GeMMMapReduce
or
Commutative Monoids and Memory Efficient Layers in Neural Networks.

Amaru Cuba Gyllensten

May 2025

Abstract

This paper aims at generalizing recent memory and cache-efficient implementations of common layers in Neural Networks, most notably FlashAttention Dao et al. (2022). The main observation is that these layers follow a common pattern, the same which has been used in MapReduce in the Big Data setting, and Monoidal folds in the functional programming setting. We present monoids which enable streaming and efficient implementations of Attention, CrossEntropy, and Two-Layer MLPs, and identify a property of (some) Commutative Monoids which enable these efficient folds.

The purpose of this paper is to illuminate the connection between MapReduce or Monoidal Folds to efficient Layers in Neural Networks, and propose a primitive for folding over the result of (mapped) matrix multiplications, which we call GeMMMapReduce (General Matrix Multiply Map Reduce).

1 Introduction

1.1 Notation

This paper employs liberal use of type and lambda notation borrowed from functional programming. A short primer of this notation is given below.

Types & Terms $a : A$ denotes that the term a has type A. For example, $a : \mathbb{R}$ denotes that the term a is a real number.

Functions $A \to B$ denotes the type of functions from A to B. Lambda functions are written out as $f = x \mapsto x^2 + 3$, meaning $f(x) = x^2 + 3$.

$\to$ is interpreted in a right-associative manner, meaning $A \to B \to C = A \to (B \to C)$, which is isomorphic to $A \times B \to C$.

Product Types $A \times B$ denotes the product type of A and B. For brevity, terms of product types are occasionally written out as tuples. If $a : A$ and $b : B$, we can write $(a, b) : A \times B$, and vice versa. For a term $x : A \times B$, x_0 is the term corresponding to the left-hand type (A), and x_1 corresponds to the right hand-type. We also used named product types, where $\{a : A, b : B\}$ is the type of pairs of A and B indexed by the names a and b, i.e. for $x : \{a : A, b : B\}$ x_a is the A-part of the product, and x_b is the B-part of the product. If y and z are terms of type A and B, the term $x = \{a : y, b : z\}$, i.e. x such that $x_a = y$ and $x_b = z$.

Exponent Types A type A^N where N is an integer denotes the type of N-tuples of A. In other words, $\mathbb{R}^2$ denotes the type of pairs of reals, or points in a plane. More generally, a term $v : A^B$ can be thought of as an array of As indexed by B (or equivalently, $B \to A$), or — when appropriate — a tensor (in the machine learning sense).

1.2 What are commutative monoids?

A monoid S is a tuple $(T, \odot, \mathbf{id})$, where T is the type, and $\odot : T \to T \to T$ is a binary operation. The binary operation must be associative, i.e. $a \odot (b \odot c) = (a \odot b) \odot c$, and $\mathbf{id}$ must be an identity element, i.e. $\mathbf{id} \odot a = a = a \odot \mathbf{id}$. Commutative monoids are monoids where the binary operation is commutative as well as associative, i.e. $a \odot b = b \odot a$.

There are many instances of commutative monoids in modern deep learning:

Sum Summation over real numbers:

$$\begin{aligned} T &= \mathbb{R} \\ x \odot y &= x + y \\ \mathbf{id} &= 0 \end{aligned}$$

WSum Weighted sums of vectors.

$$\begin{aligned} T &= \{v : \mathbb{R}^D, w : \mathbb{R}_{\geq 0}\} \\ x \odot y &= z \\ \text{where } z_w &= x_w + y_w \\ z_v &= \begin{cases} x_v \frac{x_w}{z_w} + y_v \frac{y_w}{z_w}, & z_w > 0 \\ 0, & \text{otherwise} \end{cases} \\ \mathbf{id} &= \{v : 0, w : 0\} \end{aligned}$$

LogWSum Weighted sums of vectors with weights in logspace[1]:

$$
\begin{aligned}
T &= \{v : \mathbb{R}^D, w : \mathbb{R}\} \\
x \odot y &= z \\
\text{where } z_w &= \ln(\exp(x_w) + \exp(y_w)) \\
z_v &= \begin{cases} x_v \exp(x_w - z_w) + y_v \exp(y_w - z_w), & z_w \neq -\infty \\ 0, & \text{otherwise} \end{cases} \\
\mathbf{id} &= \{v : 0, w : -\infty\}
\end{aligned}
$$

Given a commutative monoid S, we can derive a function $fold$ that takes as input a sequence of Ts and returns the product over all elements in the sequence: $fold : [T] \to T = X \mapsto \bigodot_i X_i$.

2 Properties of Commutative Monoids in the context of Machine Learning

There are a couple of benefits of using folds over commutative monoids in the context of machine learning:

- There is no need to materialize all values in memory during the computation of a fold, enabling better usage of GPU-memory.
- Partial results can be computed and combined in arbitrary order, reducing the need for synchronization and enabling a myriad of parallelization schemes.
- *If* the derivative $\frac{d\ x \odot y}{d\ x}$ can be expressed as a function of $x \odot y$ and x, then the same holds for $\frac{d\ fold(X)}{d\ X_i}$, allowing local computation of partial gradients and reducing the need for communication, computation, and synchronization during gradient calculation.

Theorem 2.1. *Given a commutative monoid $(T, \odot, \mathbf{id})$ whose derivative $\frac{d\ x \odot y}{d\ x}$ is a function (D) of $x \odot y$ and x, i.e:*

$$\frac{d\ x \odot y}{d\ x} = D(x \odot y, x) \tag{1}$$

We have that for any sequence $X : [T]$, the following holds:

$$\frac{d\ fold(X)}{d\ X_i} = D(fold(X), X_i)$$

[1] This monoid applies to attention: $\text{softmax}(a)V = x_0 \odot x_1 \odot \cdots$, where $x_i = \{v = V_i, w = a_i\}$

Proof. let $P = fold(X)$, for any partition of X into two sets X_L, X_R, with corresponding partial products $P_L = fold(X_L)$ and $P_R = fold(X_R)$.

$$
\begin{aligned}
P &= P_L \odot P_R = P_R \odot P_L \qquad \text{(By commutativity and associativity)} \\
\frac{d\ P}{d\ P_L} &= \frac{d\ P_L \odot P_R}{d\ P_L} \\
&= D(P_L \odot P_R, P_L) \\
&= D(P, P_L)
\end{aligned}
$$

This extends trivially to $P_L = X_i$, i.e. $X_L = [X_i]$.

□

Corollary 2.1.1. *let $P = fold(X)$. If we satisfy eq. 1, then for any partition of X into sets $X^1, X^2, \ldots, X^n$, with corresponding partial products $P_i = fold(X^i)$, we have that*

$$\frac{d\ P}{d\ P_i} = D(P, P_i) \tag{2}$$

this extends trivially to $P_i = X_i$, i.e. $X^i = [X_i]$.

Corollary 2.1.2. *Consider the scenario where $X_i = f(a, B_i)$ and let $P = fold(X)$. If we satisfy eq. 1, then for any partition of X into sets $X^1, X^2, \ldots X^n$ with corresponding partial products $P_i = fold(X^i)$, we have that:*

$$
\begin{aligned}
\frac{d\ P}{d\ a} &= \sum_i \frac{d\ P_i}{d\ a} \cdot D(P, P_i) \\
\frac{d\ P}{d\ X^i} &= \frac{d\ P_i}{d\ X^i} \cdot D(P, P_i)
\end{aligned}
$$

this extends trivially to $P_i = X_i$, i.e. $X^i = [X_i]$.

Corollary 2.1.3. *Consider the scenario where $X_{ij} = f(A_i, B_j)$ and let $P_i = fold(X_i) = \bigodot_j X_{ij}$ be a vector of products. If we satisfy eq. 1, then we have that:*

$$
\begin{aligned}
\frac{d\ P}{d\ A_i} &= \sum_j \frac{d\ X_{ij}}{d\ A_i} \cdot D(P_i, X_{ij}) \cdot \frac{d\ P}{d\ P_i} \\
\frac{d\ P}{d\ B_j} &= \sum_i \frac{d\ X_{ij}}{d\ B_j} \cdot D(P_i, X_{ij}) \cdot \frac{d\ P}{d\ P_i}
\end{aligned}
$$

And more generally that for any partition of A into $A^1, A^2, \ldots, A^m$ and B into $B^1, B^2, \ldots, B^n$, with corresponding partial product vectors P^{ij}, where P^i

denotes the vector of products $\bigodot_j P^{ij}$*:*

$$\frac{d\ P}{d\ A^i} = \sum_j \frac{d\ P^{ij}}{d\ A^i} \cdot D(P^i, P^{ij}) \cdot \frac{d\ P}{d\ P^i}$$
$$\frac{d\ P}{d\ B^j} = \sum_i \frac{d\ P^{ij}}{d\ B^j} \cdot D(P^i, P^{ij}) \cdot \frac{d\ P}{d\ P^i}$$

Corollary 2.1.2 provides a general method for computing the gradient for a during a fold over B. It is, however, limited by how efficiently the sum can be represented. In the case where T is essentially a scalar[2] this is trivial, but for more complex monoids the cost of keeping this gradient in memory might be prohibitively expensive.

2.1 GeMMMapReduce

The benefit of not materializing all partial values is most apparent in the case where the fold is applied to the results of matrix multiplications, one such instance is attention:

$$\begin{aligned}
Q &: \mathbb{R}^{M \times F} \\
K &: \mathbb{R}^{N \times F} \\
V &: \mathbb{R}^{N \times D} \\
\text{attention}(Q, K, V) &= \text{softmax}(QK^\intercal)V \\
\text{attention}(Q, K, V)_i &= Y_{iv} \\
&\text{where} \\
&Y_i = \bigodot_j H_{ij} \\
&H_{ij} = \{z = Q_i K_j^\intercal,\ v = V_j\} \\
&a \odot b = \begin{Bmatrix} z = & \ln(e^{a_z} + e^{b_z}) \\ v = & a_v e^{a_z - z} + b_v e^{b_z - z} \end{Bmatrix}
\end{aligned}$$

Here, we can think of H as an M by N matrix of T-values, with each such value consisting of an attention weight and value. Combining two such values a and b results in a new total weight, corresponding to the sum of their respective non log-space weights, and a new value corresponding to the weighted average of the two values. Since the result is a fold over the N-dimension, the full matrix does not have to be materialized. We can compute appropriate chunks of the matrix, fold the chunk, and add to a running total, only realizing the M normalizing factors, and the $M \times D$-sized weighted sum. Flash attention Dao et al. (2022) can be seen as an instance of this general approach.

[2]For example, where $a \odot b =$ logaddexp(a, b) i.e. $fold(X) =$ logsumexp(X)

We also have that condition 1 holds:

$$
\begin{aligned}
c &= a \odot b \\
\frac{d\, c}{d\, a} &= g \mapsto \begin{pmatrix} z: & (g_z + \langle g_v, a_v - c_v \rangle) \exp(a_z - c_z) \\ v: & g_v \exp(a_z - c_z) \end{pmatrix} \\
&= D(c, a)
\end{aligned}
$$

Or in matrix form:

$$
D(c, a) = \exp(a_z - c_z) \begin{bmatrix} 1 & (a_v - c_v) \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \tag{3}
$$

Which due to theorem 2.1 and 2.1.3 implies that the gradient w.r.t. Q, K and V can be calculated based on the final product Y and local (recomputed) H-values.

Listing 1: Attention example using elementwise MMMapReduce.

```
def forward(Q, K, V):
    """
    Q: M x F matrix
    K: N x F matrix
    V: N x D matrix
    returns:
    A: M-vector of T-values
    """
    A = full(id) # M-vector of identity T-values

    for i in M:
        q = Q[i]  # F-vector (query)
        a = id    # identity T-value
        for j in N:
            k = K[j]
            v = V[j]
            h = {z: q @ k, v: v}
            a = a o h
        A[i] = a
        gQ[i] = gq
    return A

def backward(A, Q, K, V, gA):
    """
    A: M-vector of T-values
    gA: output (A) gradient
    returns: gQ, gK, gV
    """
    gQ, gK, gV = [zeros_like(p) for p in [Q, K, V]]
    for i in M:
        gq = zeros_like(Q[i])
        a = A[i]
        for j in N:
            k = K[j]
            v = V[j]
            # local h and h-gradient
```

```
            h = {z: q @ k, v: v}
            gh = D(a, h)
            gq += gh.z * k
            gK[j] += gh.z * q
            gV[j] += gh.v
        gQ[i] = gq
    return gQ, gK, gV
```

In table 1 we give further examples of this approach applied to cross entropy against class indices, cross entropy between two distributions with logits given by matrix multiplication, and two-layer MLPs. For proofs of condition 1 for these, see Appendix A.

Input	Map	Reduce
$Q : \mathbb{R}^{M \times F}$ $K : \mathbb{R}^{N \times F}$ $V : \mathbb{R}^{N \times D}$ $\downarrow$ $Y : \mathbb{R}^{M \times D}$	$T : \{z : \mathbb{R}, v : \mathbb{R}^D\}$ $H : T^{M \times N}$ $H_{ij} = \begin{Bmatrix} z : & Q_i K_j^\intercal \\ v : & V_j \end{Bmatrix}$	$A : T^M$ $A_i = \bigodot_j H_{ij}$ $a \odot b = \begin{Bmatrix} z : & \ln(e^{a_z} + e^{b_z}) \\ v : & a_v e^{a_z - z} + b_v e^{b_z - z} \end{Bmatrix}$ $Y_i = A_{iv}$
Attention: $Y = \text{softmax}(QK^\intercal)V$		
$P : \mathbb{R}^{M \times D}$ $C : \mathbb{R}^{N \times D}$ $T : N^M$ $\downarrow$ $Y : \mathbb{R}^M$	$T : \{p : \mathbb{R}, n : \mathbb{R}\}$ $H : T^{M \times N}$ $H_{ij} = \begin{Bmatrix} p : & P_i C_j^\intercal \\ n : & T_i = j \ ? \ p \ : \ 0 \end{Bmatrix}$	$A : T^M$ $A_i = \bigodot_j H_{ij}$ $a \odot b = \begin{Bmatrix} p : & \ln(e^{a_p} + e^{b_p}) \\ n : & a_n + b_n \end{Bmatrix}$ $Y_i = A_{ip} - A_{in}$
Cross Entropy: $Y = \text{cross-entropy}(\text{logits} = PC^\intercal, \text{targets} = T)$		
$P^s : \mathbb{R}^{M \times D}$ $C^s : \mathbb{R}^{N \times D}$ $P^t : \mathbb{R}^{M \times E}$ $C^t : \mathbb{R}^{N \times E}$ $\downarrow$ $Y : \mathbb{R}^M$	$T : \{p : \mathbb{R}, q : \mathbb{R}, n : \mathbb{R}\}$ $H : T^{M \times N}$ $H_{ij} = \begin{Bmatrix} q : & P_i^s C_j^{s\intercal} \\ p : & P_i^t C_j^{t\intercal} \\ n : & q \end{Bmatrix}$	$A : T^M$ $A_i = \bigodot_j H_{ij}$ $a \odot b = \begin{Bmatrix} q : & \ln(e^{a_q} + e^{b_q}) \\ p : & \ln(e^{a_p} + e^{b_p}) \\ n : & a_n e^{a_p - p} + b_n e^{b_p - p} \end{Bmatrix}$ $Y_i = A_{iq} - A_{in}$
Cross Entropy: $Y = \text{cross-entropy}(\text{logits} = P^s C^{s\intercal}, \text{targets} = \text{softmax}(P^t C^{t\intercal}))$		
$X : \mathbb{R}^{B \times M}$ $P : \mathbb{R}^{M \times K}$ $Q : \mathbb{R}^{K \times N}$ $\downarrow$ $Y : \mathbb{R}^{B \times N}$	$T : \{v : \mathbb{R}^N\}$ $H : T^{B \times K}$ $H_{ij} = \{v : \sigma(X_i P_{:j}) Q_j\}$	$A : T^B$ $A_i = \bigodot_j X_{ij}$ $a \odot b = a_v + b_v$ $Y_i = A_{iv}$
MLP: $Y = \sigma(XP)Q$		

Table 1: Examples of Monoids T, map functions from inputs to intermediate H-values, reduction operations, and out projections that realizes different operations used in Artificial Neural Networks.

2.2 Batching and Compute efficiency

The pseudocode in listing 1 works elementwise, this is not really appropriate for efficient computation. A more GPU-friendly approach would be to compute the intermediate values and the gradient over appropriately sized slices of the H-matrix, while simultaneously folding it along the fold dimension, as is done in listing 2, where we show a general proof of concept implementation of the GeMMMapReduce operation, given user-provided functions `proj_fold`, `proj_fold_bwd`, `binary_reduce`, and helper functions `chunker`, `init`.

Listing 2: Generalized, Batched, GeMMMapReduce example, implemented as torch function.

```
class GeMMMapReduce(torch.autograd.Function):
    @staticmethod
    def forward(*X):
        A = init(X)
        for aslice, xslice in chunker(X):
            a = aslice(A)
            x = xslice(X)
            local_a = proj_fold(x)
            new_a = binary_reduce(a, local_a)
            for view, new_val in zip(a, new_a):
                view.copy_(new_val)
        return A

    @staticmethod
    def setup_context(ctx, inputs, outputs):
        ctx.num_inputs = len(inputs)
        ctx.save_for_backward(*inputs, *outputs)

    @staticmethod
    @once_differentiable
    def backward(ctx, *gA):
        X = ctx.saved_tensors[:ctx.num_inputs]
        A = ctx.saved_tensors[ctx.num_inputs:]
        gX = [p.new_zeros(p.shape) for p in X]
        for aslice, xslice in chunker(X):
            # Extract chunks
            x = xslice(X)
            a = aslice(A)
            ga = aslice(gA)
            # recompute and calculate local gradients.
            gx = proj_fold_bwd(x, a, ga)
            # Add local gradients to global.
            for g, d in zip(xslice(gX), gx):
                g.add_(d)
        return tuple(gX)
```

3 Discussion

In general, the benefit of activation recomputation is to reduce memory at the cost of the extra compute needed for recomputation. This obviously holds for

GeMMMapReduce as well. If the amount of memory saved is inconsequential, or is dwarfed by the cost of recomputation, there is little benefit of the approach. One notable example of this is the MLP-example: $Y = \sigma(XP)Q$, $X : \mathbb{R}^{B\times M}$, $P : \mathbb{R}^{M\times K}$, $Q : \mathbb{R}^{K\times N}$, if $K \gg M$, the approach is viable, whereas if $K \approx M$, there is little to no benefit.

The hope is that with a general and *simple* framework that enables memory and cache-efficient layers, we might be able to iterate and innovate faster by making memory efficient layers by design.

One such avenue would be to investigate FeedForward networks using the monoidal MLP: Let X, P, Q be matrices of shape $B \times D$, $D \times K$, and $K \times D$, respectively, where B and K are very large, but D is small. Naive implementation would result in space complexity $BD + 2KD + BK$, with BK being the dominant factor. GeMMMapReduce can remove the BK-term, at a cost of extra FLOPs during recomputation: Naive time: $12BKD$. GeMMMapReduce time: $14BKD$. i.e. $\approx 17\%$ increase in FLOPs. If we let $K = B = 16384$, and $D = 128$, this would result in a memory requirement of $\approx 2\%$ of the naive implementation.

A Gradients for CrossEntropy and MLP

For the examples in table 1 we also have that condition 1 holds, and by extension that theorem 2.1 holds:

References

Dao, T., Fu, D. Y., Ermon, S., Rudra, A., and Re, C. (2022). Flashattention: Fast and memory-efficient exact attention with IO-awareness. In Oh, A. H., Agarwal, A., Belgrave, D., and Cho, K., editors, *Advances in Neural Information Processing Systems*.

Monoid	Gradient
$T : \{p : \mathbb{R}, n : \mathbb{R}\}$ $a \odot b = \begin{Bmatrix} p : & \ln(e^{a_p} + e^{b_p}) \\ n : & a_n + b_n \end{Bmatrix}$ $= c$	$\frac{d\ c}{d\ a} = g \mapsto \begin{Bmatrix} p : & g_p e^{a_p - c_p} \\ n : & g_n \end{Bmatrix}$
CrossEntropy	
$T : \{q : \mathbb{R}, p : \mathbb{R}, n : \mathbb{R}\}$ $a \odot b = \begin{Bmatrix} q : & \ln(e^{a_q} + e^{b_q}) \\ p : & \ln(e^{a_p} + e^{b_p}) \\ n : & a_n e^{a_p - p} + b_n e^{b_p - p} \end{Bmatrix}$ $= c$	$\frac{d\ c}{d\ a} = g \mapsto \begin{Bmatrix} q : & g_q e^{a_q - c_q} \\ p : & (g_p + g_n(a_n - c_n))e^{a_p - c_p} \\ n : & g_n e^{a_p - c_p} \end{Bmatrix}$
CrossEntropy between two distributions	
$T : \{v : \mathbb{R}^D\}$ $a \odot b = \{v : a_v + b_v\}$ $= c$	$\frac{d\ c}{d\ a} = g \mapsto g$
MLP	