AN ALGEBRAIC INTERFACE FOR BACKPROPAGATION THROUGH FUSED MAP-FOLDS

Anonymous authors
Paper under double-blind review

ABSTRACT

Memory-efficient neural kernels often have the structure of a fused map-fold: a large logical interaction space is mapped to monoidal factors and immediately folded, avoiding materialization of tensors such as attention logits. While this explains the forward pass of kernels such as FlashAttention and fused cross entropy, their backward passes are usually derived and implemented separately. We identify a sufficient algebraic condition, the Local Gradient Property, under which the reverse pass of a map-fold can be computed with the same fused structure: local factors are recomputed tile by tile, and their cotangents are recovered from compact saved fold state. We extend this condition with a finalize-embed factorization that exposes a compact algebraic backward interface, and cover both commutative folds and ordered non-commutative folds with prefix state. We present CuTileReduce, an implementation of the interface built on CuTile, and instantiate it for cross entropy, attention, and a non-commutative attention variant, demonstrating correct memory-efficient forward and backward kernels generated from the same map-fold abstraction.

1 NOTATION

Types and Products: We write $a \in A$ to denote that the term a is of type A. Named product types are written as $\{a \in A,\, b \in B\}$, with corresponding instantiated terms $v \in A, w \in B$, written as $\{a : v,\, b : w\}$.

Functions and Application: Function arrows associate to the right: $A \to B \to C \equiv A \to (B \to C)$. Standard parentheses $f(x)$ denote function application on primal parameters or state. Conversely, square brackets $f[v]$ are reserved for operations acting on dual objects (gradients, cotangents, or accumulations). A function acting on a dual object v parameterized by primal state x is written as $f(x)[v]$.

Composition and Multiplication: Symbol $\circ$ denotes function composition: $(f \circ g)(x) = f(g(x))$. Matrix multiplication is denoted using standard juxtaposition (e.g., AB). $a \cdot \mathbf{x}$ denotes scalar or elementwise multiplication. When applicable, this also applies to records: $\alpha \cdot \{a : v,\, b : w\} = \{a : \alpha \cdot v,\, b : \alpha \cdot w\}$

Derivatives, Pullbacks and Cotangents: For $a \in A, b \in B$, $\frac{\mathrm{d}\, b}{\mathrm{d}\, a} : B^* \to A^*$ denotes the pullback linear map (the transpose Jacobian) that transforms cotangents $\bar{b} \in B^*$ into cotangents $\bar{a} \in A^*$. Accordingly: $\frac{\mathrm{d}\, c}{\mathrm{d}\, a} = \frac{\mathrm{d}\, b}{\mathrm{d}\, a} \circ \frac{\mathrm{d}\, c}{\mathrm{d}\, b}$. Unless stated otherwise, monoids (S, T) and inputs (X), requiring gradients are assumed to be finite dimensional vector spaces or manifolds.

2 INTRODUCTION

A broad class of functions $\mathbf{y} = f(\mathbf{x})$ can be decomposed as:

$$f = \mathrm{fold} \circ \mathrm{map}_\rho \tag{1}$$

$$\mathrm{map}_\rho(\mathbf{x}) = [\mathbf{l}_1, \mathbf{l}_2, \ldots, \mathbf{l}_n], \quad \mathbf{l}_i = \rho(\mathbf{x}_i) \tag{2}$$

$$\mathrm{fold}(\mathbf{l}) = \mathbf{l}_1 \odot \mathbf{l}_2 \odot \cdots \odot \mathbf{l}_n \tag{3}$$

where $\rho : X \to S$ maps an interaction in X to a local factor in S, map_ρ applies that mapping to each i-indexed interaction in $\mathbf{x}$, and the subsequent fold combines all resulting local factors using the associative binary product $\odot : S \times S \to S$. A common approach to reduce space complexity is map-fold fusion: Instead of realizing all intermediate local factors $\mathbf{l}$, factors are combined in place, meaning that the fold can be computed in constant, rather than linear space, relative to the number of interactions.

In the context of machine learning we are not only interested in computing the forward pass $\mathbf{y} \leftarrow f(\mathbf{x})$, but also the backward pass $\bar{\mathbf{x}} \leftarrow \frac{\mathrm{d}\, \mathbf{y}}{\mathrm{d}\, \mathbf{x}}\, [\bar{\mathbf{y}}]$. While forward pass map-fold fusion is quite simple, the backward pass is usually derived

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

manually on a case by case basis. This paper introduces an algebraic approach that enables fusion in the backward pass, using what we call the Local Gradient Property; a sufficient condition for when backward passes of map-fold functions over monoids can be made memory efficient.

Contributions.

- A sufficient algebraic criterion, the Local Gradient Property, for reverse-mode recomputation of fused map-fold kernels without storing the logical factor tensor.
- A compact finalize-embed backward interface that exposes only the state needed for tile-local gradient reconstruction, including an extension to ordered non-commutative folds via prefix state.
- Execution-carrier and transport constructions. Execution-carriers connect the semantic algebra used for differentiation to numerically stable kernel state, while the transport construction help derive new valid instances and provide proof of generality.
- CuTileReduce, a CuTile[1] implementation with analytical scheduling and empirical tuning, evaluated on cross entropy, attention, and a non-commutative attention variant.

3 The Local Gradient Property

3.1 Commutative Folds

Definition 1 (Commutative Local Gradient Property)**.** *Let $(S, \odot)$ be a commutative monoid, S a finite-dimensional smooth manifold, and let $\odot$ be differentiable. We say that $(S, \odot)$ satisfies the commutative Local Gradient Property if there exists a function*

$$D : S \times S \to S^* \to S^*$$

such that, for all $a, b \in S$,

$$\frac{\mathrm{d}\, a \odot b}{\mathrm{d}\, a} = D(a \odot b, a).$$

Proposition 1 (Local gradients for commutative folds)**.** *Let $l_i \in S$, and $g = l_1 \odot \cdots \odot l_n$. If $(S, \odot)$ satisfies the commutative Local Gradient Property (definition 1), then*

$$\frac{\mathrm{d}\, g}{\mathrm{d}\, l_i} = D(g, l_i).$$

Proof. By the monoid laws and commutativity, there exists b such that $g = l_i \odot b$. From there the result follows directly from definition 1. □

Example: LogWeightedSum As an illustrative example, we introduce the LogWeightedSum monoid, which forms the basis for attention, cross-entropy, and other information theoretic measures:

$$\begin{aligned} LWS &\coloneqq \left\{z \in \mathbb{R},\, \mu \in \mathbb{R}^D\right\} \\ a \odot_{LWS} b &= \{z : \mathrm{logaddexp}(a_z,\, b_z),\, \mu : a_\mu \cdot \exp(a_z - z) + b_\mu \cdot \exp(b_z - z)\} \end{aligned}$$

The LWS monoid is commutative, with identity[2] $\{z : -\infty,\, \mu : 0\}$ and satisfies the commutative LGP:

$$\begin{aligned} \frac{\mathrm{d}\, a \odot b}{\mathrm{d}\, a}[\Delta] &= \exp(a_z - (a \odot b)_z) \cdot \{z : \Delta_z + \langle \Delta_\mu,\, a_\mu - (a \odot b)_\mu \rangle,\, \mu : \Delta_\mu\} \\ &\Downarrow \\ D(g, l)[\Delta] &= \exp(l_z - g_z) \cdot \{z : \Delta_z + \langle \Delta_\mu,\, l_\mu - g_\mu \rangle,\, \mu : \Delta_\mu\} \end{aligned}$$

[1] https://github.com/nvidia/cutile-python

[2] With some minor tweaking of the $\odot$ function.

3.2 General Folds

Definition 2 (General Local Gradient Property)**.** *Let $(S, \odot)$ be a monoid and let $\odot$ be differentiable. We say that $(S, \odot)$ satisfies the general Local Gradient Property if there exists a function*

$$D : S \times S \times S \to S^* \to S^*$$

such that

$$\frac{\mathrm{d}\, p \odot l \odot s}{\mathrm{d}\, l} = D(p \odot l \odot s, p, l).$$

Proposition 2 (Local gradients for general folds)**.** *Let $l_i \in S$, $g = l_1 \odot \cdots \odot l_n$, and $p_i = l_1 \odot \cdots \odot l_{i-1}$ for $i \leq n$. If $(S, \odot)$ satisfies the general Local Gradient Property (definition 2), then*

$$\frac{\mathrm{d}\, g}{\mathrm{d}\, l_i} = D(g, p_i, l_i)$$

Proof. By the monoid laws, there exists s such that $g = p_i \odot l_i \odot s$. From there the result follows directly from definition 2. □

Corollary 1. *For pairwise kernels with $l_{ij} = \rho(a_i, b_j)$, $g_i = \bigodot_j l_{ij}$, and exclusive prefixes $p_{ij} = \bigodot_{k<j} l_{ik}$, the Local Gradient Property gives*

$$\bar{a}_i = \sum_j \frac{\mathrm{d}\, l_{ij}}{\mathrm{d}\, a_i}[\bar{l}_{ij}] \qquad \bar{b}_j = \sum_i \frac{\mathrm{d}\, l_{ij}}{\mathrm{d}\, b_j}[\bar{l}_{ij}] \tag{4}$$

where $\bar{l}_{ij} = D(g_i, l_{ij})[\bar{g}_i]$ in the commutative case, and $\bar{l}_{ij} = D(g_i, p_{ij}, l_{ij})[\bar{g}_i]$ in the general case.

Remark: Is the LGP about calculus? Properties 1 and 2 do not, in a strict sense, depend on differentiability, but presuppose it: they could be generalized to any property of a local factor relative to a global product. As such they are properties of monoids that are useful for differentiation, rather than properties of differentiation. Their utility lies exactly in simplifying the global question of how l_i relates to $l_1 \odot \cdots \odot l_n$ to the narrower question of whether we can determine how a relates to $a \odot b$ without knowledge of b[3].

Note on the identity element The Local Gradient Property is stated for monoids, many common operations in deep learning are naturally semigroups where an identity element is not immediately present. However, any semigroup $(S, \odot)$ can be extended to a monoid by adjoining an identity element $\mathbf{1}$. Such identity elements usually correspond to "null" initial states or masked values whose derivatives are not computed. In such cases, we can adjoin a purely formal identity element $\mathbf{1}$ and allow $D(g, l)$ (or $D(g, p, l)$) to remain undefined at $l = \mathbf{1}$.

Note on cancellability If $(S, \odot)$ is cancellable, the LGP is trivially satisfied[4]. However, the local gradient property extends beyond those instances (See appendix C).

4 The finalize-embed factorization

The Local Gradient Property is, in itself, sufficient to obtain a memory efficient recomputation based backward pass. However, as a further refinement, we factorize D through a gradient representation A:

$$\begin{aligned}
&\mathrm{embed} : S \to S^* \to A \\
&\mathrm{finalize} : S \to A \to S^* \quad \text{(commutative)} \\
&\mathrm{finalize} : S \times S \to A \to S^* \quad \text{(general)}
\end{aligned}$$

$$D(g, l) = \mathrm{finalize}(l) \circ \mathrm{embed}(g) \quad \text{(commutative)} \tag{5}$$

$$D(g, p, l) = \mathrm{finalize}(p, l) \circ \mathrm{embed}(g) \quad \text{(general)} \tag{6}$$

Where gradient computation is factorized into a nonlocal computation embed, depending on the global folded value g and its cotangent $\bar{g}$, and a local computation $\mathrm{finalize}$ depending on a (recomputed) local factor (and prefix state in

[3] Or analogous parts for the non-commutative case.

[4] Simply by reconstructing the missing factors: $D(g, p, l) = \frac{\mathrm{d}\, p \odot l \odot ((p \odot l) \oslash g)}{\mathrm{d}\, l}$ in the general case, and $D(g, l) = \frac{\mathrm{d}\, l \odot (l \oslash g)}{\mathrm{d}\, l}$ in the commutative case. This is the approach to parallel backward passes usually employed in State Space Models.

the general case). For folds, and the purpose of this paper, the benefit of this factorization lies primarily in decreasing total work, broadcast traffic, and tile-level memory pressure, since the gradient representation can be precomputed and broadcast instead of S and S^*, while usually being smaller than $S \times S^*$.[5]

Example: Factorization of D_{LWS} The LGP-style gradient function for the LogWeightedSum monoid is:

$$D(g, l)[\Delta] = \exp(l_z - g_z) \cdot \{z : \Delta_z + \langle \Delta_\mu,\, l_\mu - g_\mu \rangle,\, \mu : \Delta_\mu\}$$

By isolating terms dependent on g, we get the following:

$$\begin{aligned} \text{embed}(g)[\Delta] &= \exp(-g_z) \cdot \{z : \Delta_z - \langle \Delta_\mu,\, g_\mu \rangle,\, \mu : \Delta_\mu\} \\ \text{finalize}(l)[\eta] &= \exp(l_z) \cdot \{z : \eta_z + \langle \eta_\mu,\, l_\mu \rangle,\, \mu : \eta_\mu\} \end{aligned}$$

Since $\exp(-g_z)$ and $\exp(l_z)$ can lead to numerical instability, the alternative compact gradient representation below is preferable:

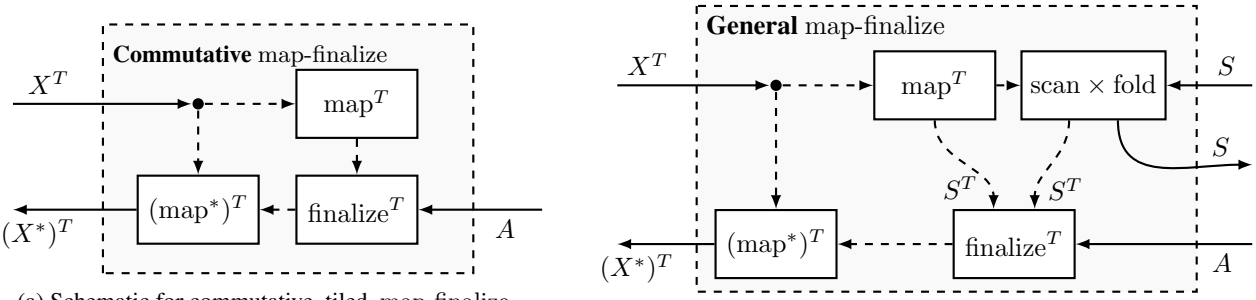
$$\begin{aligned} A &\coloneqq \left\{z \in \mathbb{R},\, s \in \mathbb{R},\, v \in \mathbb{R}^D\right\} \\ \text{embed}(g)[\Delta] &= \{z : -g_z,\, s : \Delta_z - \langle \Delta_\mu,\, g_\mu \rangle,\, v : \Delta_\mu\} \\ \text{finalize}(l)[\eta] &= \exp(l_z + \eta_z) \cdot \{z : \eta_s + \langle \eta_v,\, l_\mu \rangle,\, \mu : \eta_v\} \end{aligned}$$

In this case, the factorization shrinks tile-level residency and broadcast traffic by D: From $2D + 2$ to $D + 2$.

5 Implementation sketch

With the algebraic machinery above, we can now sketch our implementation of tiled forward and backward computation for map-fold functions over monoids satisfying the Local Gradient Property with a finalize-embed factorization.

5.1 Interface



(a) Schematic for commutative, tiled, map-finalize.

(b) Schematic for general, tiled, map-finalize.

Figure 1: The backward interface for tiled map-fold fusion: The implementation exposes callbacks for batched, tile-local map-finalize functions acting on the result of embed, input tensor tiles X^T, and, in the case of non-commutative folds, exclusive prefix state.

For both the commutative and non-commutative case, the implementation requires user-supplied functions $\odot$, embed, map-fold and map-finalize, which are map-fused folds and finalize-functions, respectively. The logic and interface for map-finalize differs between the commutative and non-commutative case, as shown in Figure 1, due to the need (in the non-commutative case) to thread prefix state through the map-finalize logic.

5.2 Commutative Tiled map-fold Fusion

Forward pass The forward and backward passes for tiled commutative folds are illustrated in figure 2. The input X^F is tiled into a Program, Loop, and Tile-partitioned view $X^{P \times L \times T}$, which is then fed into a partial map-fold stage:

[5] However, a slight extension of this factorization is crucial for extending to framework to scans, where suffix accumulation over A is required. See appendix B.1.

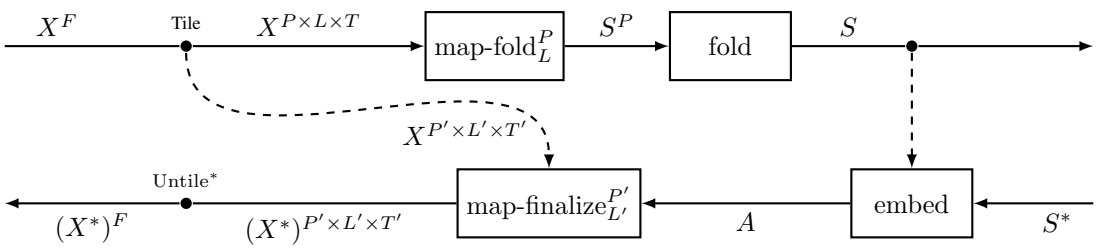


Figure 2: Forward and backward passes of map-fold functions over commutative $(S, \odot)$.

map-fold$_L^P$ denotes a fused map-fold function parallelized over P programs, with each program looping over L tiles of size T, producing output of type S^P:

$$\text{map-fold}_L^P : X^{P\times L\times T} \to S^P$$
$$\text{map-fold}_L^P(\mathbf{x})_p = \bigodot_{l\in L} (\text{map-fold}(\mathbf{x}_{p,l}))$$

The P independent partial results are written to memory and fed to a separate fold stage[6] to produce the final result S. Only the input (X^F) and final result (S) are persisted for the backward pass.

Example: Attention Consider attention. The interaction X is a Query, Key, and Value-vector triplet: $\{q \in \mathbb{R}^D,\ k \in \mathbb{R}^D,\ v \in \mathbb{R}^D\}$. map transforms that triplet to a single LWS factor:

$$\text{map}(x) = \{z : \langle x_q,\ x_k\rangle,\ \mu : x_v\} \tag{7}$$

The user supplied function map-fold corresponds to $\text{fold}^{T_B} \circ \text{map}^{T_B\times T_F}$:

$$\text{map-fold}(\mathbf{x}) : X^{T_B\times T_F} \to S^{T_B} \tag{8}$$
$$\text{map-fold}(\mathbf{x}) = \{z : \text{lse}(\mathbf{l}),\ \mu : \exp(\mathbf{l} - z)\mathbf{x}_v\}, \qquad \mathbf{l} = \mathbf{x}_q\mathbf{x}_k^\intercal \tag{9}$$

Where $\mathbf{x}_q \in \mathbb{R}^{T_B\times D}, \mathbf{x}_k \in \mathbb{R}^{T_F\times D}, \mathbf{x}_v \in \mathbb{R}^{T_F\times D}$

Backward pass The backward pass uses the finalize-embed-factorization to first embed the output and output gradient into compact representation $\eta : A$. This is then broadcast to the P'-parallel L'-looping map-finalize$_{L'}^{P'}$-function[7]. For gradient buffers whose corresponding input does not depend on the loop axis, the gradient contributions are accumulated in place across loop iterations and written to global gradients at program end. For gradient buffers that depend on the loop axis, the local gradient is written to global gradients directly in the loop. In both cases, the write is an atomic add if several programs are contributing to the same storage element, and a plain write if not.

Example: Attention For attention we use the compact gradient representation for LWS discussed in 4, i.e:

$$A = \{z \in \mathbb{R},\ s \in \mathbb{R},\ v \in \mathbb{R}^D\} \tag{10}$$

embed and finalize are direct consequences of the LWS monoid. The user supplied embed function is that same function vectorized over the batch axes. However, map-finalize depends on the map function, and takes the following form:

$$\text{map-finalize} : X^{T_B\times T_F} \times A^{T_B} \to (X^*)^{T_B} \tag{11}$$
$$\text{map-finalize}(\mathbf{x}, \eta) = \{q : \bar{\mathbf{l}}\mathbf{x}_k,\ k : \bar{\mathbf{l}}^\intercal\mathbf{x}_q,\ v : \mathbf{p}^\intercal\eta_v\} \tag{12}$$
$$\text{where} : \begin{matrix}\mathbf{p} & = \exp(\mathbf{x}_q\mathbf{x}_k^\intercal + \eta_z) \\ \bar{\mathbf{l}} & = \mathbf{p}\cdot(\eta_s + \eta_\mu\mathbf{x}_v^\intercal)\end{matrix} \tag{13}$$

[6]In the case where one program is responsible for the full fold (i.e. $P = 1$), the fold stage is omitted. Note that this often is a valid strategy: With large enough batch dimension, parallelizing over batches can achieve sufficient occupancy.

[7]Under batching, it can be better to loop over a batch axis during the backward pass. This is handled by the framework: During the analytical sweep, we vary the loop axis across all relevant axes. This has been omitted from the diagrams due to brevity

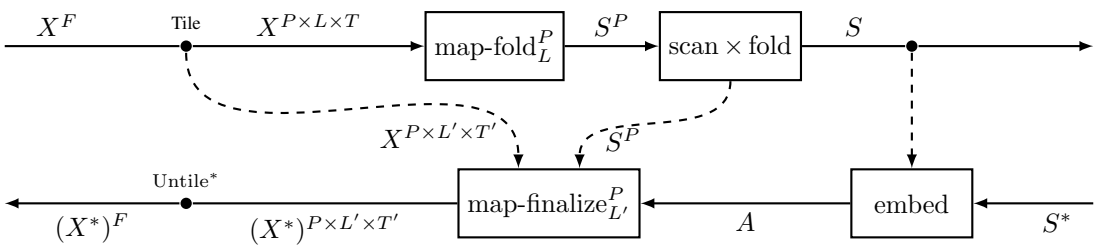


Figure 3: Forward and backward passes of map-fold functions over non-commutative $(S, \odot)$.

5.3 General Tiled map-fold Fusion

Forward pass For a general monoid, the partial summaries cannot be combined in an arbitrary order. The partial map-fold stage therefore produces an ordered sequence $\mathbf{q} \in S^P$, where $\mathbf{q}_p$ summarizes the fold tiles assigned to program p. The second stage performs an ordered exclusive scan over these summaries, and also returns the total fold:

$$\begin{aligned} \text{scan} \times \text{fold} &: S^P \to S^P \times S \\ (\text{scan} \times \text{fold})(\mathbf{q}) &= (\mathbf{s}, g), \\ \mathbf{s}_p = \bigodot_{j<p} \mathbf{q}_j, &\qquad g = \bigodot_{j} \mathbf{q}_j. \end{aligned}$$

The value s_p is the left-boundary state for program p. It is exactly the state needed to resume the ordered fold inside that program. The implementation may either persist these boundary states for the backward pass or recompute them from the partial summaries.

Backward pass The backward pass first applies embed to the global output state and its cotangent, producing the compact gradient accumulator A. Each backward program receives this same accumulator together with its left-boundary state $\mathbf{s}_p$. It then recomputes the local factors for its assigned tiles and threads a running prefix through the loop:

$$\mathbf{p}_{p,0} = \mathbf{s}_p, \qquad \mathbf{p}_{p,l+1} = \mathbf{p}_{p,l} \odot \mathbf{l}_{p,l}.$$

For each tile, map-finalize consumes the current prefix, the recomputed local factor, and the embedded gradient accumulator, and emits gradients for the input tile. Thus the only additional state required by the general case, compared to the commutative case, is the ordered prefix boundary and the intra-program prefix threaded through the recomputation loop.[8]

6 Execution Carriers

Semantic monoids are not always the representation used in an efficient kernel. For example, log-sum-exp is often computed using a maximum and a rescaled exponential sum. We model this using the notion of an execution carrier.

Definition 3 (Execution Carrier)**.** *Let $(S, \odot_S)$ and $(E, \odot_E)$ be monoids. We call E an execution carrier for S if $\Psi : (E, \odot_E) \to (S, \odot_S)$ is a surjective monoid homomorphism (with right inverse $\Phi : S \to E$).*

$$\Psi(e_1 \odot_E e_2) = \Psi(e_1) \odot_S \Psi(e_2), \qquad \Psi \circ \Phi = \mathrm{id}_S \tag{14}$$

If E is an execution carrier for S, it follows immediately that $\bigodot_i l_i = \Psi(\bigodot_i \Phi(l_i))$, for $l_i : S$. This bring two benefits

- Execution carriers offer representational freedom: We can compute folds of S in E, without threading the gradient computation through E. The fused max trick can be seen as an instances of this.
- Some monoids S do not, in themselves, satisfy a Local Gradient Property, but do it via an execution carrier E, in which case we say the E resolves the LGP for S. This applies primarily to non-cancellable monoids (See Appendix C).

[8] In contrast to the commutative case, this limits the feasibility of looping over batch axes in the backward pass: That is only possible in the current framework if map-finalize does not need to thread prefix state computation in the loop.

Example: Fused Max-Trick as an Execution Carrier for LWS The online scaling and denominator tracking used in the forward pass of FlashAttention (Dao et al., 2022) can be formalized as the following execution carrier for the LWS monoid, essentially fusing the logaddexp max-trick:

$$E_{LWS} \coloneqq \left\{m \in \mathbb{R},\, e \in \mathbb{R},\, u \in \mathbb{R}^D\right\}$$
$$a \odot_{E_{LWS}} b = \{m : h_m,\, e : h_e + \alpha \cdot l_e,\, u : h_u + \alpha \cdot l_u\}$$
$$\text{where } \alpha = \exp(l_m - h_m), \quad (h, l) = \begin{cases}(a, b) & a_m > b_m \\ (b, a) & a_m \le b_m\end{cases}$$

$$\Phi(x) = \{m : x_z,\, e : 1,\, u : x_\mu\}, \quad \Psi(y) = \left\{z : y_m + \log(y_e),\, \mu : \frac{y_u}{y_e}\right\}$$

7 TRANSPORT

While we can not give a complete characterization of when monoids satisfy the Local Gradient Property, we can show that the local gradient property is preserved by smooth, injective, monoid homomorphisms, and thus offer a way to construct new LGP-satisfying monoids from existing ones.

Definition 4 (Transport)**.** *For monoids $(S, \odot)$ and $(T, \otimes)$, we say that S transports to T if there exists an injective, smooth homomorphism $\phi : (S, \odot) \hookrightarrow (T, \otimes)$, with a smooth left inverse on its image ($\psi : \phi(S) \to S$):*

$$\phi(s_1 \odot_S s_2) = \phi(s_1) \odot_T \phi(s_2), \qquad \psi \circ \phi = \mathrm{id}_S \tag{15}$$

Proposition 3 (Transport of the Local Gradient Property)**.** *If S transports to T, and T satisfies a Local Gradient Property with $D_T = \mathrm{finalize}_T \circ \mathrm{embed}_T$, then so does S with*

$$D_S(g, l) = \frac{\mathrm{d}\,\phi(l)}{\mathrm{d}\,l} \circ D_T(\phi(g), \phi(l)) \circ \frac{\mathrm{d}\,\psi(\phi(g))}{\mathrm{d}\,\phi(g)} \tag{16}$$

$$\mathrm{finalize}_S(l) = \frac{\mathrm{d}\,\phi(l)}{\mathrm{d}\,l} \circ \mathrm{finalize}_T(\phi(l)) \tag{17}$$

$$\mathrm{embed}_S(g) = \mathrm{embed}_T(\phi(g)) \circ \frac{\mathrm{d}\,\psi(\phi(g))}{\mathrm{d}\,\phi(g)} \tag{18}$$

in the commutative case and

$$D_S(g, p, l) = \frac{\mathrm{d}\,\phi(l)}{\mathrm{d}\,l} \circ D_T(\phi(g), \phi(p), \phi(l)) \circ \frac{\mathrm{d}\,\psi(\phi(g))}{\mathrm{d}\,\phi(g)} \tag{19}$$

$$\mathrm{finalize}_S(p, l) = \frac{\mathrm{d}\,\phi(l)}{\mathrm{d}\,l} \circ \mathrm{finalize}_T(\phi(p), \phi(l)) \tag{20}$$

$$\mathrm{embed}_S(g) = \mathrm{embed}_T(\phi(g)) \circ \frac{\mathrm{d}\,\psi(\phi(g))}{\mathrm{d}\,\phi(g)} \tag{21}$$

in the general case.

Proof. The results is a direct consequence of the properties of monoid homomorphisms, the chain rule, and associativity of function composition. □

Two base cases are particularly useful: $GL_N(\mathbb{R})$ satisfies the general LGP while $(\mathbb{R}^N, +)$ satisfies the commutative LGP. As an example LWS transports to $(\mathbb{R}^{D+1}, +)$, with the unstable log-weighted-sum factorization in Section 4 the direct result from Proposition 3. For derivation see Appendix D.

8 CUTILEREDUCE

CuTileReduce is a CUDA Tile implementation of the fold interface sketched above. A user provides a fold specification and a collection of callbacks. The fold specification is used to declare symbolic axes, their roles, and how these are shared between input, execution carrier, and semantic buffers, while also supplying a rudimentary model for intermediate values realized inside the kernel and matrix multiplication work. The concrete implementation follows

Table 1: Scaling timings in milliseconds.

Workload	N	forward		forward + backward	
		CuTileReduce	PyTorch	CuTileReduce	PyTorch
Cross-Entropy	2,048	0.077	0.086	0.389	0.389
	4,096	0.132	0.244	0.589	0.461
	8,192	0.464	0.498	1.4	1.2
Attention	2,048	0.954	0.908	3.3	2.9
	4,096	3.6	3.1	11.4	10.4
	8,192	13.7	11.5	42.9	39.3
Affine Attention	2,048	0.586	0.823	2.2	1.9
	4,096	1.9	3.4	7.6	7.5
	8,192	8.7	13.0	36.8	30.5

Table 2: PyTorch-managed CUDA allocation (disregarding inputs and cotangents) in MiB.

Workload	N	forward		forward + backward	
		CuTileReduce	PyTorch	CuTileReduce	PyTorch
Cross-Entropy	2,048	0.134	80.152	64.040	80.148
	4,096	0.267	40.164	64.079	42.133
	8,192	0.345	136.203	64.157	140.141
Attention	2,048	32.251	32.251	128.501	164.501
	4,096	64.501	64.501	242.001	329.001
	8,192	129.001	129.001	484.001	658.001
Affine Attention	2,048	4.126	200.125	72.188	266.625
	4,096	8.251	840.125	80.376	1,037.125
	8,192	16.501	4,168.125	96.751	4,178.125

the staged structure in Figures 2 and 3, extended to account for batching and execution carriers[9]. Scheduling choices, i.e. tile sizes, number of programs along the fold axis, backward loop axis (in the commutative case), are selected based on an analytical, roofline-based, sweep over legal tile and partition choices, based on the user provided fold specification and device-properties. The top-K performing tile and partition choices are then empirically timed to select stage candidates. For more details, see Appendix H.

9 Evaluation

We report peak PyTorch-managed CUDA allocation and wall-clock benchmarks for CuTileReduce-derived kernels, and compare these to pytorch references. We do this for three instances: Cross-entropy, Attention, and a non-commutative attention-variant (See appendix E.5) on an RTX5080.

The CuTileReduce kernels are tuned for each problem size using an analytical sweep followed by empirical benchmarking. Both CuTileReduce kernel and PyTorch reference use `torch.compile`. For Cross-Entropy and Affine Attention we use a naive PyTorch reference, whereas the Attention reference uses PyTorch SDPA[10]. Timing results can be seen in Table 1, whereas peak memory usage can be seen in Table 2.

10 Related Work

To mitigate the memory and bandwidth bottlenecks of backpropagation, conventional approaches rely on gradient checkpointing (Chen et al., 2016), which trades recomputation for peak memory reduction based on checkpointing schedules (Griewank and Walther, 2000). More recently, hardware-aware design has shifted towards custom fused kernels. FlashAttention (Dao et al., 2022; Dao, 2023) utilizes online softmax reductions to calculate attention without storing intermediate logits, and uses kernel-local recomputation during the backward pass. Similarly, work such as Liger Kernel (Hsu et al., 2024) and Cut Cross-Entropy (Wijmans et al., 2024) extend that same idea to cross-entropy. Our work formalizes these disparate engineering efforts under a unified algebraic perspective: They are map-fold functions over monoids satisfying the Local Gradient Property, using a corresponding map-finalize fused backward pass.

A separate line of related work is that in State Space Models, and efficient *scans* over associative operators (usually affine). However, the situation there is similar to that of existing map-fold fused kernels: Instances are hand-derived

[9] In particular, $\mathrm{map\text{-}fold}$ produces folded execution carrier state, and, in the general case, $\mathrm{map\text{-}finalize}$ threads execution carrier prefix state through the computation, seeded by the persisted execution carrier state from the forward pass. The forward pass outputs final semantic state computed via Ψ, this final semantic state is reused by embed in the backward pass.

[10] The exact setup of CrossEntropy, Attention, and Affine Attention can be seen in Appendix F

Table 3: Mean absolute error against the PyTorch FP64 reference. "Worst" is the maximum MAE over all input gradients for the indicated workload and size. CuTile uses BF16 inputs with FP32 execution-carrier accumulation.

Function	N	|CuTileReduce BF16 − PyTorch FP64|		|PyTorch BF16 − PyTorch FP64|	
		out	worst (gradient)	out	worst (gradient)
Cross-Entropy	1,024	2.8e−7	8.8e−5	7.9e−3	9.0e−5
	4,096	3.9e−7	8.0e−5	1.6e−2	1.1e−4
Attention	1,024	6.0e−5	8.8e−4	8.8e−5	1.1e−3
	4,096	3.2e−5	4.4e−4	4.5e−5	5.5e−4
Affine attention	1,024	1.9e−4	3.1e−2	1.7e−2	7.5e−1
	4,096	1.9e−4	6.1e−2	3.8e−2	3.0

and treated on a case by case basis, with no general approach to the problemYang et al. (2025); Guo et al. (2025); Gu and Dao (2024). It should be noted that we do not present a general approach, either, but believe that the machinery presented extends to scans as well (See Appendix B.1, G).

MapReduce (Dean and Ghemawat, 2008) has long been used for cluster-scale map-fold fusion. Recently, frameworks such as DrJAX (Rush et al., 2024) have brought MapReduce primitives to automatic differentiation (AD) frameworks. However, DrJAX only addresses sum/mean reductions, a case with a trivial (identity) Local Gradient Property, and is mainly concerned with sharding and cross-runtime translation rather than tile-level memory efficiency. Closer in spirit to our approach is the development of AD in functional array compilers. Specifically, Schenck et al. (2022) and Bruun et al. (2023) derived reverse-mode AD rules for parallel building blocks like reductions and scans, including a general case structurally similar to our general Local Gradient Property, but requiring invertibility. Unlike our setting, they do not extend this to map-fold fusion, in effect requiring realization of all local factors.

11 Discussion

In this paper we've shown that the Local Gradient Property and the $\mathrm{finalize\text{-}embed}$ factorization can be used as an algebraic backward interface for map-fold operators, while the implementation CuTileReduce, demonstrates that this interface can generate practical CUDA Tile forward and backward kernels for such operators, both known and new.

However, that implementation should be viewed primarily as a Proof of Concept to support the algebraic contribution: It has not been extensively tested, only supports a single fold/loop axis, and requires that inner axes (i.e. axes that are neither batch nor fold axes) fit within one program tile. The latter limitation is only one of several that arises from the chosen callback interface ($\mathrm{map\text{-}fold}$, $\mathrm{map\text{-}finalize}$): the interface also blocks possible optimizations at the map-level, decomposition of monoids, (e.g. splitting along the payload dimension for LWS-style monoids), and more.

On the algebraic side, the Local Gradient Property is a sufficient condition, not a necessary one. Some operations may admit memory-efficient backward passes outside this framework. Conversely, satisfying the property does not imply that recomputation is always a good performance tradeoff: If the cost of recomputation exceeds the memory savings benefit, there is little benefit to the approach.

12 Future Work

The most immediate future work is extending the framework to map scan fusion, and in particular, map scan fold fusion, where the scan is performed over one axis, and the fold performed on scanned values over a separate axis. In the appendix, we develop an extension to the $\mathrm{finalize\text{-}embed}$ factorization for that purpose (Appendix B.1) and present a rough theoretical blueprint in Figure 5

Another avenue of future work is to generalize the proposed framework beyond user supplied instantiations of $\mathrm{map\text{-}finalize}$ and $\mathrm{map\text{-}fold}$. The current framework requires that the user supplies tile-local CuTile functions, which are then composed to construct a fused map-fold kernel for both forward and backward passes. This enforces the limitation discussed earlier, that inner axes are resident in the kernel. A generalized Domain Specific Language could alleviate this limitation, enabling problem sizes, kernels, and optimizations that are not possible in the current framework.

Another immediate extension is to adapt the framework to the distributed setting: The algebraic logic remains the same, with the fold and $\mathrm{scan} \times \mathrm{fold}$-stages lifted to the distributed setting.

REFERENCES

Bruun, L. M., Larsen, U. S., Hinnerskov, N. H., and Oancea, C. E. (2023). Reverse-mode ad of multi-reduce and scan in futhark. In *Proceedings of the 35th Symposium on Implementation and Application of Functional Languages*, pages 1–14.

Chen, T., Xu, B., Zhang, C., and Guestrin, C. (2016). Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*.

Dao, T. (2023). Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*.

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*.

Dean, J. and Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.

Griewank, A. and Walther, A. (2000). Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Trans. Math. Softw.*, 26(1):19–45.

Gu, A. and Dao, T. (2024). Mamba: Linear-time sequence modeling with selective state spaces. In *First Conference on Language Modeling*.

Guo, H., Yang, S., Goel, T., Xing, E. P., Dao, T., and Kim, Y. (2025). Log-linear attention. *arXiv preprint arXiv:2506.04761*.

Hsu, P.-L., Dai, Y., Kothapalli, V., Song, Q., Tang, S., Zhu, S., Shimizu, S., Sahni, S., Ning, H., and Chen, Y. (2024). Liger kernel: Efficient triton kernels for llm training. *arXiv preprint arXiv:2410.10989*.

Rush, K., Charles, Z., Garrett, Z., Augenstein, S., and Mitchell, N. (2024). Drjax: Scalable and differentiable mapreduce primitives in jax. *arXiv preprint arXiv:2403.07128*.

Schenck, R., Rϕnning, O., Henriksen, T., and Oancea, C. E. (2022). Ad for an array language with nested parallelism. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE.

Wijmans, E., Huval, B., Hertzberg, A., Koltun, V., and Krähenbühl, P. (2024). Cut your losses in large-vocabulary language models. *arXiv preprint arXiv:2411.09009*.

Yang, S., Kautz, J., and Hatamizadeh, A. (2025). Gated delta networks: Improving mamba2 with delta rule. In *International Conference on Learning Representations*, volume 2025, pages 29687–29707.

A AI USE STATEMENT

In this work, we used generative AI tools for the purpose of refining, proofreading and providing feedback on the algebraic claims made. The code for CuTileReduce is written almost exclusively by code agents, based on a smaller working prototype. Generative AI tools have also been used extensively during the writing process, both for drafting sections, and for refining text. In some instances (Appendix C, H) AI generated text has been used with very little post-processing. We have reviewed all AI-assisted work and take responsibility for the final content of this work.

B SCANS

B.1 THE finalize-embed FACTORIZATION FOR SCANS

For scans, the interface proposed so far breaks down, since $\bar{l}_i$ require access to all future prefixes p_j and their gradients $\bar{p}_j$:

$$\bar{l}_i = \sum_{j>i} D(p_j, p_i, l_i)\,[\bar{p}_j]\,, \quad \text{with the convention } p_{n+1} \coloneqq g,\ \bar{p}_{n+1} \coloneqq \bar{g} \tag{22}$$

To address this limitation, we extend on the finalize-embed factorization, requiring that it factors through a partial semigroup $(A, \oplus)$, such that

$$\sum_{j>i} D(p_j, p_i, l_i)\,[\bar{p}_j] = \text{finalize}(p_i, l_i)\left[\bigoplus_{j>i} \text{embed}(p_j)\,[\bar{p}_j]\right] \tag{23}$$

From which it follows that:

$$\bar{l}_i = \text{finalize}(p_i, l_i)\,[\eta_i] \qquad \eta_i = \bigoplus_{j>i} \text{embed}(p_j)\,[\bar{p}_j] \tag{24}$$

The transport property (Proposition 3) remains unchanged under this stricter notion of the finalize-embed factorization.

Example: Gradient accumulator for LWS *is* LWS In 4 we gave an unstable finalize-embed factorization of LWS as:

$$\begin{aligned}\text{embed}(g)[\Delta] &= \exp(-g_z) \cdot \{z : \Delta_z - \langle \Delta_\mu,\, g_\mu \rangle,\ \mu : \Delta_\mu\} \\ \text{finalize}(l)[\eta] &= \exp(l_z) \cdot \{z : \eta_z + \langle \eta_\mu,\, l_\mu \rangle,\ \mu : \eta_\mu\}\end{aligned}$$

This illustrates an interesting duality: The gradient accumulator for a LogWeightedSum *is* a LogWeightedSum, giving the following alternative gradient accumulator: [11]

$$\begin{aligned}A &\coloneqq LWS(\{s \in \mathbb{R},\, v \in \mathbb{R}^D\}) \\ \text{embed}(g)[\Delta] &= \{z : -g_z,\ s : \Delta_z - \langle \Delta_\mu,\, g_\mu \rangle,\ v : \Delta_\mu\} \\ \text{finalize}(l)[\eta] &= \exp(l_z + \eta_z) \cdot \{z : \eta_s + \langle \eta_v,\, l_\mu \rangle,\ \mu : \eta_v\}\end{aligned}$$

The execution carrier discussed in section 6 for LWS can be used for the gradient accumulator as well.

C Beyond Cancellability

Non-cancellable monoids fail to satisfy the bare Local Gradient Property whenever their algebraic collapse (e.g., absorption, idempotency, or right-annihilation) destroys the information required to identify an input factor's contribution. However, this failure is often an artifact of differentiating solely over the coarse semantic monoid S. By lifting computation to an appropriate execution carrier E, the Local Gradient Property and its finalize-embed factorization can be resolved without materializing intermediate factor tensors.

Definition 5 (Resolving LGP via Carrier)**.** *Let $(S, \odot_S)$ and $(E, \odot_E)$ be monoids, where E is an execution carrier for S. We say that E* resolves the LGP for *S if there exist functions D,* embed*, and* finalize *such that:*

$$\textit{Commutative:} \quad \frac{\mathrm{d}\, g}{\mathrm{d}\, l_i} = D(\hat{g}, l_i) = \text{finalize}(l_i) \circ \text{embed}(\hat{g})$$

$$\textit{General:} \quad \frac{\mathrm{d}\, g}{\mathrm{d}\, l_i} = D(\hat{g}, \hat{p}_i, l_i) = \text{finalize}(\hat{p}_i, l_i) \circ \text{embed}(\hat{g})$$

where $g = l_1 \odot_S \cdots \odot_S l_n$, $\hat{g} = \Phi(l_1) \odot_E \cdots \odot_E \Phi(l_n)$, $\hat{p}_i = \Phi(l_1) \odot_E \cdots \odot_E \Phi(l_{i-1})$.

Below, we demonstrate how this carrier extension resolves the LGP for three non-cancellable monoids.

[11] Here, $LWS(x)$ denotes a LWS-monoid with a weighted sum of type x. That is, for $LWS(\{s \in \mathbb{R},\, v \in \mathbb{R}^D\})$

$$a \odot b = \begin{Bmatrix} z : \text{logaddexp}(a_z, b_z) \\ s : \exp(a_z - z) \cdot a_s + \exp(b_z - z) \cdot b_s \\ v : \exp(a_z - z) \cdot a_v + \exp(b_z - z) \cdot b_v \end{Bmatrix} \tag{25}$$

C.1 $(\mathbb{R}, \max)$ WITH TIE-COUNTING SUBGRADIENTS

Let the semantic monoid be $(S, \odot_S) = (\mathbb{R}, \max)$. Because $\max$ is idempotent ($x \max x = x$), S cannot record the multiplicity of maximal elements. Under the subgradient convention where gradient mass is distributed equally among all k tied maximizers, S alone cannot determine k.

Carrier. We define the tie-counting execution carrier:

$$E \coloneqq \{m \in \mathbb{R},\, t \in \mathbb{N}\}$$
$$a \odot_E b \coloneqq \begin{cases} \{m : a_m,\, t : a_t + b_t\} & \text{if } a_m = b_m \\ a & \text{if } a_m > b_m \\ b & \text{if } b_m > a_m \end{cases}$$

with identity $\mathbf{1}_E = \{m : -\infty,\, t : 0\}$, embedding $\Phi(x) = \{m : x,\, t : 1\}$, and projection $\Psi(\hat{x}) = \hat{x}_m$.

Resolved LGP and Factorization. Given the accumulated carrier state $\hat{g} = \bigodot_j \Phi(l_j)$, E resolves the commutative LGP for S via:

$$D(\hat{g}, l)[\Delta] = \begin{cases} \frac{1}{\hat{g}_t} \cdot \Delta & \text{if } l = \hat{g}_m \\ 0 & \text{if } l < \hat{g}_m \end{cases} \tag{26}$$

which factors through $A \coloneqq \{m \in \mathbb{R},\, \gamma \in \mathbb{R}\}$:

$$a \oplus b = \begin{cases} \{m : a_m,\, \gamma : a_\gamma + b_\gamma\} & \text{if } a_m = b_m \\ a & \text{if } a_m < b_m \end{cases}$$
$$\text{embed}(\hat{g})[\Delta] = \left\{ m : \hat{g}_m,\, \gamma : \frac{1}{\hat{g}_t}\Delta \right\}$$
$$\text{finalize}(l)[\eta] = \begin{cases} \eta_\gamma & \text{if } l = \eta_m \\ 0 & \text{if } l < \eta_m \end{cases}$$

Note: The partiality of $a \oplus b$, and the reversed ordering is not a typo: Since $a = \text{embed}(x)$, and $b = \text{embed}(x \odot y)$, we have that $a_m \leq b_m$, while the reverse ordering ensures the correct attribution.

C.2 $(\mathbb{R}, \times)$ WITH ZERO-COUNTING

Let the semantic monoid be $(S, \odot_S) = (\mathbb{R}, \cdot)$. The element 0 is an absorbing annihilator ($x \cdot 0 = 0$). If $g = 0$ and a factor $l_i = 0$, S cannot determine whether l_i was the *unique* zero factor (in which case $\frac{\mathrm{d}\, g}{\mathrm{d}\, l_i} = \prod_{j \neq i} l_j$) or whether other zeros were present (in which case $\frac{\mathrm{d}\, g}{\mathrm{d}\, l_i} = 0$).

Carrier. We define an execution carrier separating the non-zero product from the zero multiplicity:

$$E \coloneqq \{p \in \mathbb{R},\, n \in \mathbb{N}\}$$
$$a \odot_E b \coloneqq \{p : a_p \cdot b_p,\, n : a_n + b_n\}$$

with identity $\mathbf{1}_E = \{p : 1,\, n : 0\}$. The morphisms are:

$$\Phi(x) = \begin{cases} \{p : x,\, n : 0\} & \text{if } x \neq 0 \\ \{p : 1,\, n : 1\} & \text{if } x = 0 \end{cases}, \qquad \Psi(\hat{x}) = \begin{cases} \hat{x}_p & \text{if } \hat{x}_n = 0 \\ 0 & \text{if } \hat{x}_n > 0 \end{cases} \tag{27}$$

Resolved LGP and Factorization. E resolves the LGP for S through $(A, \oplus)$ via:

$$A \coloneqq \{p \in \mathbb{R},\, n \in \mathbb{N}\}$$
$$a \oplus b \coloneqq \begin{cases} \{p : a_p + b_p,\, n : a_n\} & \text{if } a_n = b_n \\ a & \text{if } a_n < b_n \end{cases}$$
$$\text{embed}(\hat{g})[\Delta] = \{p : \hat{g}_p \cdot \Delta,\, n : \hat{g}_n\}$$
$$\text{finalize}(l)[\eta] = \begin{cases} \frac{\eta_p}{\Phi(l)_p} & \text{if } \eta_n = \Phi(l)_n \\ 0 & \text{if } \eta_n > \Phi(l)_n \end{cases}$$

C.3 $\mathrm{Right}(X)$ WITH LENGTH TRACKING

Consider the non-commutative right-projection monoid over a smooth space X with an adjoined identity: $S \coloneqq X \cup \{\bot\}$, with operation

$$a \odot_S b \coloneqq \begin{cases} a & \text{if } b = \bot \\ b & \text{if } b \neq \bot \end{cases} \tag{28}$$

S is non-cancellable, and the semantic state g alone cannot identify which factor index was the final valid update.

Carrier. We track the running rightmost value alongside the number of non-identity elements:

$$E \coloneqq \{v \in X \cup \{\bot\},\, c \in \mathbb{N}\}$$
$$a \odot_E b \coloneqq \{v : v',\, c : a_c + b_c\}\,, \quad \text{where } v' = \begin{cases} b_v & \text{if } b_c > 0 \\ a_v & \text{if } b_c = 0 \end{cases}$$

with identity $\mathbf{1}_E = \{v : \bot,\, c : 0\}$, embedding $\Phi(x) = \left\{v : x,\, c : \mathbf{1}_{\{x \neq \bot\}}\right\}$, and projection $\Psi(\hat{x}) = \hat{x}_v$.

Resolved LGP and Factorization. Under the general Local Gradient Property, a local factor $l \neq \bot$ receives a non-zero gradient from g if and only if no subsequent element overrode it, i.e., when $\hat{p} \odot \Phi(l) = \hat{g}$:

$$D(\hat{g}, \hat{p}, l)[\Delta] = \begin{cases} \Delta & \text{if } \hat{p}_c + \Phi(l)_c = \hat{g}_c \\ 0 & \text{otherwise} \end{cases} \tag{29}$$

This factors through $A \coloneqq \{v \in X^*,\, c \in \mathbb{N}\}$ via:

$$a \oplus b = \begin{cases} \{a_v + b_v,\, a_c\} & \text{if } a_c = b_c \\ a & \text{if } a_c < b_c \end{cases}$$
$$\mathrm{embed}(\hat{g})[\Delta] = \{v : \Delta,\, c : \hat{g}_c\}$$
$$\mathrm{finalize}(\hat{p}, l)[\eta] = \begin{cases} \eta_v & \text{if } \hat{p}_c + \Phi(l)_c = \eta_c \\ 0 & \text{otherwise} \end{cases}$$

D TRANSPORT

D.1 $GL_N(\mathbb{R})$ SATISFIES THE GENERAL LGP

For GL_N we have that

$$D(g, p, l)[\Delta] = p^\intercal \Delta g^\intercal p^{-\intercal} l^{-\intercal} \tag{30}$$
$$\mathrm{finalize}_{GL_N(\mathbb{R})}(p, l)[\eta] = p^\intercal\, \eta\, p^{-\intercal} l^{-\intercal} \tag{31}$$
$$\mathrm{embed}_{GL_N(\mathbb{R})}(g)[\Delta] = \Delta g^\intercal \tag{32}$$

By Proposition 3, any monoid S with transport $(S, \odot) \hookrightarrow GL_N(\mathbb{R})$ satisfies the General LGP with the following concrete instances of $\mathrm{finalize}$ and embed, factoring through the additive matrix semigroup $M_N(\mathbb{R})$ (see Appendix B.1):

$$\mathrm{finalize}_S(p, l)[\eta] = \frac{\mathrm{d}\,\phi(l)}{\mathrm{d}\,l}\left[\phi(p)^\intercal\, \eta\, \phi(p)^{-\intercal} \phi(l)^{-\intercal}\right] \tag{33}$$
$$\mathrm{embed}_S(g)[\Delta] = \frac{\mathrm{d}\,\psi(\phi(g))}{\mathrm{d}\,\phi(g)}\,[\Delta]\ \phi(g)^\intercal \tag{34}$$

D.2 $(\mathbb{R}^N, +)$ SATISFIES THE COMMUTATIVE LGP

A simpler and quite common special case of $GL_N(\mathbb{R})$ is $(\mathbb{R}^N, +)$. This satisfies the Commutative LGP:

$$D_{(\mathbb{R}^N,+)}(a \odot b, a) = \mathrm{id}$$
$$\mathrm{finalize}_{(\mathbb{R}^N,+)}(l) = \mathrm{id}$$
$$\mathrm{embed}_{(\mathbb{R}^N,+)}(g) = \mathrm{id}$$

By Proposition 3, any monoid S with transport $(S, \odot) \hookrightarrow (\mathbb{R}^N, +)$ satisfies the Commutative LGP with the following concrete instances of finalize and embed, factoring through the additive semigroup $(\mathbb{R}^N, +)$ (see Appendix B.1):

$$\mathrm{finalize}_S(l) = \frac{\mathrm{d}\,\phi(l)}{\mathrm{d}\,l} \tag{35}$$
$$\mathrm{embed}_S(g) = \frac{\mathrm{d}\,\psi(\phi(g))}{\mathrm{d}\,\phi(g)} \tag{36}$$

Example: LWS is transportable to $(\mathbb{R}^N, +)$ LogWeightedSum is transportable to $(\mathbb{R}^N, +)$ by transforming the log-space weight to the corresponding total weight by exponentiation, and by transforming the mean to the total sum by multiplication by the total weight, along with the inverse operation:

$$\phi(a) = \{w : \exp(a_z),\ s : w \cdot a_\mu\}$$
$$\psi(x) = \left\{z : \log(x_w),\ \mu : \frac{1}{x_w} \cdot x_s\right\}$$

D.3 Cancellability is transported

If S transports to T, and T is left and right cancellable, then so is S: $a \oslash_S b = \psi(\phi(a) \oslash_T \phi(b))$, which specializes to $\psi(\phi(a) \odot_T \phi(b)^{-1})$ in the case where T is a group. Note that $\oslash$ is not, in general, a total function: $\phi(a) \oslash_T \phi(b)$ is not necessarily in the image ϕ, and thus outside the domain of ψ.

D.4 Transport as existence proof, not implementation

The transport propositions (3) should be read as existence results and as a derivation aid: it guarantees that relevant properties exists for a monoid S whenever an injective, smooth homomorphism into another monoid T with those properties exist, and they supply a first candidate formula by direct substitution. They are not a claim that the substituted formula is numerically preferable to implement. In particular, the transported formulas route every quantity through $\phi(g)$ and $\phi(l)$ explicitly e.g. $\exp(g_z)$ in the LWS-case. In practice, we treat the transported factorization as a specification to be refined.

E Instances

E.1 Attention

For single-headed attention, we use the LWS-monoid:

$$\mathrm{attention}(Q, K, V)_i = \left(\bigodot_j \{z : \langle Q_i,\ K_j\rangle,\ \mu : V_j\}\right)_\mu \tag{37}$$

E.2 Single-distribution information theoretic measures

For a distribution p, parameterized as unnormalized logits, we can reuse the same LWS-monoid, but with scalar μ:

$$\mathrm{stat}(p) = \bigodot_i \{z : p_i,\ \mu : p_i\} \tag{38}$$

For $g = \mathrm{stat}(p)$,

$$\begin{aligned} g_z &= \ln \sum_i \exp(p_i) \\ g_\mu &= \sum_i \exp(p_i - g_z) \cdot p_i \\ H(p) &= g_z - g_\mu \end{aligned}$$

E.3 Information Theoretic Measures on Two Distributions

For two distributions p and q, parameterized as unnormalized logits, we use what is essentially the product of two LWS-monoids with $\odot$ applied independently on the two parts:

$$\begin{Bmatrix} \mathbf{p} : LWS(\{\mu \in \mathbb{R},\, \sigma \in \mathbb{R}\}) \\ \mathbf{q} : LWS(\{\mu \in \mathbb{R},\, \sigma \in \mathbb{R}\}) \end{Bmatrix} \cong LWS\left(\mathbb{R}^2\right) \times LWS\left(\mathbb{R}^2\right) \tag{39}$$

Then, for

$$\mathrm{stat}(p, q) = \bigodot_i \{\mathbf{p} : \{z : p_i,\, \mu : p_i,\, \sigma : q_i\},\, \mathbf{q} : \{z : q_i,\, \mu : q_i,\, \sigma : p_i\}\} \tag{40}$$

we can compute the KL-divergence, Cross Entropy, and Jeffreys divergence given $g = \mathrm{stat}(p, q)$:

$$\begin{aligned} \mathrm{KL}(p\|q) &= g_{\mathbf{p}_\mu} - g_{\mathbf{p}_z} - g_{\mathbf{p}_\sigma} + g_{\mathbf{q}_z} \\ \mathrm{CE}(p\|q) &= H(p) + \mathrm{KL}(p\|q) \\ &= g_{\mathbf{q}_z} - g_{\mathbf{p}_\sigma} \\ \mathrm{Jeffreys}(p, q) &= g_{\mathbf{p}_\mu} - g_{\mathbf{p}_\sigma} + g_{\mathbf{q}_\mu} - g_{\mathbf{q}_\sigma} \end{aligned}$$

Which can be shrunk by removing unused terms, i.e. when computing cross-entropy, we only need $\{\mathbf{p} : LWS(\sigma),\, \mathbf{q} : LSE\}$, where LSE is the LogSumExp monoid.

Since the monoid is a simple product of two LWS-monoids, the finalize-embed factorization is simply the two parallel finalize-embed factorizations corresponding to the LWS-monoids of $\mathbf{p}$ and $\mathbf{q}$.

E.4 One-hot Cross-Entropy

For a distribution p, parameterized as unnormalized logits, and q a one-hot distribution $q_i = \delta_{it}$, we have

$$\mathrm{CE}(q\|p) = \ln\left(\sum_i \exp(p_i)\right) - p_t$$

As such, the carrier needs only the log-partition of p and the logit p_t:

$$\begin{aligned} S &= \{z \in LSE,\, \ell \in \mathbb{R}\} \cong LSE \times (\mathbb{R}, +) \\ a \odot b &= \{z : a_z \odot_{LSE} b_z,\, \ell : a_\ell + b_\ell\} \end{aligned}$$

Then with:

$$\mathrm{stat}(p, t) = \bigodot_i \{z : p_i,\, \ell : \delta_{it} p_i\} \tag{41}$$

we have that $\mathrm{CE}(t\|p) = g_z - g_\ell$ for $g = \mathrm{stat}(p, t)$.

Since the monoid is a simple product of an LSE monoid and an additive one, the finalize-embed factorization is simply the two parallel finalize-embed factorizations of the two monoids.

E.5 AFFINE TRANSFORMS

An affine transform $f(x) = f_a \cdot x + f_b$ is a noncommutative monoid under composition and, as commonly used in sequence models, reverse composition: $g(f(x)) = (g \circ f)(x) = g_a(f_a \cdot x + f_b) + g_b$, where $f \odot g = g \circ f$. This gives the carrier $\{a \in \mathbb{R}_{\neq 0},\, b \in \mathbb{R}^D\}$, where $(f \odot g) = \{a : g_a f_a,\, b : g_a \cdot f_b + g_b\}$, which transports to $GL_{D+1}(\mathbb{R})$.

With

$$D(g, p, l)[\Delta] = \frac{g_a}{p_a \cdot l_a} \cdot \{a : p_a \cdot \Delta_a + \langle p_b,\, \Delta_b \rangle,\, b : \Delta_b\} \tag{42}$$

The direct, additive, finalize-embed factorization then takes the form:

$$\begin{aligned} \text{embed}(g)[\Delta] &= g_a \cdot \{a : \Delta_a,\, b : \Delta_b\} \\ \text{finalize}(p, l)[\eta] &= \frac{1}{p_a \cdot l_a} \cdot \{a : p_a \cdot \eta_a + \langle p_b,\, \eta_b \rangle,\, b : \eta_b\} \end{aligned}$$

Example: Monoid discovery by inverse transport Proposition 3 is not only a method to prove that a given monoid S satisfies the LGP, it can also be used in the other direction: Given T, we can define a new monoid S by specifying ϕ and ψ. As an example, we can construct an Affine function with log-space scaling and LWS-style addition:

$$\begin{aligned} \phi(a) &= \{a : \exp(a_\alpha),\, b : \phi_{LWS}(a_\beta)\} \\ \psi(x) &= \{\alpha : \log(x_a),\, \beta : \psi_{LWS}(x_b)\} \end{aligned}$$

The binary operation for this monoid is then:

$$\begin{aligned} a \odot b &= \psi(\phi(a) \odot_{Affine} \phi(b)) \\ &= \begin{Bmatrix} \alpha : \log(\exp(a_\alpha) \cdot \exp(b_\alpha)) \\ \beta : \psi_{LWS}(\exp(b_\alpha) \cdot \phi_{LWS}(a_\beta) + \phi_{LWS}(b_\beta)) \end{Bmatrix} \\ &= \begin{Bmatrix} \alpha : a_\alpha + b_\alpha \\ \beta : (b_\alpha \star a_\beta) \odot_{LWS} b_\beta \end{Bmatrix} \\ &\text{where } s \star a = \{z : a_z + s,\, \mu : a_\mu\} \end{aligned}$$

The result is a monoid where the scaling factor a is in logspace, and acts on an LWS monoid by *reweighting* the payload, rather than rescaling it. A consequence of this is that, like in attention, the output payloads (β_μ) are convex combinations of the input payloads.

The finalize-embed factorization given by direct transport (to the affine case) is:

$$\begin{aligned} \text{embed}(g)[\Delta] &= \begin{Bmatrix} a : \Delta_\alpha \\ b : \exp(g_\alpha - g_{\beta_z}) \cdot \begin{Bmatrix} w : \Delta_{\beta_z} - \langle \Delta_{\beta_\mu}, g_{\beta_\mu} \rangle \\ s : \Delta_{\beta_\mu} \end{Bmatrix} \end{Bmatrix} \\ \text{finalize}(p, l)[\eta] &= \begin{Bmatrix} \alpha : \eta_a + \exp(p_{\beta_z} - p_\alpha) \cdot \left(\eta_{b_w} + \langle \eta_{b_s}, p_{\beta_\mu} \rangle\right) \\ \beta : \exp(l_{\beta_z} - p_\alpha - l_\alpha) \cdot \begin{Bmatrix} z : \eta_{b_w} + \langle \eta_{b_s}, l_{\beta_\mu} \rangle \\ \mu : \eta_{b_s} \end{Bmatrix} \end{Bmatrix} \end{aligned}$$

Which due to the raw exponents is unstable, it can, however, be refined to a stable carrier by substituting b with an LWS-carrier (which in turn can use a corresponding execution carrier for LWS):

$$A \coloneqq \left\{\tau \in \mathbb{R},\, e \in LWS(\left\{w : \mathbb{R},\, s : \mathbb{R}^D\right\})\right\} \tag{43}$$

$$\cong (\mathbb{R}, +) \times LWS(\left\{w : \mathbb{R},\, s : \mathbb{R}^D\right\}) \tag{44}$$

$$a \odot_A b = \{\tau : a_\tau + b_\tau,\, e : a_e \odot_{LWS} b_e\} \tag{45}$$

$$\mathrm{embed}(g)[\Delta] = \left\{\begin{array}{l} \tau : \Delta_\alpha \\ e : \left\{\begin{array}{l} z : g_\alpha - g_{\beta_z} \\ w : \Delta_{\beta_z} - \langle \Delta_{\beta_\mu}, g_{\beta_\mu}\rangle \\ s : \Delta_{\beta_\mu} \end{array}\right. \end{array}\right\} \tag{46}$$

$$\mathrm{finalize}(p, l)[\eta] = \left\{\begin{array}{l} \alpha : \eta_\tau + \lambda \cdot \left(\eta_{e_w} + \langle \eta_{e_s}, p_{\beta_\mu}\rangle\right) \\ \beta : \gamma \cdot \left\{\begin{array}{l} z : \eta_{e_w} + \langle \eta_{e_s}, l_{\beta_\mu}\rangle \\ \mu : \eta_{e_s} \end{array}\right. \end{array}\right\} \tag{47}$$

$$\text{where} \quad \begin{array}{l} \lambda = \exp(\eta_{e_z} - p_\alpha + p_{\beta_z}) \\ \gamma = \exp(\eta_{e_z} - p_\alpha - l_\alpha + l_{\beta_z}) \end{array} \tag{48}$$

In the case where the observed output is restricted to the internal LWS payload (β_μ), Δ_α and Δ_{β_z} are both zero, giving the simplified formulation:

$$A \coloneqq LWS(\left\{w \in \mathbb{R},\, s \in \mathbb{R}^D\right\}) \tag{49}$$

$$\mathrm{embed}(g)[\Delta] = \left\{\begin{array}{l} z : g_\alpha - g_{\beta_z} \\ w : -\langle \Delta_{\beta_\mu}, g_{\beta_\mu}\rangle \\ s : \Delta_{\beta_\mu} \end{array}\right\} \tag{50}$$

$$\mathrm{finalize}(p, l)[\eta] = \left\{\begin{array}{l} \alpha : \lambda \cdot \left(\eta_w + \langle \eta_s, p_{\beta_\mu}\rangle\right) \\ \beta : \gamma \cdot \left\{\begin{array}{l} z : \eta_w + \langle \eta_s, l_{\beta_\mu}\rangle \\ \mu : \eta_s \end{array}\right. \end{array}\right\} \tag{51}$$

$$\text{where} \quad \begin{array}{l} \lambda = \exp(\eta_z - p_\alpha + p_{\beta_z}) \\ \gamma = \exp(\eta_z - p_\alpha - l_\alpha + l_{\beta_z}) \end{array} \tag{52}$$

F Evaluation details

F.1 CrossEntropy

For Cross-entropy, the CuTileReduce implementation is doing the equivalent of the below, using the monoid discussed in E.4:

$$\mathrm{CrossEntropy} : \mathbb{R}^{L\times D} \times \mathbb{N}^L \times \mathbb{R}^{R\times D} \to \mathbb{R}^L \tag{53}$$

$$\mathrm{CrossEntropy}(\mathbf{h}, \mathbf{t}, \mathbf{y})_l = \mathbf{g}_{lz} - \mathbf{g}_{l\ell}, \quad \mathbf{g}_l = \bigodot_r \{z : \langle \mathbf{h}_l,\, \mathbf{y}_r\rangle,\, \ell : z \cdot \mathbb{I}(\mathbf{t}_l = j)\} \tag{54}$$

Where $N = L = R$, and $D = 128$.

F.2 Attention

For Attention, the CuTileReduce implementation is doing the equivalent of the below, using the LWS-monoid:

$$\mathrm{Attention} : \mathbb{R}^{H\times L\times G\times D_l} \times \mathbb{R}^{H\times R\times D_l} \times \mathbb{R}^{H\times R\times D_v} \to \mathbb{R}^{H\times L\times G\times D_v} \tag{55}$$

$$\mathrm{Attention}(\mathbf{q}, \mathbf{k}, \mathbf{v})_{hlg} = \mathbf{g}_{hlg\mu}, \quad \mathbf{g}_{hlg} = \bigodot_r \{z : \langle \mathbf{q}_{hlg},\, \mathbf{k}_{hr}\rangle,\, \mu : \mathbf{v}_{hr})\} \tag{56}$$

Where $N = L = R$, $H = 8$, $G = 4$, and $D_l = D_v = 128$.

F.3 Affine Attention

For Attention, the CuTileReduce implementation is doing the equivalent of the below, using the monoid discussed in E.5:

$$\text{AffineAttention} : \mathbb{R}^{H\times L\times G\times D_l} \times \mathbb{R}^{H\times R\times D_l} \times \mathbb{R}^{H\times L\times G\times D_b} \times \mathbb{R}^{H\times R\times D_b} \times \mathbb{R}^{H\times R\times D_v} \to \mathbb{R}^{H\times L\times G\times D_v} \tag{57}$$

$$\text{AffineAttention}(\mathbf{q}^l, \mathbf{k}^l, \mathbf{q}^b, \mathbf{k}^b, \mathbf{v})_{hlg} = \mathbf{g}_{hlg\mu}, \quad \mathbf{g}_{hlg} = \bigodot_r \left\{\alpha : \langle \mathbf{q}^b_{hlg}, \mathbf{k}^b_{hr}\rangle, \beta : \left\{z : \langle \mathbf{q}^l_{hlg}, \mathbf{k}^l_{hr}\rangle, \mu : \mathbf{v}_{hr}\right\}\right\} \tag{58}$$

Where $N = L = R, H = 4, G = 2, D_b = 32$, and $D_l = D_v = 64$.

G Abstract implementation sketch

In Figure 4 we present the map-fold function over commutative and non-commutative monoids in its most general form: Without fused functions or tiling. In Figure 5 we present map-scan-fold function over commutative monoids in the same manner.

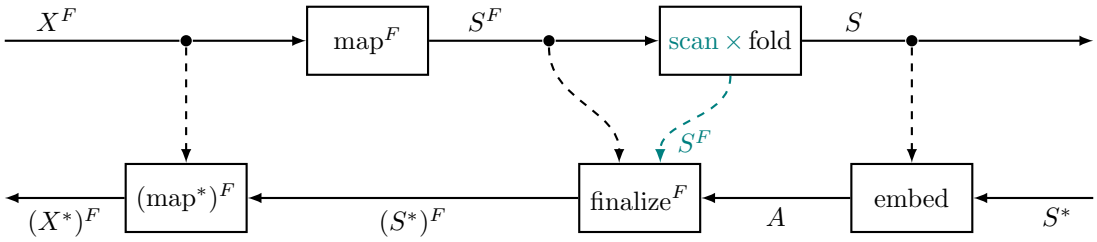


Figure 4: General forward and backward passes of map-fold functions over monoids. The colored parts, i.e. the exclusive prefix scan fed into finalize, is only required for non-commutative monoids.

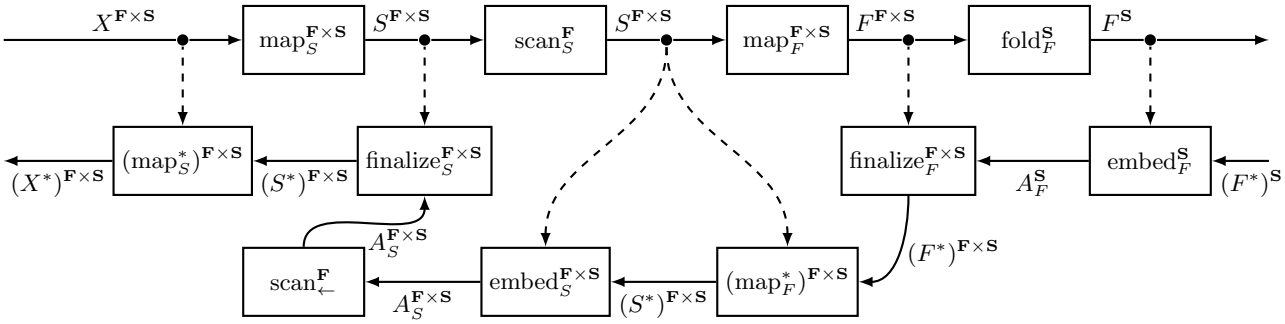


Figure 5: General forward and backward passes of map scan fold functions over commutative monoids (S and F), with S being scanned over axis $\mathbf{S}$ and F being constructed from scan state S, and folded over axis $\mathbf{F}$.

H Implementation Details: CuTileReduce

CuTileReduce implements the fold interface described in Sections 5, and 8 as a small kernel-generation layer on top of CuTile. The implementation does not infer monoids or derive tile-local gradient code automatically: users provide the algebraic callbacks, while CuTileReduce constructs the staged forward and backward kernels, searches over legal schedules, and builds a PyTorch-callable operator.

H.1 Fold Specification

A fold specification declares symbolic axes, input buffers, execution-carrier buffers, semantic buffers, and intermediate tile-resident values. Each axis is classified as a batch axis, fold axis, or tile-resident inner axis. The current prototype supports one fold axis; all other non-batch axes must fit within a single program tile.

The specification also records which input buffers require gradients and gives simple work and residency models used by the analytical scheduler.

H.2 User Callbacks

The user supplies the following callbacks:

$$\text{map-fold}, \quad \text{combine}, \quad \text{to-semantic}, \quad \text{to-output}, \quad \text{embed}, \quad \text{map-finalize}\,.$$

The forward callbacks map input tiles to execution-carrier states and combine partial states. The backward callbacks embed semantic output cotangents into a compact accumulator and reconstruct tile-local input gradients from recomputed local factors.

For commutative folds, map-finalize consumes input tiles, gradient accumulators, and the embedded cotangent. For general folds, it additionally consumes and returns the ordered prefix state.

H.3 Generated Forward Stages

CuTileReduce lowers a fold into one or two forward stages. The first stage tiles the logical interaction space and runs a per-program map-fold loop, producing carrier summaries $E^{B\times P}$. If $P = 1$, this is already the complete fold. Otherwise, a second fold stage combines the summaries into the semantic output S^B. In the general case, this second stage is an ordered exclusive scan plus fold, producing prefix boundaries $E^{B\times P}$ and the final semantic state.

H.4 Generated Backward Stages

The backward pass first applies embed to the saved semantic state and its cotangent, producing A^B. It then launches tiled recomputation kernels which reads the original inputs and gradient accumulator state (A) and apply map-finalize: this recomputes local factors, combines them with the gradient accumulator to produce local factor cotangents, pulls those back to input cotangents, which are then scattered or accumulated into the input-gradient buffers.

For commutative folds, each tile can be finalized using only the local factor and A^B. For general folds, each program also receives its exclusive prefix boundary (persisted from the forward pass) and threads an intra-program prefix through the recomputation loop.

H.5 Gradient Accumulation

CuTileReduce determines whether each gradient buffer depends on the loop axis. If it does not, contributions are accumulated in program-local storage and written once at program end. If it does, contributions are written during the loop. Writes use ordinary stores when ownership is unique and atomic adds when multiple programs may contribute to the same element.

H.6 Schedule Search

Given problem sizes and a hardware description, CuTileReduce enumerates legal tile sizes, program counts along the fold axis, loop axes for backward recomputation, and contention-group choices. Schedules are rejected when they exceed tile-resident capacity, violate axis constraints, require unsupported decompositions, or exceed a user specified limit on auxiliary storage.

The remaining schedules are ranked by a roofline-style analytical model using user-provided device-properties as well as user-provided work and residency estimates. The top-ranked candidates are then built and empirically timed; the fastest valid candidate is selected for each stage.

H.7 Plan Representation and Reuse

A selected plan records axis sizes, tile sizes, program counts, loop choices, stage kinds, algebra kind, and CuTile/version metadata. Plans can be saved and loaded to avoid repeating the analytical and empirical sweep. The plan stores scheduling decisions rather than generated machine code; CuTile handles its own kernel cache.

H.8 Current Limitations

The prototype currently supports a single fold axis, requires non-batch non-fold axes to be tile-resident, and relies on user-written CuTile callbacks. It does not yet infer algebraic structure, synthesize map-finalize or map-fold, support first-class scan-fold APIs, or perform distributed folds. The perhaps strongest limitation is due to the callback-based interface itself: While forcing fusion in the callbacks minimizes memory, it also makes it impossible for the framework to reason about alternative optimizations.

H.9 CrossEntropy Implementation

The user facing interface consists of supplying a general specification, declaring involved buffers, axes and their roles, simple work and residency models, and whether the monoid is commutative or not.

```
import math

import cuda.tile as ct
import torch

from cutilereduce.core.buffer import buffer_spec
from cutilereduce.fold import (
    fold_functions,
    make_fold_spec,
    matmul,
    workmodel,
)


LOG2E = math.log2(math.e)
LN2 = math.log(2)


spec = make_fold_spec(
    input={
        "ctx": buffer_spec("b d", ct.bfloat16, req_grad=True, default=0),
        "trg": buffer_spec("v d", ct.bfloat16, req_grad=True, default=0),
        "targets": buffer_spec("b", ct.int32, default=-100),
    },
    execution={
        "m": buffer_spec("b", ct.float32, default=float("-inf")),
        "e": buffer_spec("b", ct.float32, default=0),
        "u": buffer_spec("b", ct.float32, default=0),
    },
    semantic={
        "z": buffer_spec("b", ct.float32, default=float("-inf")),
        "l": buffer_spec("b", ct.float32, default=0),
    },
    embedded={
        "z": buffer_spec("b", ct.float32, default=0),
        "g_z": buffer_spec("b", ct.float32, default=0),
        "g_l": buffer_spec("b", ct.float32, default=0),
    },
    map_fold_intermediate={
        "logits": buffer_spec("b v", ct.float32),
```

```
    },
    map_finalize_intermediate={
        "logits": buffer_spec("b v", ct.float32),
    },
    batch="b",
    fold="v",
    map_fold_work=workmodel(matmul(M="b", N="v", K="d")),
    map_finalize_work=workmodel(
        matmul(M="b", N="v", K="d"),
        matmul(M="b", N="d", K="v"),
        matmul(M="v", N="d", K="b"),
    ),
    algebra=AlgebraKind.commutative,
)


@ct.function
def xentropy_map(tid, ctx, trg, targets):
    indices = tid.indices("v")
    mask = tid.mask("v")
    batch, fold = tid.shape("b", "v")

    logits = ct.zeros((batch, fold), ct.float32)
    logits = ct.mma(ctx, trg.transpose(), logits) * LOG2E
    logits = ct.where(mask[None, :], logits, float("-inf"))
    hits = (targets[:, None] == indices[None, :]) & mask[None, :]
    return logits, hits


@ct.function
def map_fold(tid, ctx, trg, targets):
    logits, hits = xentropy_map(tid, ctx, trg, targets)
    maximum = ct.max(logits, axis=1)
    exponential_sum = ct.sum(ct.exp2(logits - maximum[:, None]), axis=1)
    target_logit = ct.sum(ct.where(hits, logits, 0.0), axis=1)
    return maximum, exponential_sum, target_logit


@ct.function
def combine(am, ae, av, bm, be, bv):
    left_is_high = am > bm
    high_m = ct.where(left_is_high, am, bm)
    high_e = ct.where(left_is_high, ae, be)
    low_m = ct.where(left_is_high, bm, am)
    low_e = ct.where(left_is_high, be, ae)
    skip = high_m == float("-inf")
    scaling = ct.exp2(low_m - high_m)
    return high_m, ct.where(skip, high_e, high_e + low_e * scaling), av + bv


@ct.function
def to_semantic(maximum, exponential_sum, target_logit):
    return (
        (maximum + ct.log2(exponential_sum)) * LN2,
        target_logit * LN2,
    )


def to_output(logsumexp, target_logit):
```

```
    return logsumexp - target_logit


@ct.function
def embed(logsumexp, target_logit, g_logsumexp, g_target_logit):
    return logsumexp * LOG2E, g_logsumexp, g_target_logit


@ct.function
def map_finalize(tid, ctx, trg, targets, g_ctx, g_trg, z, g_z, g_l):
    logits, hits = xentropy_map(tid, ctx, trg, targets)
    scale = ct.exp2(logits - z[:, None])
    g_logits = (scale * g_z[:, None] + hits * g_l[:, None]).astype(ct.bfloat16)
    g_ctx = ct.mma(g_logits, trg, g_ctx)
    g_trg = ct.mma(g_logits.transpose(), ctx, g_trg)
    return g_ctx, g_trg


operator = FoldOperator(
    spec,
    fold_functions(
        map_fold=map_fold,
        combine=combine,
        to_semantic=to_semantic,
        to_output=to_output,
        embed=embed,
        map_finalize=map_finalize,
    )
)
# tuning the operator for a specific problem size and device
sizes = {'b': 16384, 'v': 16384, 'd': 128}
hw = cutilereduce.util.spec.SPECMAP['rtx5080']
plan = operator.tune(sizes, hardware=hw)
# Building the actual torch function
ce_fun = operator.build(plan)
...
ce_loss = ce_fun(...)
ce_loss.mean().backward() # with a fused map-finalize backward pass!
```