suppose we use Welford's algorithm to calculate the mean and variance for lists A and B. Now we want to obtain the mean and variance for AB, which is the concatenation of A and B. To do this, we can use the formula described in [1].

$$\begin{aligned} N_{AB} &= N_A + N_B \\ \delta &= \mu_B - \mu_A \\ \mu_{AB} &= \mu_A + \delta\frac{N_B}{N_{AB}} \\ M_{2,AB} &= M_{2,A} + M_{2,B} + \delta^2 * \frac{N_A N_B}{N_{AB}} \end{aligned}$$

B Gradient Computation

Let $\circ$ stands for elementwise multiplication.

B.1 Flash Attention

The standard attention:

$$\mathbf{S} = \mathbf{Q}\mathbf{K}^T \in \mathbb{R}^{N\times N}, \quad \mathbf{P} = \text{softmax}(\mathbf{S}) \in \mathbb{R}^{N\times N}, \quad \mathbf{O} = \mathbf{P}\mathbf{V} \in \mathbb{R}^{N\times d}$$

If we have the matrices $\mathbf{O}$, $\mathbf{Q}$, $\mathbf{K}$, $\mathbf{V}$, and $\mathbf{dO}$, the backward pass propagates the gradient $\mathbf{dO}$ from the output to the inputs. To compute the back propagation of $\mathbf{dQ}$, $\mathbf{dK}$, and $\mathbf{dV}$ by hand, we can follow the steps of the backpropagation algorithm. Let's first consider a scalar loss function ϕ and an output gradient $\mathbf{dO}$ with dimensions $\mathbb{R}^{N\times d}$.

To obtain the gradient of $\mathbf{V}$ in matrix notation, we can simply use the following formula: $\mathbf{dV} = \mathbf{P}^T\mathbf{dO}$. This means that the blockified version can be expressed as follows:

$$\mathbf{dV}_j = \sum_i \mathbf{P}_{ij}^T \mathbf{dO}_i = \sum_i \frac{\exp(\mathbf{K}_j{}^T\mathbf{Q}_i)}{\mathbf{L}_i}\mathbf{dO}_i \tag{4}$$

We can then move on to computing $\mathbf{dP}$ using the formula $\mathbf{dP} = \mathbf{dO}\mathbf{V}^T$. In blockified form, this can be expressed as follows:

$$\mathbf{dP}_{ij} = \mathbf{dO}_i\mathbf{V}_j^T$$

Since the Jacobian of $\bar{y} = \text{softmax}(\bar{x})$ is represented by the matrix $\mathbf{J}_{\bar{x}} = \text{diag}(\bar{y}) - \bar{y}^T\bar{y}$ where $\bar{x}$ and $\bar{y}$ are row vectors, we can use this to derive the following:

$$\mathbf{dS}_{i:} = (\text{diag}(\mathbf{P}_{i:}) - \mathbf{P}_{i:}^T\mathbf{P}_{i:})\mathbf{dP}_{i:} = \mathbf{P}_{i:} \circ \mathbf{dP}_{i:} - \mathbf{P}_{i:}(\mathbf{P}_{i:}^T\mathbf{dP}_{i:})$$

Define:

$$\mathbf{D}_i = \mathbf{P}_{i:}^T\mathbf{dP}_{i:} = \sum_j \frac{\exp(\mathbf{K}_j\mathbf{Q}_i^T)}{\mathbf{L}_i}\mathbf{dO}_i\mathbf{V}_j^T = \mathbf{dO}_i^T\sum_j \frac{\exp(\mathbf{Q}_i^T\mathbf{K}_j)}{\mathbf{L}_i}\mathbf{V}_j = \mathbf{dO}_i^T\mathbf{O}_i$$

Then:

$$\mathbf{dS}_{i:} = \mathbf{P}_{i:} \circ \mathbf{dP}_{i:} - \mathbf{P}_{i:}\mathbf{D}_i$$

Hence:

$$\mathbf{dS}_{ij} = \mathbf{P}_{ij} \circ \mathbf{dP}_{ij} - \mathbf{P}_{ij}\mathbf{D}_i = \mathbf{P}_{ij}(\mathbf{dP}_{ij} - \mathbf{D}_i)$$

We can obtain $\mathbf{dQ}$ and $\mathbf{dK}$ in a blockified style by utilizing $\mathbf{dS}_{ij}$ and the following relationships: $\mathbf{dQ} = \mathbf{dSK}$ and $\mathbf{dK} = \mathbf{dS}^T\mathbf{Q}$:

$$\begin{aligned}
\mathbf{dQ}_i &= \sum_j \mathbf{dS}_{ij}\mathbf{K}_j \\
&= \sum_j \mathbf{P}_{ij}(\mathbf{dP}_{ij} - \mathbf{D}_i)\mathbf{K}_j \\
&= \sum_j \frac{\exp(\mathbf{Q}_i^T\mathbf{K}_j)}{\mathbf{L}_i}(\mathbf{dO}_i\mathbf{V}_j^T - \mathbf{D}_i)\mathbf{K}_j \qquad (5) \\
\mathbf{dK}_j &= \sum_i \mathbf{dS}_{ij}^T\mathbf{Q}_i \\
&= \sum_i (\mathbf{dP}_{ij}^T - \mathbf{D}_i^T)\mathbf{P}_{ij}^T\mathbf{Q}_i \\
&= \sum_i (\mathbf{V}_j\mathbf{dO}_i^T - \mathbf{D}_i^T)\frac{\exp(\mathbf{K}_j\mathbf{Q}_i^T)}{\mathbf{L}_i}\mathbf{Q}_i \qquad (6)
\end{aligned}$$

Equations (4), (5), and (6) describe a blockified computational process for obtaining $\mathbf{dQ}$, $\mathbf{dK}$, and $\mathbf{dV}$.

B.2 Multi-scale Attention

The multi-scale attention computes as:

$$\mathbf{S} = \mathbf{QK}^T \circ \mathbf{M} \in \mathbb{R}^{N\times N}, \quad \mathbf{P} = \frac{\bar{1}}{\max(\text{rowsum}(\text{abs}(\mathbf{S})), \bar{1})} \in \mathbb{R}^{N\times 1}, \quad \mathbf{O} = \mathbf{SV} \circ \mathbf{P} \in \mathbb{R}^{N\times d}$$

To perform the backward computation, the system must compute the derivatives of $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$ with respect to the output $\mathbf{O}$, denoted as $\mathbf{dQ}$, $\mathbf{dK}$, and $\mathbf{dV}$, respectively. The computation of $\mathbf{dV}$ in matrix notation is straightforward:

$$\mathbf{dV} = \mathbf{S}^T\mathbf{dO} \circ \mathbf{P}$$

And the blockified version of $\mathbf{dV_j}$ is:

$$\mathbf{dV}_j = \sum_i \mathbf{S}_{ij}^T \mathbf{dO_i} \circ \mathbf{P}_i \tag{7}$$

By applying the chain rule, the next step in the backward computation is to compute the derivative matrix $\mathbf{dP}$:

$$\mathbf{dP} = \text{rowsum}(\mathbf{dO} \circ \mathbf{SV})$$

The blockified version of $\mathbf{dP}_{ij}$ is:

$$\mathbf{dP}_{ij} = \text{sum}(\mathbf{dO}_i \circ \mathbf{S}_{ij}\mathbf{V}_j)$$

Computing $\mathbf{dS}$ is a somewhat a bit more complicated, so it is best to break it down into smaller steps. Let's assume the following to simplify the process:

$$\begin{aligned} \mathbf{y}_1 &= \text{abs}(\mathbf{S}) \\ \mathbf{y}_2 &= \text{rowsum}(\mathbf{y}_1) \\ \mathbf{y}_3 &= \max(\mathbf{y}_2, \mathbf{1}) \end{aligned}$$

$$\mathbf{P} = \frac{\bar{\mathbf{1}}}{\mathbf{y}_3}$$

Then we can compute $\mathbf{dS}$ step by step:

$$\frac{\mathbf{dP}}{\mathbf{dy}_3} = -\frac{\bar{\mathbf{1}}}{\mathbf{y}_3 \circ \mathbf{y}_3} = -\mathbf{P} \circ \mathbf{P}$$

$$\frac{\mathbf{dy}_3}{\mathbf{dy}_2} \equiv \nabla_{\max} = \begin{cases} 1, & \mathbf{y}_2 > \bar{\mathbf{1}} \\ 0, & \mathbf{y}_2 \leq \bar{\mathbf{1}} \end{cases}$$

$$\frac{\mathbf{dy}_2}{\mathbf{dy}_1} = \bar{\mathbf{1}}_{N \times N}$$

$$\frac{\mathbf{dy}_1}{\mathbf{dS}} \equiv \nabla_{\text{abs}} = \begin{cases} 1, & \mathbf{S} \geq \bar{\mathbf{0}} \\ -1, & \mathbf{S} < \bar{\mathbf{0}} \end{cases}$$

And:

$$\begin{aligned} \frac{\mathbf{dP}}{\mathbf{dS}} &= \frac{\mathbf{dP}}{\mathbf{dy}_3} \circ \frac{\mathbf{dy}_3}{\mathbf{dy}_2} \circ \frac{\mathbf{dy}_2}{\mathbf{dy}_1} \circ \frac{\mathbf{dy}_1}{\mathbf{dS}} \\ &= -\mathbf{P} \circ \mathbf{P} \circ \nabla_{\max} \circ \nabla_{\text{abs}} \end{aligned}$$

The blockified computation for $\frac{\mathbf{d}\phi_{ij}}{\mathbf{dS_{ij}}} = \mathbf{dS}_{ij}$ is:

$$
\begin{aligned}
\mathbf{dS}_{ij} &= -\text{sum}(\mathbf{dO}_i \circ \mathbf{S}_{ij}\mathbf{V}_i) \circ \mathbf{P}_{ij} \circ \mathbf{P}_{ij} \circ \nabla^{ij}_{\max} \circ \nabla^{ij}_{\text{abs}} \\
&= -\frac{\text{sum}(\mathbf{dO}_i \circ \mathbf{O}_i)}{\mathbf{P}_{ij}} \circ \mathbf{P}_{ij} \circ \mathbf{P}_{ij} \circ \nabla^{ij}_{\max} \circ \nabla^{ij}_{\text{abs}} \\
&= -\text{sum}(\mathbf{dO}_i \circ \mathbf{O}_i) \circ \mathbf{P}_{ij} \circ \nabla^{ij}_{\max} \circ \nabla^{ij}_{\text{abs}}
\end{aligned}
$$

where:

$$
\nabla^{ij}_{\max} = \begin{cases} \bar{\mathbf{1}}, & \text{rowsum}(\text{abs}(\mathbf{S}_{ij})) \geq \bar{\mathbf{1}} \\ \bar{\mathbf{0}}, & \text{rowsum}(\text{abs}(\mathbf{S}_{ij})) < \bar{\mathbf{1}} \end{cases}
$$

$$
\nabla^{ij}_{\text{abs}} = \begin{cases} \bar{\mathbf{1}}, & \text{abs}(\mathbf{S}_{ij}) \geq \bar{\mathbf{0}} \\ -\bar{\mathbf{1}}, & \text{abs}(\mathbf{S}_{ij}) < \bar{\mathbf{0}} \end{cases}
$$

We can obtain $\mathbf{dQ}$ and $\mathbf{dK}$ in a blockified style by utilizing $\mathbf{dS}_{ij}$ and the following relationships: $\mathbf{dQ} = \mathbf{dSK} \circ \mathbf{M}$ and $\mathbf{dK} = \mathbf{dS}^T\mathbf{Q} \circ \mathbf{M}$:

$$
\begin{aligned}
\mathbf{dQ}_i &= \sum_j \mathbf{dS}_{ij}\mathbf{K}_j \circ \mathbf{M}_{ij} \\
&= -\sum_j \text{sum}(\mathbf{dO}_i \circ \mathbf{O}_i) \circ \mathbf{P}_{ij} \circ \nabla^{ij}_{\max} \circ \nabla^{ij}_{\text{abs}}\mathbf{K}_j \circ \mathbf{M}_{ij} \qquad (8) \\
\mathbf{dK}_j &= \sum_i \mathbf{dS}^T_{ij}\mathbf{Q}_i \circ \mathbf{M}_{ij} \\
&= -\sum_i \left(\text{sum}(\mathbf{dO}_i \circ \mathbf{O}_i) \circ \mathbf{P}_{ij} \circ \nabla^{ij}_{\max} \circ \nabla^{ij}_{\text{abs}}\right)^T \mathbf{Q}_i \circ \mathbf{M}_{ij} \qquad (9)
\end{aligned}
$$

Equations (7), (8), and (9) describe a blockified computational process for obtaining $\mathbf{dQ}$, $\mathbf{dK}$, and $\mathbf{dV}$.

References

[1] Tony F Chan, Gene H Golub, and Randall J LeVeque. Updating formulae and a pairwise algorithm for computing sample variances. In *COMPSTAT 1982 5th Symposium held at Toulouse 1982: Part I: Proceedings in Computational Statistics*, pages 30–41. Springer, 1982.

[2] BP Welford. Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4(3):419–420, 1962.