LAPMERGE: DIFFERENTIABLE EXACT-BUDGET TOKEN MERGING IN LINEAR TIME

Anonymous authors
Paper under double-blind review

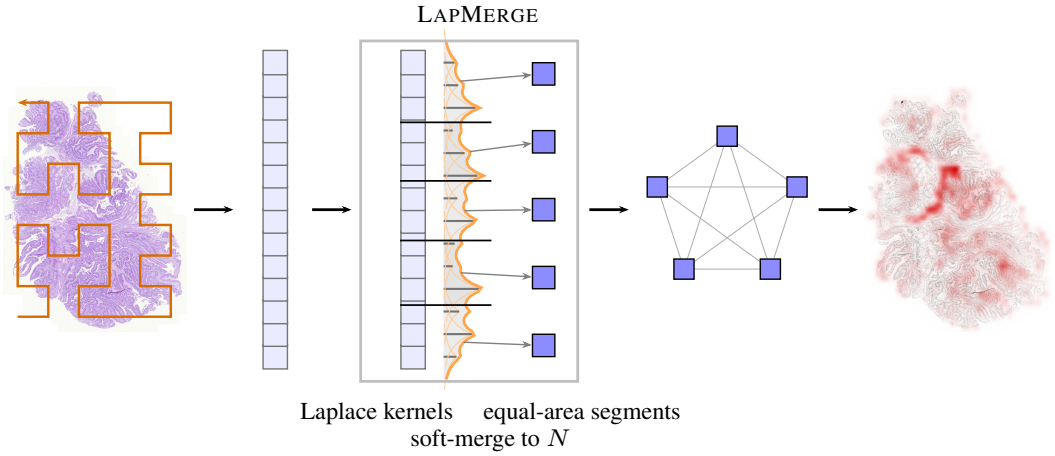

Figure 1: **Overview of the standard LapMerge pipeline.** Given an ordered sequence of M input tokens, LapMerge produces exactly N budget-exact, soft-merged output tokens. A learned scorer assigns each input token a mass, represented as a Laplace kernel. The summed density is scanned cumulatively and partitioned into N equal-mass segments, each of which is collapsed into a single output token. The resulting N tokens are then processed by a Transformer classifier. The rightmost panel shows a real histopathology slide with the learned scorer weights overlaid on the tissue; opacity indicates the importance attributed to each patch. These same scores determine the token merging and can therefore be read as a learned routing-mass map: a direct view of where the model allocates its token budget.

ABSTRACT

Modern vision and language models process ever longer token sequences, and reducing them is the main lever on downstream cost. Token compression, however, faces a fundamental trade-off: selecting tokens discards information, while existing merging methods preserve it only through pairwise token-to-token matching or MN token-to-output interactions. We introduce LapMerge, a differentiable merging operator that resolves this trade-off for ordered sequences: a learned mass distribution partitions the sequence into exactly N equal-mass segments under a Laplace-kernel density, and prefix/suffix scans aggregate each segment, so both the forward and the backward pass run in $\mathcal{O}(M + N)$ time. This combination – an exact output budget, end-to-end differentiability, and linear cost – is precisely what previous reducers trade off against one another. The budget can be changed without changing operator parameters, and the learned mass doubles as a token-level map of where the budget is allocated. Across vision and vision-language tasks, LapMerge matches or improves on strong token reducers while being up to two orders of magnitude faster at large sequence lengths; it is particularly effective when token order reflects spatial or temporal locality, and works both as a drop-in reduction module and as a trainable architectural component.

1 INTRODUCTION

Modern vision, language, and multimodal encoders increasingly operate on long token sequences. High-resolution images yield hundreds of patches, documents can contain thousands of tokens, and

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.

videos produce long spatiotemporal traces. While efficient attention implementations reduce the memory-access overhead of Transformer computation (Dao et al., 2022), they do not change the fundamental dependence of computation on sequence length. With a frozen encoder, the key efficiency problem is to reduce the number of tokens processed while preserving the information the downstream task needs.

A large body of work addresses this problem through token reduction. Existing approaches predominantly follow one of two strategies: pruning or merging. Pruning estimates the importance of individual tokens and retains only a subset, discarding the remainder (Rao et al., 2021; Liang et al., 2022; Fayyaz et al., 2022). This makes pruning computationally attractive: token scores are computed in a single pass, and selecting the top-N tokens requires no interaction between input and output tokens. Its limitation is information loss – pruning suits settings where the relevant evidence is concentrated in a small number of tokens, but can be brittle when the prediction depends on multiple weak cues distributed across the input.

Merging takes the opposite approach, aggregating tokens into a smaller set of representations rather than discarding them (Ryoo et al., 2021; Renggli et al., 2022; Bolya et al., 2023; Tran et al., 2024). This can preserve information distributed across many tokens. However, existing learned merging mechanisms generally require computing token-to-token or token-to-output interactions. For example, PatchMerger (Renggli et al., 2022) and TokenLearner (Ryoo et al., 2021) construct learned aggregation weights over the input tokens, while Perceiver (Jaegle et al., 2021)-style architectures use cross-attention between a fixed set of latent queries and the input sequence. Such global aggregation explicitly evaluates MN token-to-output interactions. When the output budget is a fixed fraction of the input, $N = \rho M$, this becomes quadratic in M; even when N is fixed, the method still carries an MN interaction cost.

This raises a natural question: can token reduction preserve information from all M inputs while producing exactly N outputs without an explicit $M \times N$ interaction? We answer this by exploiting sequence order. LapMerge learns a scalar mass at each position and uses its cumulative distribution to partition the sequence into N contiguous, equal-mass regions, each producing one output token. This structured assignment replaces unrestricted token-to-output interactions with prefix/suffix scans, yielding $\mathcal{O}(M + N)$ cost in both forward and backward passes for given masses and ordered positions; producing the masses with the soft top-k router used in our experiments (Struski et al., 2025) adds an $O(M \log M)$ sort, which Fast LapSum (Struski et al., 2026) makes expected-linear and pointwise sigmoid or softplus routing avoids entirely. The resulting outputs are convex combinations of their inputs, with a log-space implementation ensuring numerical stability at long sequence lengths. Figure 1 gives an overview of the method.

The key inductive bias is locality: nearby positions are preferentially aggregated together. This makes LapMerge particularly suited to spatially or temporally ordered tokens, while offering less advantage when sequence order is unrelated to semantic structure. In practice, the operator is implemented in native PyTorch, optionally accelerated with CUDA, and scales to sequences of more than 10^7 tokens. Our main contributions:

- We introduce LapMerge, a token merging operator that combines three properties prior reducers trade off against one another: an exact output budget of N tokens, end-to-end differentiability with closed-form gradients, and $\mathcal{O}(M + N)$ cost in both passes.
- We show that LapMerge is versatile: a training-free drop-in that improves a released vision-language pipeline, and a trainable component in from-scratch and frozen-backbone models.
- Across vision and vision-language benchmarks, LapMerge matches or improves on strong token reducers at equal budgets while scaling beyond 10^7-token inputs, up to two orders of magnitude faster than matched SDPA poolers.

2 Related Work

Prior work addresses the cost of long token sequences along three main axes: selecting a subset of tokens to retain, aggregating multiple tokens into fewer representations, or modifying the sequence model to process long inputs more efficiently.

Token pruning. Token pruning reduces computation by identifying and discarding tokens that are unlikely to contribute to the downstream task. DynamicViT (Rao et al., 2021), EViT (Liang et al., 2022), ATS (Fayyaz et al., 2022), and DiffRate (Chen et al., 2023) learn token importance and retain a subset of the input. More recent methods consider the frozen-backbone setting, including VisionSelector (Zhu et al., 2025), AutoSelect (He et al., 2026b), and DiffPrune (He et al., 2026a). These approaches are closely related to ours: they learn a lightweight reduction policy under a fixed token budget, but use it to select tokens rather than aggregate them. Differentiable top-k selection itself has been studied extensively through continuous relaxations and differentiable sorting (Grover et al., 2019; Prillo & Eisenschlos, 2020; Blondel et al., 2020; Cordonnier et al., 2021). LapMerge instead uses the differentiable budget assignment of LapSum (Struski et al., 2025) to determine how the output budget is distributed across an ordered sequence.

Token merging and pooling. Token merging provides an alternative to pruning by combining multiple input tokens into a smaller set of representations. ToMe (Bolya et al., 2023) uses bipartite matching to merge similar tokens, while PiToMe (Tran et al., 2024) extends this approach with improved matching. Other approaches perform learned or structured pooling. Token Pooling (Marin et al., 2023) uses clustering, TokenLearner (Ryoo et al., 2021) and PatchMerger (Renggli et al., 2022) learn attention-based pooling, and Perceiver (Jaegle et al., 2021) and Flamingo (Alayrac et al., 2022) use cross-attention from a fixed set of latent representations. Set Transformer (Lee et al., 2019) similarly introduces learned inducing points for scalable set aggregation.

Efficient sequence processing. A complementary line of work reduces the cost of processing long sequences by modifying the sequence model rather than reducing its input. Sparse and linear attention (Child et al., 2019; Katharopoulos et al., 2020; Choromanski et al., 2021) restrict or approximate token interactions, while Longformer (Beltagy et al., 2020) introduces sparse attention patterns. Funnel Transformer (Dai et al., 2020) progressively reduces sequence length within the model, and state-space models such as S4 (Gu et al., 2022) and Mamba (Gu & Dao, 2023) provide alternative mechanisms for processing long sequences. These approaches are orthogonal to token reduction: LapMerge can be applied before the sequence model and combined with different architectures.

Existing work primarily explores either which tokens to discard or how to aggregate tokens. LapMerge instead asks how the structure already present in an ordered token stream can be used to make aggregation both efficient and adaptive. This places the method between pruning and merging, preserving distributed evidence through aggregation while exploiting order to avoid a global assignment.

3 LapMerge: Laplace Token Merging

This section gives the main construction and explains why the Laplace kernel makes exact-budget merging linear in sequence length. We keep the derivations brief here; Appendix A gives the complete construction, proofs, scan algorithms, and backward formulas.

Throughout the paper, $LapMerge$ denotes the standard equal-mass barrier operator. Its input positions are assumed to arrive in the order in which they are to be processed; for an ordinary sequence this is simply the original token order, and no additional sorting is performed by the operator. The deployment variant used in POPE – the same operator driven by an external scorer, with output positions taken from hard selection – is described separately as *score-routed LapMerge*.

A natural approach to merging, and why it is not differentiable. Suppose a lightweight scoring head has already told us how important every token is. Let $(x_i)_{i=1}^{M}$, $x_i \in \mathbb{R}^d$, denote the input tokens, $p_1 < \cdots < p_M$ their positions in the fixed scan order, and $r_i \geq 0$ its importance weight. A natural idea suggests itself: accumulate these weights r_i in scan order, cut the sequence wherever the running sum crosses a fixed increment, and average the tokens inside every one of the resulting batches into a single output token – the pipeline sketched in Figure 2, top. Do this so that the running sum crosses exactly N increments and we get exactly N output tokens, and we have almost the object we want – an adaptive partition that spends more of a fixed output budget on high-scoring regions, computed without ever comparing every input token to every output token (the same cumulative-score sweep already selects tokens for ATS (Fayyaz et al., 2022), keeping one nearest token per threshold instead of averaging a segment). As stated, though, it is not easy to use and not differentiable: the

cut points are level sets of a step function – the running sum jumps at every token position p_i and is flat in between, so an infinitesimal change to any one score can move a boundary discontinuously past a neighboring token, and the gradient of everything downstream with respect to the scores is zero almost everywhere and undefined exactly at the jumps. The bigger problem hides one step earlier: we wanted to *learn* the scoring head end to end, and a scoring head sitting behind a dead gradient receives no training signal at all – no amount of averaging inside the batches fixes that, so the scorer itself would never move.

Softening the problem. One repairs this by replacing every token's point mass with uncertainty about its position: a fixed location becomes a random position with a continuous distribution around it, as in kernel density estimation – the soft relaxation sketched in Figure 2, bottom, where each token's mass spreads into a smooth bump and the resulting mixture density is cut into intervals of exactly equal mass, rather than the top panel's discontinuous running-sum cuts. The same hard-to-soft idea can be found e.g. in speech-boundary segmentation (Hwang & Lee, 2024) where it is achieved via $\mathcal{O}(MN)$ attention with no well-posedness guarantee. We want a closed-form, provably well-posed version instead, with $\mathcal{O}(M+N)$ time cost. The rest of this section explains how to achieve that.

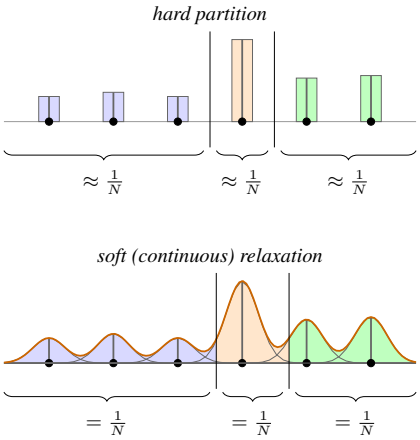

Figure 2: The same six tokens cut into $N = 3$ pieces two ways – by token *count* they are 3-1-2, but by *mass* all three pieces are comparable. *Top*: the hard partition from the running sum – barriers can only fall between tokens, so each piece's mass is merely *close* to $1/N$. *Bottom*: treating positions as noisy turns each token into a smooth bump; the shaded area under the resulting mixture density (bold) is *exactly* $1/N$ per piece (Proposition 1). These intervals partition probability mass, not tokens: one token's bump can span several intervals. The kernel shown is Gaussian, purely for illustration.

A formal construction. Let κ denote the density of this position noise – a continuous probability density on $\mathbb{R}$ – with CDF σ and scale $\alpha > 0$, and attach token x_i to a random position $P_i = p_i + \alpha E_i$ in place of the fixed p_i, with E_i drawn i.i.d. from the distribution with CDF σ. A smaller α concentrates mass near each p_i, whereas a larger α spreads each token's mass across more intervals. Draw an index I with $\Pr(I = i) = r_i$, where $r_i \geq 0$ and $\sum_{i=1}^{M} r_i = 1$ are the normalized token weights: the resulting random position P_I is then distributed with CDF

$$\Phi(x) = \sum_{i=1}^{M} r_i \, \sigma\left(\frac{x - p_i}{\alpha}\right), \tag{1}$$

a smooth, blurred version of the running sum from the earlier paragraphs. We cut $\mathbb{R}$ into N intervals $[b_{\ell-1}, b_\ell)$, $\ell = 1, \ldots, N$ (with $b_0 = -\infty$ and $b_N = +\infty$), each capturing probability exactly $1/N$ under Φ. The interior barriers satisfy

$$\Phi(b_\ell) = \ell/N, \qquad 1 \leq \ell < N. \tag{2}$$

The intervals partition the mass of P_I, not the set of input tokens; a token may contribute to more than one interval. The barriers exist uniquely and depend differentiably on the routing weights.

Proposition 1. *If κ is continuous and strictly positive, the boundaries $\Phi(b_\ell) = \ell/N$, $1 \leq \ell < N$, exist uniquely and the map $r \mapsto b_\ell$ is differentiable on the probability simplex. For any tangent $\dot{r}$ with $\sum_j \dot{r}_j = 0$, its derivative is $\dot{b}_\ell = \sum_j (\partial b_\ell / \partial r_j)\dot{r}_j$, where one representative of the partial derivatives is*

$$\frac{\partial b_\ell}{\partial r_j} = -\frac{\sigma\big((b_\ell - p_j)/\alpha\big)}{\Phi'(b_\ell)}. \tag{3}$$

Proof sketch. Since

$$\Phi'(x) = \frac{1}{\alpha} \sum_{i=1}^{M} r_i \, \kappa\left(\frac{x - p_i}{\alpha}\right) > 0,$$

Φ is continuously differentiable and strictly increasing, with limits 0 and 1 at $-\infty$ and $+\infty$. Each level ℓ/N therefore determines a unique boundary, and consecutive boundaries enclose mass $1/N$. The implicit function theorem gives differentiability of the boundaries. Differentiating

$\Phi(b_\ell(t); r(t)) = \ell/N$ gives the displayed formula. Appendix A.1 gives the complete proof and explains the simplex constraint. □

With the barriers in hand, define the merged tokens as the conditional expectation of the randomly drawn token given it lands in each interval:

Definition 1 (Merged tokens)**.**

$$t_\ell := \mathbb{E}[x_I \mid P_I \in [b_{\ell-1}, b_\ell)], \qquad \ell = 1, \ldots, N.$$

We can note that each t_ℓ is a convex combination of the input tokens – never an extrapolation outside their span. To be precise, we can obtain

$$w_i^\ell = N r_i \left[\sigma\left(\frac{b_\ell - p_i}{\alpha}\right) - \sigma\left(\frac{b_{\ell-1} - p_i}{\alpha}\right)\right], \qquad t_\ell = \sum_{i=1}^{M} w_i^\ell x_i, \tag{4}$$

Figure 3 illustrates a single interval: token i contributes its kernel's probability mass inside $[b_{\ell-1}, b_\ell)$, and may also contribute to other intervals. Summing equation 4 over ℓ telescopes the CDF differences to one, giving $N^{-1}\sum_\ell t_\ell = \sum_i r_i x_i$: the operator preserves the weighted first moment of the input. This is a conservation property, not information preservation – at $N < M$ the aggregation is not invertible, and a token with $r_i = 0$ contributes nothing.

Standard versus score-routed LapMerge. The construction above is the standard LapMerge operator with a single kernel width; for simplicity we present this base version. It also admits separate widths for routing (α_r, barriers) and read-out (α_a, with per-interval normalization), evaluated by the same scans in $\mathcal{O}((M+N)d)$; the general form, its properties and cost are in Appendix D.2. The POPE experiment uses this form with an external scorer: per-image standardized VisionSelector scores enter a LapSum soft top-k at temperature α_r, giving budget-exact masses that sum to k at the raster indices. The k outputs are injected at the rotary positions of the k highest-scoring tokens, so positions come from hard selection while values come from the merge; as both widths go to zero, the pipeline coincides with the released selection pipeline. We refer to this deployment as *score-routed LapMerge* below.

LapMerge: a fast, exact solution to merging. Everything above holds for any strictly positive and continuous κ, and we have not yet said how the weights r_i themselves are obtained from raw scores. Two problems remain, both computational:

- Under a generic kernel, no closed form exists for the boundaries b_ℓ: locating each of the $N-1$ barriers by bisecting the length-M mixture costs $\mathcal{O}(M)$ per barrier, $\mathcal{O}(MN)$ in total – reintroducing exactly the token-to-output interaction we set out to avoid.
- Even given the barriers, evaluating the conditional expectation t_ℓ (Definition 1) generically sums over all M tokens for each of the N outputs – again $\mathcal{O}(MN)$.

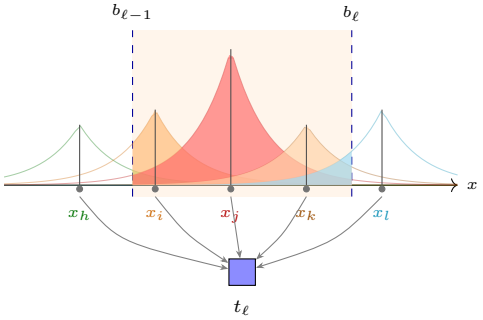

Figure 3: *Why the Laplace kernel is fast*: the tick at each kernel's peak splits it into a left half and a right half, and Equation 6 aggregates every left half with one prefix sum and every right half with one suffix sum – never a per-pair computation. Five tokens shown: three peaks fall inside $[b_{\ell-1}, b_\ell)$ (x_i, x_j, x_k) and two fall outside (x_h, x_l); only the shaded, interval-clipped part of each counts toward t_ℓ (Definition 1).

Both are solved by the same choice of kernel: the Laplace distribution, $\kappa(z) = \frac{1}{2}e^{-|z|}$, with CDF $\sigma(z) = \frac{1}{2}e^z$ for $z \le 0$ and $\sigma(z) = 1 - \frac{1}{2}e^{-z}$ for $z \ge 0$, which is strictly positive, as we need. Using and generalizing the prefix/suffix tricks that make LapSum (Struski et al., 2025) fast, both problems drop to $\mathcal{O}(M+N)$. A fused prefix/suffix scan provides further GPU acceleration.

The merging operator itself does not require raw scores or a particular scorer. It takes as input the ordered positions p_i and non-negative routing masses r_i satisfying $\sum_i r_i = 1$:

$$r_i \ge 0, \qquad \sum_{i=1}^{M} r_i = 1. \tag{5}$$

These masses may be supplied by a learned scorer, a differentiable soft-top-k routing module, or a fixed external policy. Such a routing choice is optional and upstream of LapMerge; the equations below are valid for any admissible r. Substituting the two branches of the Laplace CDF into the definition of Φ separates the tokens according to whether $p_i \le x$ or $p_i > x$. Factoring out the terms that depend on x turns Equation 1 into

$$\Phi(x) = \underbrace{\textstyle\sum_{p_i \le x} r_i}_{\text{prefix}} - \tfrac{1}{2} e^{-x/\alpha} \underbrace{\textstyle\sum_{p_i \le x} r_i e^{p_i/\alpha}}_{\text{prefix}} + \tfrac{1}{2} e^{x/\alpha} \underbrace{\textstyle\sum_{p_i > x} r_i e^{-p_i/\alpha}}_{\text{suffix}}. \tag{6}$$

Because the tokens are already sorted by p_i, all three underbraced sums are cumulative sums of the sequence (or its reverse in the suffix case), precomputed with one pass in each direction. The values $\Phi(p_i)$ are strictly increasing, as are the target levels ℓ/N; merging these two ordered lists locates all barrier cells in $O(M+N)$ time. Inside a cell the sums are constant, so $\Phi(x) = \ell/N$ reduces to one quadratic equation in $u = e^{x/\alpha}$, solved in closed form. We reuse the same prefix/suffix machinery not to define the scorer, but to invert the mixture induced by the supplied r_i and locate the b_ℓ themselves.

Aggregation, and the resulting cost. The merged tokens can be computed directly from the cumulative feature sums

$$\Phi_x(z) = \sum_{i=1}^{M} r_i x_i \sigma\left(\frac{z - p_i}{\alpha}\right), \qquad t_\ell = N\big[\Phi_x(b_\ell) - \Phi_x(b_{\ell-1})\big].$$

Applying equation 6 coordinatewise with $r_i x_i$ in place of r_i gives additional prefix/suffix tables of the same form. Signed feature coordinates are scanned as separate positive and negative parts (Appendix A). Reusing the previously identified boundary cells, we compute all merged tokens in $O((M+N)d)$ time without forming individual membership weights. The gradients needed for backpropagation have the same complexity, so linear scaling is preserved during training.

Theorem 1. *For the Laplace kernel and fixed ordered positions, the map $(r, x) \mapsto (b, t)$ and each of its Jacobian-vector and vector-Jacobian products can be evaluated in $O((M+N)d)$ work for d-dimensional tokens, or $O(M+N)$ when d is fixed.*

Proof sketch: Building the scalar prefix/suffix tables and locating all boundary cells costs $O(M+N)$. Evaluating the barriers costs $O(N)$; the feature-weighted scans and readout cost $O((M+N)d)$. By Equation equation 3, the boundary Jacobian-vector product is a Laplace-CDF sum weighted by $\dot{r}_i$. Its adjoint uses prefix/suffix scans over the ordered boundaries. Both products therefore have linear cost. For aggregation, differentiating $r_i x_i$ and the evaluations $\Phi_x(b_\ell)$ adds only further scans and pointwise operations. Their adjoints have the same linear cost; see Appendix A.2–A.3 for the full proof and the orientation of the reverse CDF scan. Thus all claimed computations take $O((M+N)d)$ time without forming dense membership matrices or Jacobians. □

The reverse barrier scan uses the CDF orientation in equation 3, and the aggregation readout is differenced only once. Appendix A.2–A.3 gives the exact formulas. The formal statement covers the (r, x) map at fixed ordered positions; the implementations additionally expose derivatives with respect to the positions p through the same scan structure (the runtime benchmark's backward pass includes them, so the reported timings already reflect this cost). The kernel width α is fixed in the LRA and POPE experiments and learned in histopathology, Cars, and arXiv (Table 5), where its gradient flows through the same native implementation; the formal statement above covers the (r, x) map at fixed positions and width.

At the lengths we operate at, the exponentials in Equation 6 must be accumulated in log-scale to avoid overflow; Appendix A provides the detailed pseudocodes and numerical implementation discussion. A fused CUDA kernel or, more simply, native PyTorch via `logcumsumexp` both implement this scan, at a modest constant-factor tradeoff; Appendix B.2 compares them.

4 Experiments and Results

We evaluate LapMerge in four settings, chosen to test the operator both where its ordering bias should help and where it should not. *(i) Computational scaling*: a pure runtime study on sequences

of up to 2^{24} ($> 10^7$) tokens, against learned poolers under identical timing conditions. *(ii) Gigapixel histopathology*: predicting microsatellite instability (MSI-H vs. MSS), a clinically actionable biomarker, on two TCGA whole-slide cohorts with a frozen pathology encoder, against specialized MIL baselines. *(iii) A frozen vision–language model*: we replace only the token-selection step of the released Qwen2.5-VL-7B VisionSelector pipeline, without any retraining, and measure object hallucination on POPE. *(iv) End-to-end training from scratch* on the five Long Range Arena tasks, against budget-matched reduction baselines. All methods are compared at matched output budgets; per-experiment configurations, training protocols, and dataset details are collected in Table 5 and Appendix E.

4.1 Computational Scaling

Figure 4 isolates the reducers, excluding upstream networks. We vary M from 2^{11} to 2^{24} at both a fixed output budget and a fixed 16-fold compression, with identical timing settings for all methods. At $M = 2^{20}$ and $N = M/16$, LapMerge takes 0.136 s versus 7.16 s for TokenLearner and 14.75 s for PatchMerger; the pure-PyTorch native path, which is the numerically faithful implementation used in all accuracy experiments, takes 0.290 s in the same setting – still 25–51× faster than the poolers, so the advantage does not rest on the fused backend. The ordered-coordinate LapMerge run reaches $M = 2^{24}$ at $N = 1024$ (1.88 s) and $M = 2^{23}$ at $N = M/16$ (1.07 s); the next ratio point is out of memory. Similar LapMerge slopes as N grows are consistent with scan-based $O((M+N)d)$ work, without an $M \times N$ assignment. Appendix B.3 separates dense, native, compiled, and fused implementations; Appendix B.4 gives the matched competitor measurements and failure diagnostics.

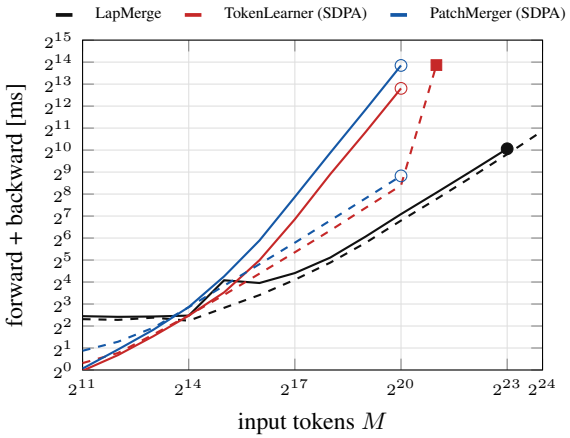

Figure 4: Reducer forward–backward latency on GH200 ($B = 1$, $d = 512$, FP32; five warm-ups, 30 repeats). Dashed: $N = 1024$; solid: $N = M/16$. LapMerge scales similarly in both regimes; the SDPA poolers retain MN interactions. Open circles precede a 600-s timeout, the filled OOM, the filled square an execution failure. Timing covers the reducer core (LapMerge gets precomputed masses; the poolers project inside the timed region); scorer/routing overheads: Appendix B.4.

4.2 LapMerge on Gigapixel Histopathology

We first evaluate LapMerge on whole-slide histopathology, a natural test bed for long-sequence modeling. A single whole-slide image (WSI) can contain thousands to tens of thousands of tissue patches, while only an unknown subset may be relevant to the diagnostic task. This creates a setting where reducing the sequence before applying expensive sequence modeling is particularly valuable. Figure 1 illustrates the complete pipeline on this task. We encode tissue patches with a frozen UNI2-h encoder (Chen et al., 2024b; Mahmood Lab, 2025), order them along a generalized Hilbert curve to preserve spatial locality, and apply LapMerge to reduce the resulting sequence from M patches to a fixed budget of $N = 1024$ tokens. The compressed sequence is then processed by a standard three-layer Transformer classifier. LapMerge is used without task-specific modification or budget-specific tuning. Because the same scores determine the mass distribution used for merging, the reduction mechanism simultaneously provides a patch-level routing-mass map, which passes a deletion-style attribution test (Appendix E.1).

We compare LapMerge with established and recent multiple-instance learning (MIL) methods, including ABMIL (Ilse et al., 2018), CLAM-SB/MB (Lu et al., 2021), DS-MIL (Li et al., 2021), DTFD-MIL (Zhang et al., 2022), TransMIL (Shao et al., 2021), MambaMIL (Yang et al., 2024), and RRT-MIL (Tang et al., 2024). All methods use the same frozen UNI2-h features and patient-level splits. Results are reported as mean ± std AUROC over six random seeds and are summarized in Table 1.

LapMerge achieves 91.77 ± 3.14 AUROC on CRC and 96.03 ± 1.99 on STAD, second-best on both cohorts. Only RRT-MIL is stronger (93.54 ± 3.03 and 97.19 ± 1.61), while every remaining baseline achieves lower mean AUROC on at least one of the two cohorts. Notably, LapMerge is the only method whose sequence model operates on at most $1{,}024$ tokens rather than the full bag (median $5{,}922$/$7{,}329$ patches on CRC/STAD; only 3.6%/8.3% of slides have $M \leq 1024$, where $N{=}1024$ is not a compression): it stays competitive at a fraction of the sequence length.

Table 1: Test AUROC (mean $\pm$ std, 6 seeds). External anchors are literature numbers, not reproduced here: Kather et al. 2019 is a CNN tile-classifier, not MIL; Wagner et al. 2023 is AttentionMIL over CTransPath features.

method	CRC	STAD
ABMIL	88.97±2.91	92.68±3.24
CLAM-SB	89.92±4.40	93.75±2.90
CLAM-MB	90.66±1.43	91.62±2.74
DS-MIL	90.33±7.60	87.11±3.35
DTFD-MIL	90.86±1.58	93.22±4.46
TransMIL	81.07±3.90	89.24±4.52
MambaMIL	87.98±5.62	93.02±5.63
RRT-MIL	**93.54±3.03**	**97.19±1.61**
LapMerge (ours)	91.77±3.14	96.03±1.99
Kather '19	*77.00*	*81.00*
Wagner '23	*78.00*	—

The computational advantage is complementary to this predictive performance. LapMerge reduces M input tokens to N output tokens in $\mathcal{O}(M{+}N)$ time, allowing the expensive sequence model to operate only on the compressed representation. In contrast to methods that retain or repeatedly process the full patch sequence, our reduction cost grows linearly with the number of input patches. Thus, this experiment serves as a proof of concept that LapMerge can act as a general-purpose front-end for long-sequence modeling.

4.3 Drop-in Replacement in a Frozen Vision-Language Model

We next evaluate whether score-routed LapMerge can improve visual token reduction in a frozen vision-language model (VLM). We take the publicly released VisionSelector pipeline built on Qwen2.5-VL-7B (Bai et al., 2025) as-is, including its vision encoder, its learned scorer, and the language model, and replace only the final selection step. VisionSelector keeps the k highest-scoring tokens and drops the remainder. Score-routed LapMerge keeps the same k positions but fills them with merged values: the standardized scores enter a soft top-k at temperature α_r, the resulting budget-exact masses define k equal-mass barrier intervals along the raster order, and each output is the mass-weighted average of all tokens under its interval's Laplace window of read-out width α_a (§3). Every output token stays at the position of the corresponding kept token, so the rotary position encoding of the language model is unchanged and no other part of the pipeline has to be adapted.

Table 2: **Drop-in token reduction on POPE** (F1, frozen Qwen2.5-VL-7B, no retraining) at matched 10% and 30% visual-token budgets. Baseline rows quote the published evaluation of Zhu et al. (2025) (their Table 1); our rerun of the released selection reproduces its F1 to 0.06 (Appendix D.3). LapMerge reuses the VisionSelector scorer at default widths $(\alpha_r, \alpha_a){=}(1, 1)$; the POPE-tuned variant is a reference ceiling (see *Ablation*).

Method	10%	30%
FastV (Chen et al., 2024a)	65.99	80.47
DART (Wen et al., 2025)	71.91	83.43
HoloV (Zou et al., 2026)	75.84	83.27
PruMerge+ (Shang et al., 2025)	76.52	83.69
VisionZip (Yang et al., 2025)	78.46	84.73
DivPrune (Alvar et al., 2025)	79.27	84.68
VisionSelector (Zhu et al., 2025)	79.73	85.05
score-routed LapMerge (ours)	**81.31**	**85.50**
score-routed LapMerge (best α_a)	*82.01*	*85.87*

We evaluate on the full POPE benchmark (Li et al., 2023), which probes object hallucination through binary existence questions, at 10% and 30% visual-token budgets. For score-routed LapMerge we use the default widths $(\alpha_r, \alpha_a) = (1, 1)$, fixed without reference to POPE. As shown in Table 2, score-routed LapMerge reaches 81.31 and 85.50 F1, improving on VisionSelector's published F1 by $+1.58$ and $+0.45$ and outperforming all pruning- and merging-based baselines at both budgets. The gain is largest at the tighter budget, where hard selection discards 90% of the visual evidence: a paired bootstrap over the $9{,}000$ questions gives 95% CIs of $[+0.9, +2.4]$ at 10% and $[-0.1, +0.9]$ at 30% (Appendix D.3). Since the scorer, the backbone and the positional scheme are all inherited unchanged, the improvement can be attributed entirely to replacing hard selection with soft merging – a strict, training-free upgrade of an already released pipeline, obtained purely at inference time.

Ablation. The free parameters score-routed LapMerge introduces into this pipeline are the two widths (α_r, α_a). We sweep the read-out width α_a at fixed α_r=1, which interpolates between a nearly hard read-out and progressively broader merging. To measure how much the default leaves on the table, we grid-search α_a per budget directly on POPE (no gradient-based tuning or parameter updates). The best widths gain $+2.28$ and $+0.82$. Because these widths are chosen on the evaluation set itself, we report them only as an ablation study and not as a competing result. The benefit of merging over selection is not an artifact of a single lucky width: it persists across the moderate-width range of the sweep in Appendix D.3, although an over-wide kernel eventually falls below hard selection. Moreover, a single dataset-agnostic default already captures the bulk of it, so LapMerge can be deployed in a frozen VLM without per-task calibration.

4.4 End-to-End Learning: Long Range Arena

Table 3: **Long Range Arena, end-to-end training, budget 10%.** Test accuracy (%), mean ± std over 3 seeds. *Unreduced* and *Performer* are full-sequence references. Best result per task in **bold**; Pathfinder is at chance level for all methods, so no entry is bolded there.

Method	ListOps	Text	Retrieval	Image	Pathfinder
Unreduced	38.10 ± 1.21	65.03 ± 0.66	83.02 ± 0.20	35.94 ± 0.58	49.9 ± 0.1
Performer	38.57 ± 0.37	65.93 ± 0.29	82.2 ± 0.15	43.85 ± 0.13	50.21 ± 0.13
Uniform pooling	29.65 ± 0.25	75.05 ± 0.35	88.12 ± 0.33	42.29 ± 0.65	49.67 ± 0.21
ToMe	35.47 ± 2.36	60.97 ± 0.15	78.85 ± 0.34	36.16 ± 0.19	49.84 ± 0.06
Top-N selection	24.92 ± 9.24	55.68 ± 2.55	74.93 ± 9.13	36.89 ± 0.18	50.78 ± 1.42
TokenLearner	38.78 ± 0.71	65.47 ± 0.26	83.51 ± 0.34	37.34 ± 0.28	49.98 ± 0.19
PatchMerger	38.67 ± 0.81	66.26 ± 0.17	82.22 ± 0.2	25.52 ± 0.18	49.79 ± 0.03
LapMerge (ours)	$\mathbf{41.42 \pm 0.47}$	$\mathbf{77.05 \pm 0.47}$	$\mathbf{88.88 \pm 0.05}$	$\mathbf{47.03 \pm 0.39}$	49.83 ± 0.14

We next evaluate LapMerge when the reducer and backbone are trained jointly from scratch, rather than inserted into a pretrained encoder, on the Long Range Arena (LRA) benchmark. Results are presented in Table 3. LapMerge achieves the best accuracy on four of the five tasks – ListOps, Text, Retrieval and Image – outperforming the strongest budget-matched baseline by $+2.6$, $+2.0$, $+0.8$ and $+4.7$ points, respectively. Notably, it also surpasses both full-sequence models on all four (e.g., $+12.0$ on Text, $+5.9$ on Retrieval and $+11.1$ on Image over the unreduced Transformer), suggesting that structure-aware merging provides a useful inductive bias rather than acting merely as compression. Learned-selection baselines are considerably less stable: Top-N shows large variance across seeds on ListOps and Retrieval, and PatchMerger collapses on Image. On Pathfinder, all models, including the unreduced ones, remain at chance level ($\approx 50\%$), so differences on this task are not meaningful. A direct order intervention (permuting only the scan coordinate, everything else fixed) confirms that order matters for methods with a local, interval-based read-out (Appendix C.4).

5 Conclusion

We presented LapMerge, a differentiable operator that merges an ordered sequence of M tokens into exactly N outputs in $\mathcal{O}(M + N)$ time. The core idea – replacing unrestricted token-to-output assignment with a mass-based partition of the input order – delivers what previous reducers trade off against one another: an exact budget, end-to-end differentiability, and linear cost. The budget can be changed without changing operator parameters, and the learned mass doubles as an interpretable routing map. Empirically, LapMerge compresses frozen encoders at almost no accuracy cost, upgrades a released VLM pipeline without retraining – beating every POPE baseline, decisively at the 10% budget – achieves the best budget-matched accuracy on four of five LRA tasks, and runs at 10^7-token scale where existing reducers fail. We believe ordered, mass-based aggregation can become a standard primitive for long-sequence models, offering linear complexity while retaining information from the entire input. However, LapMerge is not a universal reducer: it prefers local aggregation, with contributions decaying with distance along the given ordering – ideal when neighboring tokens are spatially or temporally related, but limiting when related information sits at distant positions, as in some text tasks, where order-agnostic reduction may be preferable.

AI USE STATEMENT

We used AI assistants for editing and polishing the text, for implementing and debugging code, and as an aid in designing and analyzing experiments. The mathematical construction and its proofs were developed by the authors. All experimental results and final wording were verified by the authors, who take full responsibility for the content and correctness of the paper.

REPRODUCIBILITY STATEMENT

Appendix A contains the complete construction of the operator, full proofs of the propositions and the main theorem, the scan algorithms, and the explicit backward formulas, so the method can be reimplemented from the paper alone. Appendix B documents the runtime benchmark protocol (hardware, shapes, warm-up and repeat counts, timeout and failure handling) and the backend ablation. The experimental appendices describe the frozen-encoder, VLM, and end-to-end setups, and the datasets appendix lists all data sources; the histopathology cohorts use published patient-level splits with labels from the cited sources. The supplementary material includes the code for the histopathology experiments, with a synthetic-data demo, environment notes, and instructions for obtaining the gated real data, as well as the scripts and raw measurements behind the runtime benchmarks. Each reported table maps to the runner scripts and configurations in the corresponding supplementary package (histopathology; LRA; POPE/Cars/arXiv; runtime), which record the exact command, seeds, and hyperparameters per run; the runtime package additionally contains the raw CSVs from which every plot is generated, and the POPE evaluation is greedy and therefore deterministic given the pinned VisionSelector commit it states. All accuracy experiments use the native PyTorch implementation of the operator; the CUDA-fused backend appears only in the runtime study.

REFERENCES

Jean-Baptiste Alayrac, Jeff Donahue, Pauline Luc, Antoine Miech, Iain Barr, Yana Hasson, Karel Lenc, Arthur Mensch, Katie Millican, Malcolm Reynolds, Roman Ring, Eliza Rutherford, Serkan Cabi, Tengda Han, Zhitao Gong, Sina Samangooei, Marianne Monteiro, Jacob Menick, Sebastian Borgeaud, Andrew Brock, Aida Nematzadeh, Sahand Sharifzadeh, Mikolaj Bińkowski, Ricardo Barreira, Oriol Vinyals, Andrew Zisserman, and Karen Simonyan. Flamingo: a visual language model for few-shot learning. In *Advances in Neural Information Processing Systems*, 2022.

Saeed Ranjbar Alvar, Gursimran Singh, Mohammad Akbari, and Yong Zhang. Divprune: Diversity-based visual token pruning for large multimodal models. In *2025 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 9392–9401. IEEE, 2025.

Shuai Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, et al. Qwen2.5-vl technical report. *arXiv preprint arXiv:2502.13923*, 2025.

Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

Mathieu Blondel, Olivier Teboul, Quentin Berthet, and Josip Djolonga. Fast differentiable sorting and ranking. In *International Conference on Machine Learning*, 2020.

Daniel Bolya, Cheng-Yang Fu, Xiaoliang Dai, Peizhao Zhang, Christoph Feichtenhofer, and Judy Hoffman. Token merging: Your ViT but faster. In *International Conference on Learning Representations*, 2023.

Liang Chen, Haozhe Zhao, Tianyu Liu, Shuai Bai, Junyang Lin, Chang Zhou, and Baobao Chang. An image is worth 1/2 tokens after layer 2: Plug-and-play inference acceleration for large vision-language models. In *European Conference on Computer Vision*, 2024a.

Mengzhao Chen, Wenqi Shao, Peng Xu, Mingbao Lin, Kaipeng Zhang, Fei Chao, Rongrong Ji, Y. Qiao, and Ping Luo. Diffrate : Differentiable compression rate for efficient vision transformers. *2023 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 17118–17128, 2023.

Richard J Chen, Tong Ding, Ming Y Lu, Drew FK Williamson, Guillaume Jaume, Andrew H Song, Bowen Chen, Andrew Zhang, Daniel Shao, Muhammad Shaban, et al. Towards a general-purpose foundation model for computational pathology. *Nature medicine*, 30(3):850–862, 2024b.

Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*, 2019.

Krzysztof Marcin Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Quincy Davis, Afroz Mohiuddin, Lukasz Kaiser, David Benjamin Belanger, Lucy J Colwell, and Adrian Weller. Rethinking attention with performers. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=Ua6zuk0WRH.

Jean-Baptiste Cordonnier, Aravindh Mahendran, Alexey Dosovitskiy, Dirk Weissenborn, Jakob Uszkoreit, and Thomas Unterthiner. Differentiable patch selection for image recognition. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2021.

Zihang Dai, Guokun Lai, Yiming Yang, and Quoc V. Le. Funnel-transformer: Filtering out sequential redundancy for efficient language processing. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. In *Advances in neural information processing systems*, volume 35, pp. 16344–16359, 2022.

Mohsen Fayyaz, Soroush Abbasi Koohpayegani, Farnoush Rezaei Jafari, Sunando Sengupta, Hamid Reza Vaezi Joze, Eric Sommerlade, Hamed Pirsiavash, and Juergen Gall. Adaptive token sampling for efficient vision transformers. In *European Conference on Computer Vision (ECCV)*, 2022.

Aditya Grover, Eric Wang, Aaron Zweig, and Stefano Ermon. Stochastic optimization of sorting networks via continuous relaxations. In *International Conference on Learning Representations*, 2019.

Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*, 2023.

Albert Gu, Karan Goel, and Christopher Ré. Efficiently modeling long sequences with structured state spaces. In *International Conference on Learning Representations*, 2022.

Lan He, Min Yao, Sean I. Young, and Lijian Xu. Diffprune: differentiable information throttling for token pruning in vision-language models. *arXiv preprint arXiv:2608.01985*, 2026a.

Landi He, Xiao-Yu Yang, and Lijian Xu. The model knows which tokens matter: Automatic token selection via noise gating. *ArXiv*, abs/2603.07135, 2026b.

Injune Hwang and Kyogu Lee. Removing speaker information from speech representation using variable-length soft pooling. *arXiv preprint arXiv:2404.00856*, 2024.

Maximilian Ilse, Jakub M. Tomczak, and Max Welling. Attention-based deep multiple instance learning. In *International Conference on Machine Learning*, 2018.

Andrew Jaegle, Felix Gimeno, Andy Brock, Oriol Vinyals, Andrew Zisserman, and Joao Carreira. Perceiver: General perception with iterative attention. In *International Conference on Machine Learning*, 2021.

Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention. In *International conference on machine learning*, pp. 5156–5165. PMLR, 2020.

Jakob Nikolas Kather, Alexander T Pearson, Niels Halama, Dirk Jäger, Jeremias Krause, Sven H Loosen, Alexander Marx, Peter Boor, Frank Tacke, Ulf Peter Neumann, et al. Deep learning can predict microsatellite instability directly from histology in gastrointestinal cancer. *Nature medicine*, 25(7):1054–1056, 2019.

Jonathan Krause, Michael Stark, Jia Deng, and Li Fei-Fei. 3D object representations for fine-grained categorization. In *IEEE International Conference on Computer Vision Workshops*, pp. 554–561, 2013.

Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.

Juho Lee, Yoonho Lee, Jungtaek Kim, Adam R. Kosiorek, Seungjin Choi, and Yee Whye Teh. Set transformer: A framework for attention-based permutation-invariant neural networks. In *International Conference on Machine Learning*, 2019.

Bin Li, Yin Li, and Kevin W. Eliceiri. Dual-stream multiple instance learning network for whole slide image classification with self-supervised contrastive learning. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021.

Yifan Li, Yifan Du, Kun Zhou, Jinpeng Wang, Wayne Xin Zhao, and Ji rong Wen. Evaluating object hallucination in large vision-language models. In *Conference on Empirical Methods in Natural Language Processing*, 2023.

Youwei Liang, Chongjian Ge, Zhan Tong, Yibing Song, Jue Wang, and Peng-Tao Xie. Not all patches are what you need: Expediting vision transformers via token reorganizations. *ArXiv*, abs/2202.07800, 2022. URL https://api.semanticscholar.org/CorpusID:246867285.

Drew Linsley, Junkyung Kim, Vijay Veerabadran, Charles Windolf, and Thomas Serre. Learning long-range spatial dependencies with horizontal gated recurrent units. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

Yang Liu, Nilay S Sethi, Toshinori Hinoue, Barbara G Schneider, Andrew D Cherniack, Francisco Sanchez-Vega, Jose A Seoane, Farshad Farshidfar, Reanne Bowlby, Mirazul Islam, et al. Comparative molecular analysis of gastrointestinal adenocarcinomas. *Cancer cell*, 33(4):721–735, 2018.

Ming Y. Lu, Drew F. K. Williamson, Tiffany Y. Chen, Richard J. Chen, Matteo Barbieri, and Faisal Mahmood. Data-efficient and weakly supervised computational pathology on whole-slide images. *Nature Biomedical Engineering*, 5(6):555–570, 2021.

Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 142–150, 2011.

Mahmood Lab. UNI2-h. Hugging Face model card, 2025. URL https://huggingface.co/MahmoodLab/UNI2-h.

Dmitrii Marin, Jen-Hao Rick Chang, Anurag Ranjan, Anish Prabhu, Mohammad Rastegari, and Oncel Tuzel. Token pooling in vision transformers for image classification. In *IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*, pp. 12–21, 2023.

Nikita Nangia and Samuel R. Bowman. ListOps: A diagnostic dataset for latent tree learning. In *Proceedings of the NAACL Student Research Workshop*, pp. 92–99, 2018.

Sebastian Prillo and Julian Eisenschlos. SoftSort: A continuous relaxation for the argsort operator. In *International Conference on Machine Learning*, 2020.

Dragomir R. Radev, Pradeep Muthukrishnan, Vahed Qazvinian, and Amjad Abu-Jbara. The ACL anthology network corpus. *Language Resources and Evaluation*, 47(4):919–944, 2013.

Yongming Rao, Wenliang Zhao, Benlin Liu, Jiwen Lu, Jie Zhou, and Cho-Jui Hsieh. DynamicViT: Efficient vision transformers with dynamic token sparsification. In *Advances in Neural Information Processing Systems*, 2021.

Cedric Renggli, Andre Susano Pinto, Neil Houlsby, Basil Mustafa, Joan Puigcerver, and Carlos Riquelme. Learning to merge tokens in vision transformers. *arXiv preprint arXiv:2202.12015*, 2022.

Michael S. Ryoo, AJ Piergiovanni, Anurag Arnab, Mostafa Dehghani, and Anelia Angelova. Tokenlearner: Adaptive space-time tokenization for videos. In *Advances in Neural Information Processing Systems*, 2021.

Yuzhang Shang, Mu Cai, Bingxin Xu, Yong Jae Lee, and Yan Yan. LLaVA-PruMerge: Adaptive token reduction for efficient large multimodal models. In *2025 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 22857–22867. IEEE, 2025.

Zhuchen Shao, Hao Bian, Yang Chen, Yifeng Wang, Jian Zhang, Xiangyang Ji, and Yongbing Zhang. TransMIL: Transformer based correlated multiple instance learning for whole slide image classification. In *Advances in Neural Information Processing Systems*, 2021.

Łukasz Struski, Michał B. Bednarczyk, Igor T. Podolak, and Jacek Tabor. LapSum: One method to differentiate them all: Ranking, sorting and top-k selection. In *International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 56990–57007, 2025.

Łukasz Struski, Joanna Wojciechowicz, Jakub Antczak, Marcin Mazur, Kamil Ksiazek, and Jacek Tabor. Fast LapSum: Exact differentiable top-k at million scale. *arXiv preprint arXiv:2608.06912*, 2026.

Wenhao Tang, Fengtao Zhou, Sheng Huang, Xiang Zhu, Yi Zhang, and Bo Liu. Feature re-embedding: Towards foundation model-level performance in computational pathology. In *2024 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 11343–11352. IEEE, 2024.

Yi Tay, Mostafa Dehghani, Samira Abnar, Yikang Shen, Dara Bahri, Philip Pham, Jinfeng Rao, Liu Yang, Sebastian Ruder, and Donald Metzler. Long range arena: A benchmark for efficient transformers. In *International Conference on Learning Representations*, 2021.

Hoai-Chau Tran, Duy M. H. Nguyen, Duy M. Nguyen, Trung-Tin Nguyen, Ngan Le, Pengtao Xie, Daniel Sonntag, James Y. Zou, Binh T. Nguyen, and Mathias Niepert. Accelerating transformers with spectrum-preserving token merging. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Sophia J Wagner, Daniel Reisenbüchler, Nicholas P West, Jan Moritz Niehues, Jiefu Zhu, Sebastian Foersch, Gregory Patrick Veldhuizen, Philip Quirke, Heike I Grabsch, Piet A van den Brandt, et al. Transformer-based biomarker prediction from colorectal cancer histology: A large-scale multicentric study. *Cancer cell*, 41(9):1650–1661, 2023.

Zichen Wen, Yifeng Gao, Shaobo Wang, Junyuan Zhang, Qintong Zhang, Weijia Li, Conghui He, and Linfeng Zhang. Stop looking for "important tokens" in multimodal language models: Duplication matters more. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 9972–9991, 2025.

Senqiao Yang, Yukang Chen, Zhuotao Tian, Chengyao Wang, Jingyao Li, Bei Yu, and Jiaya Jia. Visionzip: Longer is better but not necessary in vision language models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2025.

Shu Yang, Yihui Wang, and Hao Chen. Mambamil: Enhancing long sequence modeling with sequence reordering in computational pathology. In *International conference on medical image computing and computer-assisted intervention*, pp. 296–306. Springer, 2024.

Hongrun Zhang, Yanda Meng, Yitian Zhao, Yihong Qiao, Xiaoyun Yang, Sarah E Coupland, and Yalin Zheng. Dtfd-mil: Double-tier feature distillation multiple instance learning for histopathology whole slide image classification. In *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 18780–18790. IEEE, 2022.

Jiaying Zhu, Yurui Zhu, Xin Lu, Wenrui Yan, Dong Li, Kunlin Liu, Xueyang Fu, and Zhengjun Zha. Visionselector: End-to-end learnable visual token compression for efficient multimodal llms. *ArXiv*, abs/2510.16598, 2025.

Xin Zou, Di Lu, Yizhou Wang, Yibo Yan, Yuanhuiyi Lyu, Xu Zheng, Linfeng Zhang, and Xuming Hu. Don't just chase "highlighted tokens" in mllms: Revisiting visual holistic context retention. *Advances in Neural Information Processing Systems*, 38:39800–39832, 2026.

Appendix

Table of Contents

A The LapMerge Algorithm **15**
- A.1 Locating the barriers . . . 17
- A.2 Differentiating the Barriers . . . 18
- A.3 Aggregating Tokens Given the Barriers . . . 19
- A.4 Scorer and routing normalization . . . 21

B Runtime and Large-Scale Token Reduction **22**
- B.1 What is being accelerated? . . . 22
- B.2 Three implementation levels of one operator . . . 23
- B.3 Backend ablations and dispatch calibration . . . 24
- B.4 Matched comparison with learned poolers . . . 26
- B.5 Scaling and stress tests . . . 28
- B.6 Measurement protocol and interpretation . . . 29

C When Does Merging Beat Selection? **29**
- C.1 Matched evaluation protocol . . . 29
- C.2 Spatially ordered tokens: Stanford Cars . . . 30
- C.3 Distributed evidence and order ablations . . . 30
- C.4 Order intervention: permuting the scan on LRA-Image . . . 31
- C.5 Planned confirmation experiments . . . 33

D Drop-in Token Reduction in a Frozen VLM: Details **34**
- D.1 Pipeline . . . 34
- D.2 Score-routed LapMerge: general form, properties, and cost . . . 34
- D.3 Sensitivity to the routing and read-out widths . . . 35

E Datasets **35**
- E.1 Histopathology . . . 36
- E.2 LRA . . . 37
- E.3 arXiv Topic Classification . . . 38
- E.4 POPE . . . 39
- E.5 Fine-grained Vision: Stanford Cars . . . 39

A THE LAPMERGE ALGORITHM

This appendix gives the complete mathematical and scan-level construction of LapMerge. The input is an ordered sequence of M tokens, non-negative routing weights r_i summing to one, an output budget N, and a Laplace scale $\alpha > 0$. The output consists of exactly N convex combinations of the input tokens. We prove existence and differentiability of the equal-mass barriers, derive their closed-form Laplace solve and matrix-free derivatives, and then prove the corresponding aggregation and backward formulas. Implementation and measurement variants are compared in Appendix B.2. Throughout, Φ and b_ℓ have the meanings in equation 1–equation 2. A reader who mainly wants to implement the operator can go directly to Algorithms 1–3, the implementation checklist closing this section, and the configuration recipe in Table 5; the text in between derives and proves each step.

Setup and notation. Tokens $x_1, \dots, x_M \in \mathbb{R}^d$ arrive sorted by position $p_1 < \cdots < p_M$. The operator receives non-negative routing masses r_i satisfying $\sum_i r_i = 1$; an optional scorer may produce these masses upstream, but is not part of the operator defined below. As discussed in Section 3, this scorer can use a sigmoid, softplus, or differentiable soft-top-k normalization, depending on the model variant. Here we only require its final admissible masses r_i. We fix the budget N and the Laplace scale α. In practice, $\alpha \propto \mathrm{spread}(p)/M$, ensuring the kernel remains local as M grows. If position ties occur, they are broken using the original token indices. The Laplace CDF is defined as $\sigma(z) = \frac{1}{2}e^z$ for $z \le 0$ and $\sigma(z) = 1 - \frac{1}{2}e^{-z}$ for $z \ge 0$.

Why no $O(M \log M)$ sorting term appears. The order of the tokens is part of the input to this operator. For sequences, the default position is simply the original index, $p_i = i$, and for spatial or other structured inputs the caller supplies the already chosen traversal order. Consequently, all prefix/suffix scans and barrier readouts operate directly along the existing token axis; the algorithm does not sort the input at every forward pass. Under this ordered-input convention, the work of the scan and readout is $O((M+N)d)$ and the auxiliary memory is $O((M+N)d)$ (or $O(M+N)$ for the scalar barrier summaries). A generic implementation that is given unsorted coordinates may prepend a stable sort, adding $O(M \log M)$ work and the corresponding permutation/gather cost, but that is an input-preparation cost rather than part of the ordered LapMerge operator. All runtime measurements in this paper use the ordered-input setting; the original indices provide the deterministic tie-break when coordinates are equal.

Cost of the optional scorer. The linear bound above describes the LapMerge operator after the routing masses r_i have been supplied. If the upstream scorer uses a differentiable soft-top-k construction implemented by sorting the scalar logits, the end-to-end routing stage has an additional $O(M \log M)$ term. This does not change the $O((M+N)d)$ complexity of the merge itself. In practice, the extra term is often small compared with the feature-valued scan: only one scalar per token is sorted, and GPU sorting primitives are highly optimized. For this reason the asymptotic distinction should be reported explicitly, while measured runtimes should include the scorer whenever the complete reduction module is being benchmarked. If the scorer instead maps logits independently through a pointwise sigmoid or softplus and normalizes the resulting non-negative masses without top-N sorting, this additional sorting cost is absent and the routing stage is linear in M.

From a random token to a merged token. Choose I with $\Pr(I = i) = r_i$ and let $P_I = p_I + \alpha E_I$, where each E_i has CDF σ. Then $\Pr(P_I \le z) = \Phi(z)$ by conditioning on I. For $B_\ell = [b_{\ell-1}, b_\ell)$, the barrier equation gives $\Pr(P_I \in B_\ell) = \Phi(b_\ell) - \Phi(b_{\ell-1}) = 1/N$. Conditioning once more on I yields

$$\Pr(I = i \mid P_I \in B_\ell) = N r_i \left[\sigma\left(\frac{b_\ell - p_i}{\alpha} \right) - \sigma\left(\frac{b_{\ell-1} - p_i}{\alpha} \right) \right] = w_i^\ell. \tag{7}$$

Every w_i^ℓ is non-negative and $\sum_i w_i^\ell = 1$, so $t_\ell = \mathbb{E}[x_I \mid P_I \in B_\ell] = \sum_i w_i^\ell x_i$ is a convex combination. Moreover, $\sum_{\ell=1}^N w_i^\ell = N r_i$: all of a token's routing mass is accounted for across the outputs.

Full proof of Proposition 1. For a continuous, strictly positive density κ, each summand in $\Phi'(z) = \alpha^{-1} \sum_i r_i \kappa((z - p_i)/\alpha)$ is non-negative and at least one is strictly positive; hence $\Phi'(z) > 0$ for all finite z. Because $\sum_i r_i = 1$, the CDF limits are $\Phi(-\infty) = 0$ and $\Phi(+\infty) = 1$. The

intermediate value theorem and strict monotonicity give one and only one solution of $\Phi(b_\ell) = \ell/N$ for each $1 \le \ell < N$, and $b_1 < \cdots < b_{N-1}$. For differentiability, consider a differentiable path $r(t)$ in the probability simplex. Then $\sum_j \dot{r}_j = 0$ and implicit differentiation gives

$$0 = \Phi'(b_\ell)\dot{b}_\ell + \sum_j \sigma\left(\frac{b_\ell - p_j}{\alpha}\right)\dot{r}_j, \qquad \dot{b}_\ell = -\frac{\sum_j \sigma((b_\ell - p_j)/\alpha)\dot{r}_j}{\Phi'(b_\ell)}. \tag{8}$$

The denominator is positive, so the implicit function theorem applies. Equation equation 3 is one ambient representative of this derivative: on the simplex, adding the same constant to all its entries does not change any directional derivative because $\sum_j \dot{r}_j = 0$. This observation also explains why equivalent VJPs may differ by a constant vector before the upstream normalization is differentiated.

Three numerical and algorithmic pitfalls. First, directly computing $B_i = \sum_{j \le i} r_j e^{p_j/\alpha}$ and $C_i = \sum_{j > i} r_j e^{-p_j/\alpha}$ in equation 6 can overflow or underflow before the final factors $e^{\mp x/\alpha}$ bring the CDF back into $[0, 1]$; recentering each exponential at its query point avoids those large intermediate values. Second, comparing ℓ/N with the unsmoothed prefix mass $A_i = \sum_{j \le i} r_j$ can select the wrong barrier cell, because $A_i \ne \Phi(p_i)$; we search the smoothed CDF nodes instead (Safeguard A). Third, tangents, cotangents, and feature coordinates can be signed, whereas a logarithmic scan requires non-negative summands; separate positive and negative scans handle them. The in-cell solve is further stabilized by a suitable endpoint anchor and a Vieta root (Safeguard B).

Principle 1: Recenter decayed sums at the query point. For a query point x located in cell i (i.e., exactly i tokens satisfy $p_j \le x$) and a non-negative sequence $w_1, \ldots, w_M$ (with $w = r$ by default), define the log decayed sums

$$L_w(x; i) = \log \sum_{j \le i} w_j e^{(p_j - x)/\alpha}, \qquad R_w(x; i) = \log \sum_{j > i} w_j e^{(x - p_j)/\alpha}. \tag{9}$$

Both quantities are computed via global LogCumsumExp scans. For the left term, we scan $\log w_j + p_j/\alpha$ from left to right and subtract x/α upon reading out cell i. For the right term, we execute a reverse scan on $\log w_j - p_j/\alpha$ and add x/α. Here $\log 0 = -\infty$ is an empty summand. Every exponent is recentered to be non-positive at readout: $(p_j - x)/\alpha \le 0$ on the left and $(x - p_j)/\alpha \le 0$ on the right. The scan stores logarithms, so large global offsets are never exponentiated directly.

Principle 2: Decompose signed sequences prior to scanning. The LogCumsumExp operation assumes non-negative summands, a condition satisfied by r_i but violated by tangents, cotangents, and token coordinates. For a signed sequence v_i, write $v_i = v_i^+ - v_i^-$, with $v_i^+ = \max(v_i, 0)$ and $v_i^- = \max(-v_i, 0)$. Run a standard LogCumsumExp on each component independently and combine the results at readout:

$$\textstyle\sum_{j \le i} v_j = \exp\big(\textsc{LogCumsumExp}_{j \le i}(\log v_j^+)\big) - \exp\big(\textsc{LogCumsumExp}_{j \le i}(\log v_j^-)\big). \tag{10}$$

Recombination occurs only after the prefix and suffix states are evaluated. This doubles the scan operations for signed inputs but keeps the logarithms well defined. We denote this wrapper Signed-LogCumsumExp. For any signed w, let $D_w^L = e^{L_{w^+}} - e^{L_{w^-}}$ and $D_w^R = e^{R_{w^+}} - e^{R_{w^-}}$, with empty sums equal to zero. For non-negative w, these reduce to e^{L_w} and e^{R_w}.

One construction, five uses. With $A_i^w = \sum_{j \le i} w_j$, we define the recentered scan functional as:

$$\Phi_w(x; i) := A_i^w - \tfrac{1}{2} D_w^L(x; i) + \tfrac{1}{2} D_w^R(x; i). \tag{11}$$

Setting $w = r$ recovers the mixture CDF Φ within cell i. The same prefix/suffix states serve five tasks: computing node values, solving for barriers, evaluating barrier JVPs, computing barrier VJPs, and aggregating values. The reverse VJP uses the complementary orientation $\sigma((b_\ell - p_j)/\alpha) = 1 - \sigma((p_j - b_\ell)/\alpha)$ after interchanging barrier and token roles. Table 4 specifies the inputs to these scans.

Context	Weight w	Summation domain	Query point	Cell i source
§A.1, node_i	r	tokens j	p_i	index i
§A.1, solve/density	r	tokens j	b_ℓ	Safeguard A search
§A.2, FORWARD	$\dot{r}$	tokens j	b_ℓ	reused from solve
§A.2, REVERSE	c (below)	barriers ℓ	p_j	complement of swapped scan
§A.3	$r \odot x$	tokens j	b_ℓ	reused from solve

Table 4: Five applications of the recentered prefix/suffix scans. The reverse row takes the complement of $\Phi_c(p_j)$ to obtain the required CDF orientation. Cell indices are determined by a monotone merge search.

Algorithm 1 FINDBARRIERS, $\mathcal{O}(M+N)$

Require: sorted positions $p_{1:M}$, weights $r_{1:M}$, budget N, scale α
Ensure: barriers $b_{0:N}$ (with $b_0 = -\infty$, $b_N = +\infty$); cell indices $i_{0:N}$ (with $i_0 = 0$, $i_N = M$); densities $\Phi'(b_{1:N-1})$

1: $\mathrm{node}_i \leftarrow \mathrm{logit}\,\Phi_r(p_i; i)$ for all i
2: **for** $\ell = 1, \ldots, N-1$ **do**
3: $\quad i_\ell \leftarrow$ merge-search $\mathrm{logit}(\ell/N)$ against $\mathrm{node}_{1:M}$
4: $\quad q_\ell \leftarrow$ finite endpoint of cell i_ℓ with $\Phi_r(q_\ell)$ closest to ℓ/N
5: $\quad b_\ell \leftarrow$ cancellation-safe (Vieta) root of $\Phi_r(x; i_\ell) = \ell/N$, recentered at q_ℓ
6: $\quad \Phi'(b_\ell) \leftarrow \frac{1}{2\alpha}\left[e^{L_r(b_\ell; i_\ell)} + e^{R_r(b_\ell; i_\ell)}\right]$
7: **end for**
8: $b_0 \leftarrow -\infty, i_0 \leftarrow 0; \qquad b_N \leftarrow +\infty, i_N \leftarrow M$
9: **return** $b_{0:N}, i_{0:N}, \Phi'(b_{1:N-1})$

A.1 LOCATING THE BARRIERS

Algorithm 1 outlines the procedure for locating the barriers. Line 1 evaluates the smoothed CDF at every token position and stores its log-odds, $\mathrm{node}_i = \mathrm{logit}\,\Phi_r(p_i; i)$. The following non-negative increments make these log-odds accurate even when a node CDF is close to zero or one. Set $U_i = \sum_{j \le i} r_j e^{(p_j - p_i)/\alpha}$, $V_i = \sum_{j \ge i} r_j e^{(p_i - p_j)/\alpha}$, and $\delta_i = (p_{i+1} - p_i)/\alpha$. Then, with $d_0 = \frac{1}{2}V_1$, $d_M = \frac{1}{2}U_M$, and

$$d_i = \tfrac{1}{2}\big(1 - e^{-\delta_i}\big)(U_i + V_{i+1}), \qquad 1 \le i < M, \tag{12}$$

we have $d_i > 0$, $\sum_{i=0}^{M} d_i = 1$, and $\Phi(p_i) = \sum_{h=0}^{i-1} d_h$, $1 - \Phi(p_i) = \sum_{h=i}^{M} d_h$. The recurrences $U_i = r_i + e^{-\delta_{i-1}} U_{i-1}$ and $V_i = r_i + e^{-\delta_i} V_{i+1}$ compute all increments in $O(M)$. In floating point, run these recurrences in the log domain and compute $\log(1 - e^{-\delta_i})$ via $\log[-\mathrm{expm1}(-\delta_i)]$. Obtain each node logit by subtracting a suffix LOGCUMSUMEXP of $\log d_h$ from a prefix LOGCUMSUMEXP. This avoids both underflow in small increments and subtracting a CDF value from one near the right tail. The nodes are strictly increasing, so all barrier cells can then be found in a single merge pass. Two additional details make the in-cell solve safe.

Derivation of the Laplace cell formula. In cell $p_i \le x < p_{i+1}$, the CDF branch for $j \le i$ is $1 - \frac{1}{2}e^{(p_j - x)/\alpha}$ and the branch for $j > i$ is $\frac{1}{2}e^{(x - p_j)/\alpha}$. Thus, with $A_i = \sum_{j \le i} r_j$, $B_i = \sum_{j \le i} r_j e^{p_j/\alpha}$, and $C_i = \sum_{j > i} r_j e^{-p_j/\alpha}$,

$$\Phi(x) = A_i - \tfrac{1}{2} B_i e^{-x/\alpha} + \tfrac{1}{2} C_i e^{x/\alpha}. \tag{13}$$

The three arrays are an ordinary prefix, a weighted prefix, and a weighted suffix. Their raw form proves the identity; numerical evaluation uses the recentered forms in equation 9. Evaluating Φ at all M token positions and merging those ordered values with the ordered targets ℓ/N identifies every barrier cell in $O(M+N)$ work. Inside a cell, $u = e^{(x-q)/\alpha} > 0$ turns equation 13 into equation 15. Since $\Phi'(x) > 0$, the positive root in the selected cell is the unique barrier. The root and its density require constant work per barrier, proving the forward bound for barrier location.

Safeguard A: Search against Φ itself, not the plain mass A. Because $\mathrm{node}_{1:M}$ is monotone, locating the cell $[p_i, p_{i+1})$ containing each b_ℓ reduces to merging two sorted lists: the nodes and

the targets $\mathrm{logit}(\ell/N)$. A single pair of pointers advances through the M nodes and $N-1$ targets, costing $\mathcal{O}(M+N)$ time rather than requiring $N-1$ independent searches. The unsmoothed prefix mass A_i must not be used here: it ignores the kernel tails and can choose the wrong cell.

The search uses two sentinel cells. Cell $i=0$ denotes $(-\infty, p_1)$, cells $1,\dots,M-1$ denote $[p_i, p_{i+1})$, and cell $i=M$ denotes $[p_M, +\infty)$. The merge search is therefore performed on the extended node list $(-\infty, \mathrm{node}_1, \dots, \mathrm{node}_M, +\infty)$. At an exact node value, we use the right-continuous convention, so a barrier equal to p_i is assigned to cell i.

Safeguard B: Solve inside the cell anchored at whichever end stays bounded. Inside a known cell, set $u = \exp((x-q)/\alpha)$, where q is one of the finite cell endpoints. Define the recentered masses:

$$L_q = \sum_{j\le i} r_j e^{(p_j-q)/\alpha}, \qquad R_q = \sum_{j>i} r_j e^{(q-p_j)/\alpha}, \qquad A_i = \sum_{j\le i} r_j. \tag{14}$$

Holding these quantities fixed turns $\Phi_r(x;i) = \ell/N$ into the explicit quadratic equation:

$$R_q u^2 + 2(A_i - \ell/N)u - L_q = 0. \tag{15}$$

The valid solution is the positive root whose $x = q + \alpha \log u$ lies in the chosen cell. Let $B = 2(A_i - \ell/N)$ and $\Delta = B^2 + 4R_qL_q$. When $R_q > 0$, evaluate the positive root in the cancelation-safe form:

$$u = \begin{cases} \dfrac{2L_q}{B+\sqrt{\Delta}}, & B \ge 0, \\ \dfrac{-B+\sqrt{\Delta}}{2R_q}, & B < 0. \end{cases} \tag{16}$$

The first branch is derived using Vieta's formulas and avoids subtracting nearly equal numbers. If a tail is numerically zero, use the corresponding linear limit and then clamp only at the cell endpoint. For an interior cell, choose $q = p_i$ or $q = p_{i+1}$ according to which endpoint is closer in CDF mass to ℓ/N. At either endpoint, the defining sums L_q and R_q have non-positive exponents. For $i=0$, use $q = p_1$ and the one-sided right-tail formula; for $i = M$, use $q = p_M$ and the one-sided left-tail formula. Finally, verify that the root lies in the selected cell. The same local scan values give $\Phi'(b_\ell) = \left(e^{L_r(b_\ell;i_\ell)} + e^{R_r(b_\ell;i_\ell)}\right)/(2\alpha)$, so no additional scan is required.

Remark (extreme-separation corner). When the linear coefficient vanishes (B=0, exactly balanced masses on the two sides) and one tail underflows ($e^{-D/\alpha}$ below the floating-point floor, i.e. a barrier more than $\sim$88 kernel widths from all support in FP32), the Vieta branch and its linear fallback both lose the root; the remedy is the log-space branch $\log u = \frac{1}{2}(\log L_q - \log R_q)$, with L_q and R_q kept as logarithms throughout. None of our configurations enter this regime: with strictly positive routing masses and index-spaced positions, barriers fall within a few kernel widths of populated support, and the bounded-end anchoring above covers one-sided underflow. Implementations targeting extremely separated or near-degenerate mass configurations should enable this branch. Separately from computational stability, the derivative factor $1/\Phi'(b_\ell)$ can be genuinely ill-conditioned when the density at a barrier is very small; no logarithmic representation removes this, as it is a property of the problem rather than of the implementation.

A.2 Differentiating the Barriers

Derivation of the gradient formula. Each barrier satisfies the scalar equation $\Phi(b_\ell; r) = \ell/N$. Differentiating it with respect to r_j and using $\partial_x \Phi = \Phi'(b_\ell) > 0$ gives:

$$\frac{\partial b_\ell}{\partial r_j} = -\frac{\sigma((b_\ell - p_j)/\alpha)}{\Phi'(b_\ell)}. \tag{17}$$

This is a dense Jacobian entry, but training requires only its products with a tangent or a cotangent. The rest of this section demonstrates how to compute both products using scans.

Motivation against direct materialization. The entries form an $(N-1)\times M$ matrix, but materializing it directly would restore the quadratic cost removed during the forward pass. A forward-mode tangent requires $\dot{b} = (\partial b/\partial r)\dot{r}$, whereas reverse-mode training requires the transpose product $\bar{r} = (\partial b/\partial r)^\top \bar{b}$. Both operations can be formulated as matrix-free scan problems.

Algorithm 2 BARRIERSENSITIVITY – the custom backward pass, Theorem 1, $\mathcal{O}(M+N)$

Require: $p_{1:M}, r_{1:M}, \alpha$, and $(b_{1:N-1}, i_{1:N-1}, \Phi'(b_{1:N-1}))$ from Algorithm 1, computed under `no_grad`

1: **procedure** FORWARD (JVP)(tangent $\dot{r}_{1:M}$)
2: $\quad \dot{b}_\ell \leftarrow -\Phi_{\dot{r}}(b_\ell; i_\ell)/\Phi'(b_\ell)$ for every ℓ
3: $\quad$ **return** $\dot{b}_{1:N-1}$
4: **end procedure**
5: **procedure** REVERSE (VJP)(cotangent $\bar{b}_{1:N-1}$)
6: $\quad c_\ell \leftarrow -\bar{b}_\ell/\Phi'(b_\ell)$ for every ℓ
7: $\quad$ merge-search $p_{1:M}$ against the sorted barriers $b_{1:N-1}$
8: $\quad C \leftarrow \sum_{\ell=1}^{N-1} c_\ell$; $\bar{r}_j \leftarrow C - \Psi_c(p_j; k_j)$ for every j, with $k_j = \#\{\ell : b_\ell \le p_j\}$
9: $\quad$ **return** $\bar{r}_{1:M}$
10: **end procedure**

Exploiting shared structure. The kernel factor in equation 3 is identical to the factor used by Φ. Consequently, the JVP numerator is the generalized scan $\Phi_{\dot{r}}(b_\ell; i_\ell)$, evaluated at the previously identified barrier cell. For the VJP, the roles reverse: we scan over the $N-1$ barriers once and read the complement of the resulting CDF across all M token positions. Neither direction materializes individual Jacobian entries.

Analytical backward and saved state. The forward barrier solve runs under `no_grad`. A custom `torch.autograd.Function` supplies the exact JVP/VJP formulas instead of tracing through the discrete cell search, endpoint branch, and square-root solve. The benefit is a small, predictable saved state: no computation graph is stored through the branchy solver, the backward reuses the barriers, cell indices, and scan values already computed in the forward pass, and the fixed formulas avoid per-operation tracing overhead. Both products retain the $\mathcal{O}(M+N)$ cost of the forward pass.

Theorem 1 now follows operationally: both products use one scan over the longer sequence and one merge over the shorter one. For the JVP, we build $L_{\dot{r}}, R_{\dot{r}}$ with SIGNEDLOGCUMSUMEXP, evaluate them at the stored barrier cells, and divide by the stored densities. No new cell search is needed. For the VJP, substituting equation 3 into the adjoint yields:

$$\bar{r}_j = \sum_{\ell=1}^{N-1} \frac{\partial b_\ell}{\partial r_j}\,\bar{b}_\ell = \sum_{\ell=1}^{N-1} \underbrace{-\frac{\bar{b}_\ell}{\Phi'(b_\ell)}}_{=:c_\ell}\,\sigma\!\left(\frac{b_\ell - p_j}{\alpha}\right), \tag{18}$$

where c_ℓ already absorbs the minus sign. To make the role swap explicit, take the already ordered barrier support points $z_\ell = b_\ell$ (the barriers are sorted by construction, so no additional sort is performed) with weights c_ℓ and define, for a query y in cell k,

$$\Psi_c(y;k) = \sum_{\ell\le k} c_\ell - \tfrac{1}{2}\sum_{\ell\le k} c_\ell e^{(z_\ell - y)/\alpha} + \tfrac{1}{2}\sum_{\ell>k} c_\ell e^{(y-z_\ell)/\alpha}. \tag{19}$$

The displayed $\Psi_c(p_j;k_j)$ contains $\sigma((p_j-b_\ell)/\alpha)$, whereas the barrier derivative contains $\sigma((b_\ell - p_j)/\alpha)$. By Laplace CDF symmetry,

$$\boxed{\bar{r}_j = C - \Psi_c(p_j;k_j)}, \qquad C = \sum_{\ell=1}^{N-1} c_\ell. \tag{20}$$

Here k_j is the number of barriers at or below p_j; $k_j = 0$ and $k_j = N-1$ cover the sentinel regions. Equivalently, the right-hand side is $\frac{1}{2}\sum_{\ell\le k_j} c_\ell e^{(b_\ell - p_j)/\alpha} + \sum_{\ell>k_j} c_\ell - \frac{1}{2}\sum_{\ell>k_j} c_\ell e^{(p_j - b_\ell)/\alpha}$. While a direct evaluation requires $\mathcal{O}(MN)$ time, performing a single signed prefix/suffix construction on the barriers followed by a merge search over the token positions computes all $\bar{r}_j$ in $\mathcal{O}(M+N)$ time.

A.3 AGGREGATING TOKENS GIVEN THE BARRIERS

Aggregation repeats the same scan using the vector-valued signed weights $w_i = r_i x_i$. The stored cell index i_ℓ is reused for every barrier, requiring no second search. For each feature coordinate,

Algorithm 3 AGGREGATE – reading off every merged token, $\mathcal{O}((M+N)d)$

Require: tokens $x_{1:M}$, weights $r_{1:M}$, positions $p_{1:M}$, scale α, and $(b_{0:N}, i_{0:N})$ from Algorithm 1
Ensure: merged tokens $t_{1:N}$

$\Phi_x(b_0) \leftarrow 0, \quad \Phi_x(b_N) \leftarrow \sum_{i=1}^{M} r_i x_i$
for $\ell = 1, \dots, N-1$ **do**
  $\Phi_x(b_\ell) \leftarrow \Phi_{r\odot x}(b_\ell; i_\ell)$
end for
for $\ell = 1, \dots, N$ **do**
  $t_\ell \leftarrow N\big[\Phi_x(b_\ell) - \Phi_x(b_{\ell-1})\big]$
end for
return $t_{1:N}$

we evaluate $\Phi_{r\odot x}(b_\ell; i_\ell)$ and take the difference of consecutive values. The boundary limits are explicit: $\Phi_x(-\infty) = 0$ and $\Phi_x(+\infty) = \sum_{i=1}^{M} r_i x_i$. Multiplication by N yields the conditional expectation in Definition 1. The computational cost of the scan is $\mathcal{O}((M+N)d)$, including all feature coordinates.

Custom backward pass for aggregation. This backward pass differs from the barrier backward pass: it differentiates the readout with respect to both x and r. Signed scans represent $v = v^+ - v^-$ using the logarithms of their two non-negative components. Although a zero component is correctly handled as a dead branch in the forward pass, differentiating its $\log(0) = -\infty$ node directly can produce spurious NaNs. The implementation therefore executes the forward scan under `no_grad` and evaluates the exact Laplace-kernel products inside a custom backward pass. The resulting gradients remain finite and preserve the same $\mathcal{O}((M+N)d)$ asymptotic complexity.

The readout adjoint can be organized as follows. Let $\bar{t}_\ell$ denote the incoming gradient and define $F_\ell = \Phi_x(b_\ell)$, such that $t_\ell = N(F_\ell - F_{\ell-1})$. The corresponding cotangents are $\bar{F}_0 = -N\bar{t}_1$ (formal only, since $F_0 = 0$ is constant), $\bar{F}_\ell = N(\bar{t}_\ell - \bar{t}_{\ell+1})$ for $1 \le \ell < N$, and $\bar{F}_N = N\bar{t}_N$. The CDF difference has already been applied in these cotangents. Define the vector-valued quantity

$$g_i = \sum_{\ell=1}^{N} \bar{F}_\ell \sigma\left(\frac{b_\ell - p_i}{\alpha}\right) = N\bar{t}_N + \sum_{\ell=1}^{N-1} N(\bar{t}_\ell - \bar{t}_{\ell+1})\sigma\left(\frac{b_\ell - p_i}{\alpha}\right). \tag{21}$$

Here $\sigma((b_N - p_i)/\alpha) = 1$. Equivalently, one could use $g_i = N\sum_{\ell=1}^{N} \bar{t}_\ell[\sigma((b_\ell - p_i)/\alpha) - \sigma((b_{\ell-1} - p_i)/\alpha)]$; this form uses $\bar{t}$, not $\bar{F}$. The direct readout gradients are $\bar{x}_i \mathrel{+}= r_i g_i$ and $\bar{r}_i^{\text{direct}} \mathrel{+}= x_i^\top g_i$. All g_i are obtained by one barrier-support CDF scan with the *unreversed* argument $\sigma((b_\ell - p_i)/\alpha)$, plus the constant endpoint term. Moving a barrier contributes

$$\bar{b}_\ell = \bar{F}_\ell^\top \Phi_x'(b_\ell), \qquad \Phi_x'(b_\ell) = \frac{1}{\alpha}\sum_i r_i x_i \kappa\left(\frac{b_\ell - p_i}{\alpha}\right), \quad 1 \le \ell < N. \tag{22}$$

These cotangents enter Algorithm 2's reverse procedure, and its result is added to $\bar{r}^{\text{direct}}$. Both the CDF scan and the density scan in equation 22 take $O((M+N)d)$ work. No $M \times N$ assignment matrix is formed.

Full proof of Theorem 1. The forward construction first computes the scalar prefix/suffix states in equation 13. The monotone merge search and the closed-form root described in Appendix A.1 produce all barriers and densities in $O(M+N)$ work. Repeating the prefix/suffix construction for the d signed sequences $r_i x_{i,c}$ evaluates $F_1, \dots, F_{N-1}$ in $O((M+N)d)$ work. Setting $F_0 = 0$ and $F_N = \sum_i r_i x_i$ and differencing adjacent values gives all outputs in $O(Nd)$. Hence the forward bound holds.

For a tangent $\dot{r}$ on the simplex, Equation equation 8 is exactly a Laplace-CDF scan with signed weights $\dot{r}_i$, read at the stored barrier cells. It costs $O(M+N)$. In reverse mode, Equation equation 20 scans the signed barrier cotangents c_ℓ and reads them at every p_j, also in $O(M+N)$. For the feature readout, differentiating $r_i x_i$ and the CDF evaluations only introduces further signed scans and pointwise products. In reverse, equation 21 computes the direct derivatives with respect to r and x by one CDF scan, while equation 22 supplies the barrier cotangents by one density scan.

Applying the barrier VJP completes the chain rule. Each scan, gather, and merge is linear in its input and output lengths, and there are only a constant number of them per feature coordinate. Thus forward, JVP, and VJP all cost $O((M+N)d)$, reducing to $O(M+N)$ for fixed d. The case $N=1$ has no interior barriers and reduces directly to $t_1 = \sum_i r_i x_i$, satisfying the same bound.

Algorithmic bound and portable implementation. The stated linear-work bound assumes the positions are already ordered, as in Section 3, and that the two ordered searches are implemented as merge passes. A portable PyTorch implementation may use `sort` for unordered coordinates and `searchsorted` for each query. Those choices compute the same mathematical operator and retain matrix-free storage, but their worst-case work includes sorting and binary-search overhead. The linear bound describes the ordered scan-level construction, with the merge searches shown in Algorithms 1 and 2.

Implementation checklist. A correct implementation should satisfy six conditions: (i) recenter every decayed prefix/suffix sum as in equation 9; (ii) split signed inputs before scanning as in equation 10; (iii) locate cells from $\operatorname{logit}\Phi_r(p_i; i)$, not A_i; (iv) solve the in-cell quadratic using the numerically safe endpoint and root branch; (v) implement separate matrix-free custom backward passes for barriers and aggregation, running forward scans under `no_grad`; and (vi) maintain α proportional to token spacing rather than to the full sequence width. Appendix B.2 compares the dense, native-scan, and fused-CUDA implementations of this operator.

A.4 Scorer and routing normalization

The scorer is a lightweight module applied independently to the input token representations. In the default configuration, an MLP produces one scalar logit s_i for each token. The default LapSum soft-top-k construction is the differentiable routing mechanism introduced in the original LapSum work (Struski et al., 2025),

$$s_i = f_{\mathrm{MLP}}(x_i), \qquad i = 1, \dots, M. \tag{23}$$

LapMerge does not use the logits directly. They are converted into non-negative routing masses r_i and normalized so that $\sum_i r_i = 1$. We consider the following interchangeable routing maps:

$$r_i^{\mathrm{sig}} = \frac{\operatorname{sigmoid}(s_i)}{\sum_j \operatorname{sigmoid}(s_j)}, \qquad \text{sigmoid + normalization,} \tag{24}$$

$$r_i^{\mathrm{sp}} = \frac{\operatorname{softplus}(s_i)}{\sum_j \operatorname{softplus}(s_j)}, \qquad \text{softplus + normalization,} \tag{25}$$

$$r_i^{\mathrm{sm}} = \operatorname{softmax}(s)_i, \qquad \text{softmax,} \tag{26}$$

$$r_i^{\mathrm{stk}} = \frac{1}{k}\ \text{soft-top-k}(s)_i, \qquad \text{soft-top-}k\text{ + normalization.} \tag{27}$$

The first three maps differ in how strongly they suppress low-scoring tokens. The soft-top-k map additionally encodes the target budget: its unscaled outputs have total mass k, so division by k produces a probability mass function. Unless stated otherwise, the experiments in this paper use soft-top-k routing as the default scorer normalization. The other maps are ablations or alternative deployment choices; once the masses r_i have been formed, the same LapMerge operator and the same barrier equations apply.

The direct implementation of soft-top-k sorts the M scalar logits and therefore has theoretical cost $O(M \log M)$, followed by the linear LapMerge computation. This term is usually not visible in measured runtime: the sort acts on scalars rather than d-dimensional tokens and is implemented by highly optimized GPU primitives. It can nevertheless matter at extremely large sequence lengths. Fast LapSum (Struski et al., 2026) replaces the full sort of the scorer logits with probabilistic bracketing that narrows and sorts only a small band around the threshold, giving an expected-linear routing step. Note that this concerns the ordering of the *logit values* inside the routing module; it is unrelated to the ordering of the token positions p_i, which the merge operator assumes as given. We do not require that more elaborate routing implementation in the present experiments, because the sorting cost is negligible in our measured regime; the distinction becomes relevant for sequences substantially above 10^6 tokens. The sigmoid and softplus alternatives require only pointwise evaluation and normalization, so they do not introduce this sorting term.

Normalized masses are a specification, not the computational representation. We state the routing maps as normalized probability vectors, $r_i \geq 0$ with $\sum_i r_i = 1$, because that is the cleanest way to specify what the barrier construction of §A requires mathematically. The operator's interface accepts either form: the runtime benchmark, for instance, passes an explicitly materialized (numerically stabilized) softmax of the logits. The numerically critical part is not whether r is materialized but how the barrier solver represents its internal quantities: the search operates on log-scale node values, $\mathrm{node}_i = \mathrm{logit}\,\Phi_r(p_i; i)$ as constructed in §A.1, together with log-scale prefix/suffix accumulators, so extremely small tail contributions survive at large M/α where a plain-scale accumulation would underflow. In an optional fully log-space path the normalized masses need never be formed explicitly: $\log r_i = s_i - \mathrm{LSE}_j(s_j)$ feeds the same scans directly. Either way, the normalized-probability view of r_i above is a specification of the operator's semantics, not a claim about which tensors an implementation stores.

Configuration recipe per experiment. Table 5 collects, in one place, which scorer, routing map, budget, and widths each experiment uses, and which modules are trained. Two distinct temperatures appear throughout the paper and should not be conflated: the *routing* temperature inside soft top-k (fixed at 1 everywhere) and the *merge* kernel width α of equation 1, whose units follow the position coordinate: normalized positions in $[0, 1]$ for histopathology (so the learned width starts at the mean bin spacing $1/N$), integer rank indices for LRA and POPE, and a learned scalar coordinate for Cars and arXiv, where the scorer emits a position alongside the importance and the width is learned per lifted coordinate. The routing budget k coincides with the output budget N in every trained model; in POPE, $k = \lceil bM \rceil$ equals both the number of retained positions and the number of merged outputs.

experiment	scorer (input → arch)	budget N	merge width α	trained modules	backend
histopathology	feats+xy → MLP(1536+2,128,1)	1024	learned scalar, init $1/N$	scorer, width, head	native
LRA	token → linear(dim,1)	10% of M	fixed 1 (rank units)	scorer, backbone, head	native
Cars	token → MLP(384,128,2) → (r, a)	20	learned per-coord. + frame U	scorer, U, width, head	native
arXiv	token → MLP(768,·,2) → (r, a)	8–64	learned per-coord. + frame U	scorer, U, width, head	native
POPE	frozen VisionSelector scorer	$\lceil bM \rceil$	fixed (α_r, α_a)=(1, 1), index units	none (eval-only)	native

Table 5: **Configuration recipe per experiment.** Routing is LapSum soft top-k at temperature 1 in every row (POPE standardizes the frozen scores per image first); the runtime benchmark instead uses softmax routing, as stated in Appendix B. "Native" is the pure PyTorch implementation; the CUDA-fused backend appears only in the runtime study. Backbones are frozen except on LRA, where the reducer and the backbone are trained jointly from scratch; the row shows the configuration shared by all five LRA tasks (only the input dimensionality varies).

Notation-to-code mapping. The released code names the mathematical objects as follows: masses r_i appear as `p` (or `masses`), positions p_i as `a` (or `coordinates`), the kernel width α as `width` (in the POPE adapter, the pair (α_r, α_a) as `route_alpha`/`agg_alpha`), the barriers b_ℓ as `barriers`, and the output budget N as `n_out` (the routing budget k of soft top-k as `k`). Throughout the paper, κ denotes the kernel density and k is reserved for the routing/selection budget.

B Runtime and Large-Scale Token Reduction

This appendix gives the implementation and measurement details behind the runtime comparison in the main paper. We study the standard consecutive-barrier LapMerge operator itself, including barrier construction, aggregation, normalization, and its backward pass. The encoder and the external scorer are excluded (their measured share of the end-to-end time is reported in Table 10). Throughout, M and N denote the input and output lengths, while $B = 1$, $d = 512$, FP32, and an NVIDIA GH200 120GB are used unless noted otherwise.

B.1 What is being accelerated?

For a barrier interval i, the direct formulation assigns every input token j a Laplace mass and then forms a weighted sum of its value. It is easy to state, but materializing the resulting $N \times M$ assign-

ment map is the source of quadratic work. LapMerge uses the identity

$$\sum_j r_j e^{-|b-p_j|/\alpha} = e^{-b/\alpha} \sum_{p_j \le b} r_j e^{p_j/\alpha} + e^{b/\alpha} \sum_{p_j > b} r_j e^{-p_j/\alpha}, \tag{28}$$

and the same identity with r_j replaced by $r_j x_j$ for value-weighted quantities. The two sums are prefix and suffix summaries. Interval masses and numerators are obtained from the corresponding cumulative expressions at the two bounding barriers. Thus the same interval operator is evaluated without storing its explicit assignment map. For fixed B and d, the scan and readout therefore have work $O((M+N)d)$ and memory $O((M+N)d)$, rather than $O(MNd)$. The sequence order is already available, so this bound contains no additional sorting term.

The backward pass reuses the same factorization. Log-domain accumulation and local rescaling prevent the exponential factors in Eq. 28 from overflowing, while the scan structure keeps gradient computation linear in the two sequence lengths. This is why the method remains differentiable at large input sizes; the dense computation described below is only a faster implementation for small shapes, not a different model.

B.2 Three implementation levels of one operator

There are three useful implementation levels for the same differentiable operator. They differ in how much work is delegated to matrix kernels and how many intermediate tensors are materialized, but they use the same barriers, Laplace weights, normalization, and gradients.

1. Dense reference implementation. For small M and N, one can explicitly form the assignment weights between each input coordinate and each output barrier. The resulting dense matrix is then multiplied by the values and by a vector of ones to obtain the numerator and denominator of each output token. This is the simplest implementation to read and an important correctness oracle: it exposes every pairwise weight and is convenient for FP64 checks and gradient tests. Its cost is $O(MN)$ in work and memory, so it is unsuitable for long sequences, but it can be faster than several scan kernels when the matrix is small enough for a highly optimized GEMM or fused attention primitive.

2. Native PyTorch scan implementation. The scalable reference avoids the assignment matrix. It evaluates the two terms in Eq. 28 with log-domain cumulative scans, then gathers the scan states at the barriers and performs the final normalization. Numerators are computed by running the same summaries on $p_j x_j$, while the denominator uses p_j. In eager mode this is assembled from ordinary PyTorch operations such as `logcumsumexp`, gathers, and scatter reductions. `torch.compile` can fuse parts of this graph and reduce Python and kernel-launch overhead, although the compiler does not automatically remove every intermediate tensor. This variant is portable, easy to inspect, and works without a custom extension; its main limitation is that the framework may launch separate kernels for logically adjacent scan, readout, and backward stages.

3. Fused CUDA implementation. The performance version moves the same computation into custom CUDA kernels. It fuses coordinate transforms, prefix/suffix accumulation, barrier lookup, weighted readout, and normalization where this reduces global-memory traffic. The backward path follows the same decomposition rather than differentiating through a large materialized assignment map. The scan uses a decoupled-lookback design: thread blocks process local tiles in parallel and consume compact summaries from earlier tiles, preserving linear work while avoiding a global synchronization after every tile. This is the version used for the large GH200 measurements and the main-paper curve. It has the highest implementation complexity and requires compiling a CUDA extension, but it minimizes intermediate storage and kernel-launch overhead.

In practice, the operator dispatches between the dense path for small assignment problems and the scan path for large ones. The native and fused scan paths can use the same shape-aware decision, so this is not a comparison between different algorithms. The crossover depends on accelerator, dtype, batch size, and feature dimension; it is therefore calibrated empirically rather than presented as a universal threshold.

Backend agreement is a measured result. Table 6 reports the maximum relative forward deviation of every backend against a dense-FP64 reference on identical inputs (`bench_agreement.py`: B=2, d=8, shapes from M=7 to 131,072 including N=M and the formally admissible N>M, widths $\{0.1, 0.7, 3\}$, and both moderate and near-hard masses, the latter from logits scaled $10\times$). Three facts emerge. *(i)* The dense and scan factorizations are the same operator: in FP64 they agree to 5×10^{-8} or better everywhere, including the near-hard masses. *(ii)* All FP32 arms (dense, eager scan, compiled, and the auto dispatcher, which follows the documented rectangle/coupling rule) deviate together as the problem's conditioning worsens – up to a few percent at large M and far more under near-hard masses, where $1/\Phi'(b_\ell)$ is genuinely ill-conditioned; this is input-precision sensitivity of the problem, not backend disagreement. *(iii)* The CUDA-fused path matches to 10^{-12} on small shapes but is the one implementation with genuine limits: at large M it deviates by up to ~ 0.5 in max-relative terms even in FP64, and under near-hard masses it overflows outright (deviations of order 10^{10}–10^{14}, identical in FP32 and FP64, i.e. an algorithmic-path issue rather than precision). Accuracy experiments therefore use the native implementation throughout (Table 5), and the timing study drives all backends with moderate softmax masses. Two further boundary facts: N=M and N>M evaluate correctly in every backend, while N=1 – the trivial case $t_1 = \sum_i r_i x_i$ with no interior barriers (Appendix A) – is currently rejected by the implementations' input check rather than dispatched to this closed form.

	moderate masses		near-hard masses	
backend (vs dense-FP64)	small	large	small	large
scan-FP64	4×10^{-12}	3×10^{-11}	3×10^{-10}	5×10^{-8}
dense-FP32	2×10^{-3}	2×10^{-2}	0.16	30
scan-FP32 (eager)	3×10^{-3}	3×10^{-2}	0.37	56
auto-FP32 / compile-FP32	2×10^{-3}	3×10^{-2}	0.16	56
fused-FP64	3×10^{-13}	0.57	2×10^{-12}	3×10^{14}
fused-FP32	2×10^{-3}	0.57	0.16	3×10^{14}

Table 6: **Measured backend agreement**: maximum relative forward deviation from a dense-FP64 reference on identical inputs. "Small" covers $M \leq 4096$ with $N \leq 512$ (plus N=M, N>M cases); "large" covers $M \geq 65{,}536$ or $N \geq 1024$ at $M \geq 4096$. Gradient deviations follow the same pattern. FP32 rows degrade together with the problem's conditioning (shared input-precision sensitivity); the fused rows mark that backend's genuine limits, which the accuracy experiments avoid by using the native implementation.

Agreement on real trained-scorer masses. The grid above uses synthetic masses. As a stronger control, we captured the operator's *actual* inputs – masses, positions, and the learned width – from the trained histopathology checkpoints on all 98 test slides of both cohorts (M from 1,068 to 25,389) by wrapping the merge call, and repeated the agreement measurement on them (Table 7). On every real configuration the fused backend matches the scan-FP64 reference to 1.9×10^{-12} in FP64 and to 4.6×10^{-4} in FP32, indistinguishable from native FP32 itself (8.2×10^{-4}). The large-M deviations of Table 6 reappear only in synthetic million-token probes built by tiling real slide mass profiles (with the learned width and local mass geometry preserved): up to 0.40 max-relative at N=$M/16$, identical in FP32 and FP64. They are a property of that backend's algorithmic path at extreme shapes, not of realistic mass configurations; accuracy experiments use the native implementation throughout, and the pooler comparison in §B.4 is restated for the native path as well.

B.3 Backend ablations and dispatch calibration

To separate algorithmic scaling from implementation overhead, we measure four implementations of the same operator: explicit dense PyTorch, native PyTorch scan in eager mode, the native path under `torch.compile`, and the shape-aware dense/CUDA-fused implementation used in Figure 4. Every arm receives the same ordered coordinates $a_j = j$, normalized token masses, and FP32 features: none is charged for input sorting. We use one GH200, $B = 1$, $d = 512$, five warm-ups, 30 synchronized repeats, and a fresh process per shape. The measured backward includes gradients through values, coordinates, and score logits. First-call compilation time is recorded in the released CSVs but excluded from the steady-state curves.

	real slides (98)	tiled $\sim 10^6$, N=1024	tiled $\sim 10^6$, N=$M/16$
scan-FP32	8.2×10^{-4}	2.2×10^{-3}	8.3×10^{-2}
fused-FP32	4.6×10^{-4}	5.6×10^{-2}	4.0×10^{-1}
fused-FP64	1.9×10^{-12}	5.6×10^{-2}	4.0×10^{-1}

Table 7: **Backend agreement on real trained-scorer masses**: maximum relative forward deviation from the scan-FP64 reference (which agrees with dense-FP64 to $\leq 5 \times 10^{-8}$, Table 6), over the trained histopathology models' own (mass, position, width) inputs on all test slides, and over million-token probes tiled from the same profiles. Gradient deviations follow the same pattern. Identical fused-FP32/FP64 values on the tiled probes mark the algorithmic-path limit already documented above.

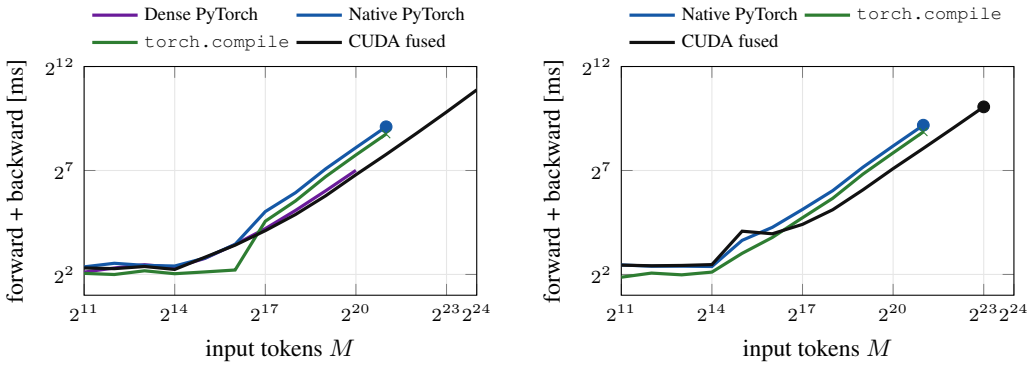

Figure 5: Matched GH200 forward–backward times for the same LapMerge operator at $B = 1$, $d = 512$, FP32. Coordinates are already ordered. Each point is the median of 30 synchronized repeats after five warm-ups. The dense path was only attempted to $M = 2^{20}$. A filled endpoint marks the last success before OOM; a cross marks the last success before a compiler failure. Lines are measured points, not extrapolations.

We sweep powers of two from $M = 2^{11}$ through 2^{24} in two regimes: $N = 1024$ isolates input-length growth, whereas $N = M/16$ grows both dimensions. The explicit dense arm was intentionally capped at $M = 2^{20}$; the other implementations were attempted through 2^{24}. Failed cells are reported as failures, not assigned an extrapolated latency. Figures 5 and 6 show the full matched time and incremental-memory measurements.

The dense path forms an explicit $M \times N$ assignment matrix and is a useful correctness oracle. Its $O(MN)$ work and storage can nevertheless be competitive at modest sizes. At fixed $N = 1024$, both dense and scan paths are formally linear in M; the striking difference is the memory constant. When $N = M/16$, the dense work instead grows quadratically while the scan retains $O((M+N)d)$ work and storage. This is the decisive regime for the scaling claim, rather than an apparent slope difference at fixed N.

The crossover is visible rather than assumed. At $N = 1024$ and $M = 2^{16}$, the fused dispatcher still uses its dense branch; it takes 10.61 ms versus 10.86 ms for forced dense. When M doubles to 2^{17}, the dispatcher switches to the CUDA scan: incremental allocation falls from 2.38 to 0.33 GiB even as the input grows, and latency remains close to the dense reference (17.21 versus 18.40 ms). This discontinuity is an implementation switch, not a change to the LapMerge operator. For $N = M/16$ the branch changes earlier, between $M = 2^{14}$ and 2^{15}, because the output length also grows.

At $M = 2^{20}$, Table 8 shows that fused CUDA is $2.47\times$ faster than eager native PyTorch for fixed N and $2.13\times$ faster for $N = M/16$; its peak incremental allocation is 2.53 and 3.64 GiB, respectively. Below the crossover, `torch.compile` can be the fastest path, so fusion is not presented as a universal small-shape win. From $M = 2^{20}$ to 2^{23}, an eightfold input increase raises fused latency by $8.10\times$ at fixed N and $7.84\times$ at fixed ratio. Peak allocation grows by approximately eightfold in both regimes. This is the empirical signature expected of the scan/readout factorization, although finite-range timings alone do not prove an asymptotic bound.

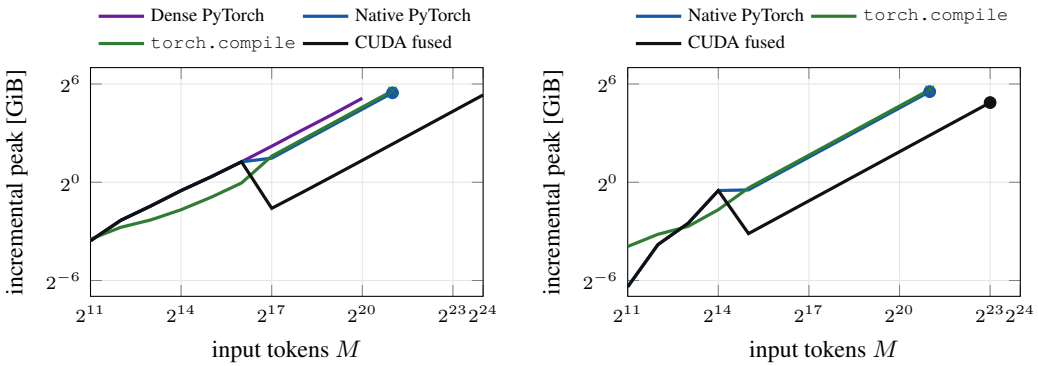

Figure 6: Incremental peak CUDA allocation for the same matched sweeps. Allocation is measured above the already live input tensors, not as total device occupancy. At fixed N, the dense path also grows linearly with M but has a much larger constant; $N = M/16$ exposes the different dependence on both sequence dimensions. Endpoint symbols have the meaning of Figure 5.

	$N = 1024$		$N = M/16$	
Implementation	time [ms]	peak [GiB]	time [ms]	peak [GiB]
Explicit dense	128.52	35.03	—	—
Native PyTorch	274.30	22.05	289.54	23.16
`torch.compile`	214.76	24.05	230.13	25.16
CUDA fused	**111.04**	**2.53**	**136.09**	**3.64**

Table 8: Matched GH200 measurements at $M = 2^{20}$, $B = 1$, $d = 512$, FP32. Time is steady-state forward plus backward. Peak is incremental CUDA allocation above live inputs. The explicit dense reference was not swept at $N = M/16$; its omission is a scope limit, not an OOM result.

The native eager runs are out of memory from $M = 2^{22}$ in both regimes. `torch.compile` also reaches 2^{21}, but its larger cells fail in Triton code generation because a decoupled-lookback temporary is inferred as both `int64` and `int32`; this is a compiler failure, not a LapMerge OOM. The fused implementation reaches $M = 2^{24}$ at fixed N and $M = 2^{23}$ at fixed ratio, where the next attempted point is OOM.

B.4 Matched comparison with learned poolers

We compare the same shape-aware LapMerge dispatcher with TokenLearner and PatchMerger. The latter two use scaled dot-product attention (SDPA) to compute their published normalized pooling maps without storing an $M \times N$ matrix; this is a memory-efficient rewrite, not a change to either pooler's output. Concretely, the implemented variants are: PatchMerger = LayerNorm followed by a bias-free $d \times N$ projection, softmax over tokens; TokenLearner = LayerNorm, Linear$(d, 64)$, GELU, Linear$(64, N)$, softmax over tokens; the SDPA forms pass the projection weights as queries with an explicit scale=1, so the bias (constant over positions) cancels in the softmax. Equivalence is verified, not assumed: in FP64 the published and SDPA forms agree to $\leq 10^{-14}$ relative error in the forward pass and in every gradient across shapes up to M=4096 (`test_sdpa_equivalence.py`); in FP32 the SDPA kernel introduces a $\sim 3 \times 10^{-4}$ relative deviation, a backend-level numerical effect, not a mathematical difference. These statements cover this implementation, not every published TokenLearner variant.

All three reducers use the GH200, FP32, $B = 1$, $d = 512$, five warm-ups, 30 synchronized repeats, a fresh process per shape, and a 600-second whole-cell limit. The input shape, mean-square loss, full backward pass, and timing rules are matched, but the parameterizations are not identical: LapMerge receives score logits and times their normalization, barriers, and readout, while the poolers also time their built-in scoring/projection layers. External encoders, LapMerge's task-specific logit predictor, and downstream networks are excluded.

Reducer	$N = 1024$ time [ms]	$N = 1024$ peak [GiB]	$N = M/16$ time [ms]	$N = M/16$ peak [GiB]
LapMerge (fused)	**111.04**	**2.53**	**136.09**	**3.64**
TokenLearner (SDPA)	336.16	4.29	7158.16	4.29
PatchMerger (SDPA)	453.95	4.02	14750.05	4.38

Table 9: Matched reducer measurements at $M = 2^{20}$ on one GH200, $B = 1$, $d = 512$, FP32. Times include forward, mean-square loss, and backward; peaks are incremental CUDA allocations above live inputs. Each cell is a median of 30 synchronized repeats after five warm-ups.

Table 9 illustrates the difference between the two regimes. These are reducer-core speedups, not whole-system ones: at fixed N, the LapMerge core is $3.03\times$ faster than TokenLearner-SDPA and $4.09\times$ faster than PatchMerger-SDPA. When $N = M/16$, the gaps become $52.6\times$ and $108.4\times$, respectively: SDPA avoids a materialized assignment map but still performs MN content-to-output interactions. Because the fused backend has the numerical limits documented in Table 6, we also state the same comparison for the always-faithful native implementation (Table 8): at $N = M/16$ it takes 289.54 ms, i.e. $24.7\times$ and $50.9\times$ faster than the two poolers – the order-of-magnitude advantage in the growing-output regime does not depend on the fused path. At fixed $N = 1024$ the native core (274.30 ms) is $1.2\times$/$1.7\times$ ahead, and the further $2.47\times$ there is what fusion buys. The memory comparison in Table 9 is more modest, because the optimized poolers also avoid the quadratic activation map. The large advantage here is primarily in computation, not a claim that SDPA is quadratic in stored activations.

Both SDPA poolers time out at the next attempted ratio point, $M = 2^{21}$; LapMerge continues through $M = 2^{23}$ in that regime. At fixed $N = 1024$, PatchMerger times out at $M = 2^{21}$. TokenLearner completes that point, but its median jumps from 0.336 s at $M = 2^{20}$ to 14.93 s at $M = 2^{21}$, then the $M = 2^{22}$ run fails with PyTorch's `value.stride(2) overflows` error. The jump and error are reported as observed behavior of this SDPA implementation, not attributed to a particular kernel switch without a separate diagnostic. A timeout is not a measured latency at the missing point.

Scorer and routing overhead (end-to-end control). The main comparison times the reducer core with precomputed token masses. As a control, Table 10 measures the full LapMerge module end to end – a lightweight scorer (Linear$(512, 256)$+GELU+Linear$(256, 1)$), LapSum soft top-k routing, and the merge – under the same protocol (GH200, B=1, d=512, FP32, forward+backward, five warm-ups, 30 repeats). The scorer grows linearly with M and the routing cost is dominated by a nearly constant sort overhead, so together they add about 11% of the fused end-to-end time at M=2^{20} and about 7% at M=2^{23}; only at small M (2^{14}) is the routing's fixed cost comparable to the merge itself. The reducer-core comparison above therefore does not hide a dominant upstream cost.

M	N	scorer	routing	merge (eager)	merge (fused)	total (fused)
2^{14}	1024	0.8	5.7	6.2	6.1	10.4
2^{17}	1024	1.3	5.6	33.3	18.5	24.7
2^{17}	8192	1.3	5.6	35.8	22.8	26.0
2^{20}	1024	7.8	5.3	274.2	111.0	122.5
2^{20}	65536	7.8	5.2	291.3	135.6	147.1
2^{23}	1024	60.8	7.2	–	898.3	979.8
2^{23}	524288	61.1	7.3	–	1059.4	1141.2

Table 10: **End-to-end decomposition of the full LapMerge module** (median ms, forward+backward, GH200, B=1, d=512, FP32). Components are timed separately; *total* runs the whole scorer→routing→merge chain with one backward. Dashes: the eager merge at M=2^{23} exceeds memory in this control harness, which keeps extra per-component buffers; the main benchmark covers that point.

Table 11: **Historical published-versus-SDPA control on one GH200 (ms).** All methods use $B = 1$, $d = 512$, and FP32. Published PatchMerger and TokenLearner materialize their $M \times N$ assignment maps. Their SDPA variants are mathematically equivalent memory-efficient implementations that do not materialize these maps. LapMerge uses its shape-aware dense/fused-CUDA dispatcher. Entries are warmed medians over 20 iterations for the competitors and 100 iterations for LapMerge.

		PatchMerger		TokenLearner		**LapMerge**
M	N	published	SDPA	published	SDPA	fused hybrid
65,536	1,024	120.22	28.01	118.52	20.64	**10.86**
65,536	2,048	123.81	37.45	123.38	22.55	**15.10**
262,144	1,024	475.24	110.60	473.75	80.46	**29.51**
262,144	2,048	493.98	151.90	490.19	89.33	**29.64**
1,048,576	1,024	1932.20	456.79	1918.33	322.97	**112.61**
1,048,576	2,048	2019.88	604.09	1999.63	363.95	**112.04**

Memory-efficient baseline construction. The SDPA implementations compute exactly the same normalized pooling as their published counterparts. PatchMerger's learned projection columns are the queries, normalized input tokens are keys, and the original tokens are values. For TokenLearner, the final linear layer of its scoring MLP provides the queries and the preceding hidden features provide the keys; its final bias is constant over input positions and therefore cancels inside the softmax. SDPA removes the explicit quadratic activation map but retains the $\mathcal{O}(MNd)$ arithmetic. Each cell in this older control ran in an isolated process with a 120-second wall-clock limit. No SDPA cell in that control reached this limit; both published dense implementations ran out of memory at $(M, N) = (1{,}048{,}576, 4{,}096)$. The memory ordering observed in this older run does not describe the newer fused LapMerge sweep reported in Table 9.

Table 11 is an older, separate control showing how much the published dense poolers benefit from mathematically equivalent SDPA rewrites. It used 20 warmed iterations for the poolers and 100 for LapMerge; we do not compute cross-method speedup factors from that table. Figure 4 and Table 9 instead use the matched 5/30 rerun.

B.5 Scaling and stress tests

Table 12 is a separate, earlier H100 stress test of feature dimension at $N = 0.1M$. It is useful context for the cost of value aggregation, but it uses another accelerator and timing protocol and is not spliced into the GH200 curves of Figure 4 or the matched backend ablation above.

Table 12: Independent H100 (80 GB) dimensional stress test of the complete reducer, forward plus backward, at $N = 0.1M$. Times are warmed medians in milliseconds from a `torch.compile` run. This is not the matched GH200 protocol; the final column shows the step ratio at $d = 512$.

M	$d = 512$	$d = 1024$	$d = 2048$	step ratio ($d = 512$)
10^3	4.8	4.7	7.1	—
10^4	5.9	6.7	10.7	1.2×
10^5	22.0	40.4	75.7	3.7×
10^6	222.5	433.5	OOM	**10.1**×
$3 \cdot 10^6$	688.3	·	·	3.1×
10^7	OOM	·	·	—

The main-paper curve now uses the ordered-coordinate, shape-aware dense/CUDA-fused LapMerge path. At fixed $N = 1024$, it completes both directions at $M = 2^{24}$. At $N = M/16$, it completes through $M = 2^{23}$ and is out of memory at the next attempted point. Table 13 gives the measured forward and forward–backward values at the large end of both sweeps. In particular, we do not infer a forward-only time for the failed $M = 2^{24}, N = M/16$ cell.

Regime	M	N	Forward [ms]	Forward + backward [ms]	Peak [GiB]
$N = 1024$	2^{22}	2^{10}	169.09	439.28	10.07
$N = 1024$	2^{23}	2^{10}	360.85	899.09	20.13
$N = 1024$	2^{24}	2^{10}	807.38	1878.61	40.24
$N = M/16$	2^{22}	2^{18}	150.64	531.17	14.57
$N = M/16$	2^{23}	2^{19}	299.83	1067.26	29.14
$N = M/16$	2^{24}	2^{20}	—	OOM	—

Table 13: Large-shape GH200 probes for the CUDA-fused LapMerge dispatcher, with ordered inputs, $B = 1$, $d = 512$, and FP32. The failed ratio cell has no reported latency or allocation; no forward-only measurement is inferred. Peak allocation is incremental above live inputs.

B.6 Measurement protocol and interpretation

The new GH200 backend sweeps use five warm-ups and 30 synchronized repeats per shape, including a mean-square loss and backward pass. Each point runs in a fresh process with a 600-second whole-cell limit. First-call/JIT time is logged separately and excluded from steady-state medians; timing IQRs, incremental peak CUDA allocation, and failure statuses are retained in the CSV files. The fixed-output and fixed-ratio sweeps use identical inputs and the same protocol across the four LapMerge implementations. Allocation is measured relative to already live tensors, so it is not total device occupancy.

The earlier H100 dimension sweep and the published-versus-SDPA table use different iteration counts and are intentionally separate from the matched GH200 figures. The new competitor sweep shares the 5/30 timing settings and 600-second whole-cell limit with the LapMerge sweeps. A timeout includes warm-up and all repeated measurements; it is not an observed steady-state latency at the missing point. Open markers in Figure 4 sit on the last successful point before a timeout; filled markers precede OOM or an execution failure. Neither marker should be read as a measurement of the failed cell.

C When Does Merging Beat Selection?

This appendix isolates the scientific question that is distinct from runtime: when is it better to preserve information by merging tokens rather than to discard all but the highest-scoring ones? We use matched output budgets on Stanford Cars and arXiv. The two settings deliberately differ in how strongly task-relevant information follows sequence order: spatially adjacent patches on Cars versus distributed evidence in long documents. Runtime and implementation choices are reported separately in Appendix B; the tables below therefore focus on downstream accuracy and on the structure of the reducer. The section closes with a direct order intervention that isolates the mechanism (§C.4) and with the list of planned confirmation experiments (§C.5).

C.1 Matched evaluation protocol

All rows use the same upstream representation, head, optimizer, and output budget within a dataset. "Soft" means that the differentiable reducer is present during evaluation. "Hard" removes the reducer and retains the N highest-scoring raw tokens, which is a useful control for distinguishing the scorer from the aggregation operator. A one-sweep model uses one ordered LapMerge pass and returns N tokens. The two-sweep model uses two independent passes of 10 tokens each, followed by concatenation; its total output budget is therefore also 20, but its reducer cost is approximately doubled. This distinction is made explicit in the Cars table rather than hidden in the parameter count.

The Cars numbers below are single-run screening results from the established training protocol; the planned multi-seed confirmation is described after the tables. The arXiv comparison is budget-matched: every arm is trained at the budget at which it is evaluated. This is intentionally conservative for cross-budget generalization, but it means that the arXiv table should be read as a comparison of operating points, not as evidence that one shared model dominates all budgets.

C.2 SPATIALLY ORDERED TOKENS: STANFORD CARS

Features are the 196 patch tokens of a frozen ViT-S/16 encoder (`timm`'s `vit_small_patch16_224`, ImageNet-pretrained) at 224^2 resolution; the reducer and classification head are trained on top. On Stanford Cars (Table 15), LapMerge improves over top-N selection at the same 20-token budget. A single sweep reaches 75.46% versus 74.10% for selection, while two complementary sweeps reach 76.35%, within 0.36 points of the 76.71% dense baseline. The learned poolers perform worse: TokenLearner reaches 73.60% and PatchMerger 71.83%. Thus, on this spatially structured task, retaining information through merging is more effective than discarding tokens, while the particular structure of LapMerge provides an additional advantage over generic pooling. Removing the operator at inference by retaining the highest-scoring raw tokens reduces accuracy to 67.88%. We read this as a *head-dependence test*: the classifier was trained on merged representations, so the drop measures how strongly the trained head relies on them – it is not the accuracy of an equivalently trained hard-selection model, which reaches 74.10%.

The Cars result supports the ordered-merge hypothesis: neighboring visual patches often contain complementary evidence, so replacing a group by its mass-preserving representative can retain information that hard selection throws away. The two-sweep result should not be interpreted as a free increase in capacity, however, since it uses two reducer passes. The relevant control is consequently the one-sweep row at the same output budget.

Budget dose-response. Repeating the one-sweep-versus-selection comparison at $N \in \{10, 20, 40\}$ with three seeds each (Table 14) shows that the merge advantage is a tight-budget phenomenon: $+3.4$ points at $N{=}10$, $+0.9$ at the capacity-matched $N{=}20$, and -0.8 at $N{=}40$, where selection alone already retains enough evidence (at 77.3 it even edges past the 196-token dense ceiling of 76.71). This mirrors the POPE pattern – decisive at the 10% budget, within noise at 30% (§D.3): merging pays where compression is aggressive. The operator-free column shows the same trend from the head's side: its reliance on the merge fades as the budget grows (soft$-$hard gap $15.4 \to 7.6 \to 3.3$).

	LapMerge one sweep (soft)	operator-free (hard)	Top-N selection
$N{=}10$	$\mathbf{74.91{\pm}0.60}$	$59.51{\pm}0.66$	$71.47{\pm}0.31$
$N{=}20^{\dagger}$	$\mathbf{75.10{\pm}0.28}$	$67.53{\pm}0.26$	$74.20{\pm}0.44$
$N{=}40$	$76.52{\pm}0.40$	$73.25{\pm}0.27$	$\mathbf{77.28{\pm}0.13}$

Table 14: **Budget dose-response on Stanford Cars** (top-1, mean $\pm$sd over three shared seeds, original protocol). †The $N{=}20$ row is the capacity-matched frozen-frame configuration of Table 15; the full one-sweep single run reaches 75.46 there.

Figure 7 visualizes the routing mass learned by the reducer on two test images: mass concentrates on discriminative car parts, while background patches receive near-zero mass, mirroring the attribution behaviour measured quantitatively in histopathology (Appendix E.1).

C.3 DISTRIBUTED EVIDENCE AND ORDER ABLATIONS

Protocol. The arXiv task is 11-way topic classification on the public `ccdv/arxiv-classification` dataset with its released train/test split. Features come from a frozen `allenai/longformer-base-4096` checkpoint with its own tokenizer, truncated to $M{=}1024$ tokens; the classifier is a linear head on the reduced sequence. Every arm trains for 30 epochs with AdamW (lr 3×10^{-3}, weight decay 0.2, batch 64) at its own output budget $N \in \{8, 16, 32, 64\}$.

The picture changes on arXiv (Table 16). LapMerge exceeds top-N selection by 5.6–6.2 points across all evaluated budgets, but does not consistently outperform learned poolers. Across eight budget-by-method comparisons, the poolers lead in five, LapMerge leads in two, and one comparison is tied; the largest gap in either direction is 1.0 point. These are single-seed runs, and the only measured spread ($\sigma \approx 0.46$, from three independent trainings of one arm at one budget) does not estimate the variance of differences between other arms, so we attach no significance claim. We



Figure 7: **Learned routing mass on Stanford Cars.** Each pair shows the input image (left) and the per-token routing mass r_i of the trained reducer as a heatmap (right; normalized per image). Mass concentrates on discriminative car parts – grille, badge, and headlight region – while background patches receive near-zero mass. Left pair: the standard 196-token single-crop configuration of Table 15; right pair: a 2×2 multi-crop variant (784 tokens) showing the same behaviour on a finer grid.

Reducer	Tokens to head	Params	Reducer cost	Soft (%)	Hard (%)
Dense transformer, no reduction	196	—	none	76.71	—
Top-N selection	20	52 k	$\mathcal{O}(Md)$ + top-N	74.10	**74.10**†
TokenLearner	20	53 k	$\mathcal{O}(MNd)$	73.60	n/a
PatchMerger	20	8 k	$\mathcal{O}(MNd)$	71.83	n/a
ToMe post-encoder merge	20	0	$\mathcal{O}(M^2d)$	69.87	n/a
LapMerge, two sweeps	2×10	0.20 M	$2\mathcal{O}((M+N)d)$	**76.35**	36.02
LapMerge, one sweep	20	0.20 M	$\mathcal{O}((M+N)d)$	75.46	67.88
LapMerge, one sweep, frozen frame	20	52 k	$\mathcal{O}((M+N)d)$	75.10±0.28	67.53±0.26
Top-N selection (same seeds)	20	52 k	$\mathcal{O}(Md)$ + top-N	74.20±0.44	74.20±0.44

Table 15: **Matched pre-head reducers on Stanford Cars.** "Reducer params" excludes the 2.44 M shared by every row; "reducer cost" is analytic. *Soft* runs the reducer as trained; *hard* keeps the N highest-scoring raw tokens and removes it entirely (a head-dependence test, not a hard-trained model). †Selection coincides on both by construction; "n/a" marks poolers, which expose no operator-free path. The ToMe row reaches 20 tokens by iterating bipartite matching roughly four times with no transformer layer in between, well outside its published schedule; read it as training-free similarity merging pushed out of its design regime. The last block is the capacity-matched control (mean±sd over three shared seeds): the read-out frame U is frozen at identity, so the reducer matches selection's $\approx$ 52k parameters while sharing its scorer – the merge advantage survives equalized capacity. Protocol in Appendix C.1.

state plainly that LapMerge shows no consistent accuracy advantage over learned pooling in this setting; it provides the same accuracy range while retaining its $\mathcal{O}((M+N)d)$ cost.

Together, the results separate two effects of merging. Aggregating tokens instead of discarding them is particularly valuable when relevant information is distributed across the sequence, while the additional benefit of LapMerge's ordered, equal-mass aggregation appears when sequence order itself carries task structure. On Cars, this yields a 1.86-point improvement over the strongest learned pooler; on arXiv, the corresponding advantage disappears. A train-budget/test-budget robustness matrix is left as a future control; since such an evaluation runs the model away from its training budget, we would treat it as a robustness analysis rather than as an additional matched merging-versus-selection comparison.

C.4 Order intervention: permuting the scan on LRA-Image

The order is the mechanism, and permuting it is the test. The comparison so far rests on a contrast *between* datasets – LapMerge wins on a patch grid (§C.2) and does not on a topic label (§C.3) – which changes modality, backbone, length and task at once and then credits one of them.

Reducer (arXiv, frozen Longformer)	Reducer	Output budget N for $M = 1024$			
every arm trained at its own budget	params	8	16	32	64
Dense (no reduction)	—		87.6		
TokenLearner (Ryoo et al., 2021)	102 k	**88.28**	87.88	**88.12**	**87.68**
PatchMerger (Renggli et al., 2022)	14 k	88.00	**88.28**	87.08	87.48
Top-N selection	100 k	81.48	81.44	82.12	81.80
LapMerge (scan coordinate)	0.75 M	87.28	87.68	87.72	**87.68**
LapMerge (published coordinate)	0.69 M	86.24	86.80	87.24	86.48

Table 16: **Budget-matched training on arXiv** (single seed, soft deploy): every entry is trained at the budget it is evaluated at, which removes the extrapolation penalty the poolers would otherwise pay and is conservative for us. The scan coordinate is the parameterization of §3 and is what the text calls LapMerge; the published-coordinate row is an ablation. For scale only: three independent trainings of the published-coordinate arm give $86.24/85.84/86.76$ at $N{=}8$, i.e. $\sigma \approx 0.46$ for that one arm; all other entries are single runs. Against LapMerge the poolers lead in five of eight cells, trail in two and tie in one, with a largest gap of 1.0 point.

Here it is an intervention instead. We permute only the scan coordinate the barriers walk, applied after the embedding so that every token keeps its content and its positional encoding, and hold the rest of the recipe fixed; the sole thing that changes is which tokens share a bin. (This ablation uses its own shorter training protocol, so absolute numbers are not directly comparable to Table 3.) Over three seeds, three locality-preserving orders stay within 3.5 points of one another – raster $48.0{\pm}1.3$, column-major $45.4{\pm}0.1$, a Hilbert curve $48.9{\pm}0.3$ – so nothing here is special about the raster axis specifically. A random permutation, by contrast – one draw, fixed for the whole run, so the Transformer is free to learn the relabelling – costs 15.5 points ($\rightarrow 32.5{\pm}0.9$) and lands the merge *inside* the order-free band it otherwise beats by ten.

Three controls hold that reading up, built on different mechanisms so that a leak in the harness would have to fool all of them: TokenLearner, PatchMerger and learned top-N selection each return their own number under the permutation, unchanged to two decimals in both mean and spread across the same three seeds (Table 17). So the permutation reaches only reducers that walk the scan – not the data, the positions, or anything else. What it reaches, it reaches hard, and not only LapMerge: uniform pooling, whose segments are equally positional, loses 15.7. ToMe does not, at $+0.2$, because its bipartite matching groups by content and the scan order it is handed is arbitrary to begin with. The intervention therefore isolates something narrower, and more useful, than "the merge needs an order": what the order buys is the advantage a positional partition holds over content-based grouping, and taking it away takes away exactly that.

reducer	raster	col-major	Hilbert	rnd (fixed)	rnd (per img)
LapMerge	$48.01{\pm}1.34$	$45.42{\pm}0.14$	$48.96{\pm}0.26$	$32.48{\pm}0.89$	$35.36{\pm}0.80$
uniform pooling	$45.15{\pm}0.27$	$43.67{\pm}0.45$	$46.71{\pm}1.14$	$29.46{\pm}0.82$	$28.11{\pm}0.28$
ToMe	$37.16{\pm}0.24$	$37.33{\pm}0.45$	$37.67{\pm}0.16$	$37.35{\pm}0.33$	$42.85{\pm}0.31$
TokenLearner	$37.08{\pm}0.34$	—	—	$37.08{\pm}0.34$	—
PatchMerger	$26.94{\pm}0.34$	—	—	$26.94{\pm}0.34$	—
learned top-N selection	$37.20{\pm}0.88$	—	—	$37.20{\pm}0.87$	—

Table 17: **Permuting the scan, with everything else held fixed.** LRA-Image at a 10% budget, mean$\pm$sd over three seeds, sharp width. Down the top block: a locality-preserving order is a locality-preserving order – raster, its transpose and a Hilbert curve agree to within 3.5 points – while a random permutation costs the merge 15.5, dropping it *into* the band of the reducers below it. Across the bottom block: those reducers cannot see the scan coordinate, and they return their own numbers to two decimals in both mean and spread, which rules out the permutation leaking through the data, the positions, or the harness. A per-image permutation is gentler than a fixed one for the merge (-12.7 against -15.5) and actually helps ToMe ($+5.7$): resampling the order every step acts as augmentation, which is why the fixed draw – one the Transformer is free to learn – is the cleaner intervention.

Two dose-response sweeps (Table 18) say what kind of dependence it is. The gap is flat across a tenfold range of budgets, 14 to 16 points from 5% to 50%, so a looser budget does not dilute it. And it shrinks as the kernel widens past the bin spacing, from 15.0 at α=1 to 11.2 at α=M/N and 3.9 at α≈M, where the operator is close to order-free mean pooling and the raster arm has fallen into the order-free band – from the raster side alone, the way a positional mechanism should, since a bin that blurs across its own boundaries partitions less.

sweep	value	raster	permuted (fixed)	gap
width α (budget 10%)	0.5	48.0±1.2	33.2±0.7	14.8
	1	48.0±1.1	33.0±0.9	15.0
	2	47.2±0.5	32.3±0.9	14.9
	5	44.4±0.1	31.5±1.0	12.9
	$10 = M/N$	42.6±0.2	31.4±1.2	11.2
	30	41.5±0.2	31.4±0.8	10.2
	100	40.5±0.1	31.9±0.7	8.6
	$1000 \approx M$	36.5±0.2	32.7±0.9	3.9
budget ($\alpha = 1$)	5%	45.2±0.3	31.0±0.7	14.2
	10%	48.8±0.8	32.8±0.6	16.0
	20%	49.1±1.0	33.2±1.1	15.9
	50%	48.4±0.9	33.3±0.5	15.1

Table 18: **How the order effect responds to width and to budget.** LRA-Image, LapMerge, mean±sd over three seeds; every row is the raster scan against one random permutation of the scan coordinate, fixed for the run, with everything else as in Table 17; the gap is the difference of means. Across the width it is not flat: sharp kernels ($\alpha \leq 2$) hold the full gap, which shrinks as α grows toward M, where the raster arm itself falls to the order-free level (the permuted arm stays between 31 and 33 throughout) – exactly what a positional mechanism should do. The 10%, α=1 configuration appears in both blocks with the same three seeds and returns 48.0±1.1 and 48.8±0.8 (permuted 33.0 and 32.8): the difference is run-to-run nondeterminism of the GPU scatter, about the size of the seed spread, so differences under a point between rows are not to be read.

One knob matters, and its natural default is the wrong one. Everything else we varied moved the result by less than seed noise; the bin width did not. Over three seeds, α=0.5 and α=1 tie at 48.0, α=2 gives 47.2, α=5 gives 44.4, and setting α to the mean bin spacing $M/N \approx 10$ – the value the construction seems to invite, since it makes the kernel as wide as a bin – costs five to six points at 42.6±0.2, and a kernel as wide as the sequence ($\alpha \approx M$) another six, to 36.5 (Table 18). Wide bins blur across their own boundaries and the partition stops being one; sharper than 1 buys nothing, so the sharp setting is a plateau, not an edge.

The speed comes from the scan, not from computing something weaker. A *dense* implementation of the same operator – same barriers, same read-out, coupling materialized – reaches 48.85% at 90.1 s/epoch against 48.90% at 12.9 s/epoch for the matrix-free form (both single seed, same job): identical accuracy, seven times the speed. So the linear-time machinery of §4.1 is what makes the reduced model cheaper than the unreduced one, and it shows up here on a downstream task, not only in a microbenchmark.

C.5 Planned confirmation experiments

One control remains planned, and two have been carried out. First, the one-sweep-versus-selection comparison has been repeated at three output budgets with three seeds each (Table 14); extending the multi-seed protocol to the two-sweep and learned-pooling rows remains future work. Second — still planned — the order-permutation control of §C.4 – carried out on LRA-Image – will be extended to the Cars and arXiv studies at their own budgets, with the permutation applied before both scoring and merging while the backbone, optimization schedule, and output budget remain unchanged. A third control, equalizing reducer capacity, has been carried out (Table 15, last block): with the read-out frame frozen at identity the LapMerge reducer matches selection's ≈ 52k parameters and still leads, 75.10 ± 0.28 against 74.20 ± 0.44 over three shared seeds.

For completeness, the implementation ablation is reported separately in Appendix B.3: it compares eager PyTorch, `torch.compile`, and the CUDA-fused hybrid, and includes forced-dense, forced-scan, and automatic dispatch. Thus the accuracy tables here answer whether merging helps, while the runtime appendix answers how the same operator is evaluated efficiently.

D Drop-in Token Reduction in a Frozen VLM: Details

This appendix complements Section 4.3. We describe the pipeline into which score-routed LapMerge is inserted, state its general two-width form with properties and cost (Appendix D.2), and report the sweep over both widths (α_a, α_r), with the released selection pipeline rerun under identical conditions. The POPE benchmark itself is described in Appendix E.4.

D.1 Pipeline

Backbone. We use Qwen2.5-VL-7B-Instruct (Bai et al., 2025) without any fine-tuning. Qwen2.5-VL encodes an image at its native resolution with a ViT, and a 2×2 patch merger then groups neighbouring patches, so the number of visual tokens M passed to the language model depends on the input resolution. Visual tokens carry multimodal rotary position embeddings (M-RoPE) indexed by their (temporal, height, width) coordinates.

VisionSelector. We take the publicly released VisionSelector pipeline (Zhu et al., 2025) as-is. It attaches a lightweight learned scorer to the frozen VLM, which assigns an importance score s_i to every visual token x_i, $i = 1, \dots, M$. The scorer is trained once by the VisionSelector authors and is not retrained by us. At a budget $b \in \{10\%, 30\%\}$, VisionSelector keeps the $k = \lceil bM \rceil$ highest-scoring tokens, discards the rest, and feeds the kept tokens to the language model at their original positions.

D.2 Score-routed LapMerge: general form, properties, and cost

Section 3 presents LapMerge with a single kernel width. The POPE deployment uses the natural two-width generalization, which we state here explicitly. Let $r_i \geq 0$ be the routing masses (normalized to sum to one; the soft top-k output sums to k, which only rescales them), placed at the raster indices $p_i = i$, and let Φ^α denote the mixture CDF equation 1 at kernel width α. With $N{=}k$ outputs, the barriers solve the equal-mass equations at the *routing* width, $\Phi^{\alpha_r}(b_\ell) = \ell/N$ ($1 \leq \ell < N$, $b_0 = -\infty$, $b_N = +\infty$), and each interval is read out at the *read-out* width α_a with its own normalization:

$$w_i^\ell = \frac{r_i\big[\sigma\big(\frac{b_\ell - p_i}{\alpha_a}\big) - \sigma\big(\frac{b_{\ell-1} - p_i}{\alpha_a}\big)\big]}{\Phi^{\alpha_a}(b_\ell) - \Phi^{\alpha_a}(b_{\ell-1})}, \qquad t_\ell = \sum_{i=1}^{M} w_i^\ell x_i = \frac{\Phi^{\alpha_a}_{r\odot x}(b_\ell) - \Phi^{\alpha_a}_{r\odot x}(b_{\ell-1})}{\Phi^{\alpha_a}(b_\ell) - \Phi^{\alpha_a}(b_{\ell-1})}. \tag{29}$$

The explicit denominator is needed because, for $\alpha_a \neq \alpha_r$, the read-out kernel does not assign mass $1/N$ to the intervals cut at the routing width, so the standard formula $N[\Phi_x(b_\ell) - \Phi_x(b_{\ell-1})]$ would no longer be a weighted average.

Properties. *(i) Exact budget*: exactly N outputs for any widths. *(ii) Convexity*: $w_i^\ell \geq 0$ and $\sum_i w_i^\ell = 1$ by construction, so every t_ℓ is a convex combination of the inputs. *(iii) Reduction to the standard operator*: for $\alpha_a = \alpha_r$ the denominator equals $1/N$ for every interval and equation 29 coincides with equation 4; in particular the default $(\alpha_r, \alpha_a) = (1, 1)$ used for Table 2 *is* the standard operator. *(iv) Selection limit*: as both widths vanish, the soft top-k masses harden to the top-k indicator, each interval contains exactly one kept token, and t_ℓ equals that token. *(v)* The first-moment identity $N^{-1}\sum_\ell t_\ell = \sum_i r_i x_i$ holds only for $\alpha_a = \alpha_r$; for other widths the outputs are normalized per interval rather than mass-conserving.

Cost. The barriers are the standard equal-mass solve at width α_r (Algorithm 1, $\mathcal{O}(M + N)$). Numerator and denominator of equation 29 are Laplace-CDF scans at width α_a with weights $r_i x_i$

and r_i, queried at the already ordered barriers – the construction of Algorithm 3 with a second set of prefix/suffix states. The variant therefore has the same $\mathcal{O}((M+N)d)$ forward and backward cost as the standard operator, up to a constant factor of about two. The POPE adapter used for our experiments evaluates equation 29 with a dense reference implementation (a per-quantile barrier solve and an interval-by-token CDF tensor, chunked over feature coordinates); with a few hundred to about two thousand visual tokens per image this cost is negligible next to the 7B language model, and the scan path is a drop-in replacement. The runtime study of Section 4.1 concerns the standard operator.

D.3 Sensitivity to the routing and read-out widths

Table 20 sweeps both widths of score-routed LapMerge in the same shared harness and the same F1 metric as Table 2: the read-out width α_a at fixed α_r=1, and the routing width α_r at fixed α_a=1, together with the released selection pipeline rerun under identical conditions (its published F1 is reproduced to within 0.06 at both budgets). The POPE-tuned row of Table 2 is the per-budget maximum of this grid and should be read as a reference ceiling rather than a fair comparison.

Three observations follow. *(i)* Merging beats hard selection across the moderate-width range: at the 10% budget every $\alpha_a \in \{0.3, 0.7, 1.0, 1.5\}$ and both routing widths improve over selection, and only the over-wide α_a=3.0 falls marginally below it (79.56 vs 79.67); at 30% the improvement persists for $\alpha_a \leq 1.0$ and disappears beyond. The gain therefore does not hinge on a finely tuned value, but the kernel must not be made arbitrarily wide. *(ii)* The optimal width shrinks as the budget grows: at 10% each output stands in for about nine discarded tokens, so a wider read-out recovers more of the lost evidence; at 30% the kept positions are dense and a narrow window suffices. *(iii)* The routing width is a flat knob: halving or doubling it changes F1 by at most 0.6.

Mechanism control: per-anchor smoothing (Recipe-B). As a control that separates equal-mass routing from mere local smoothing, we also rerun a per-anchor alternative in the same harness and metric: each kept position aggregates its neighbourhood under a softmax of width τ over tokens (Table 19). At τ=0.05 it reproduces the selection pipeline exactly (79.67/85.11, the designed $\tau \to 0$ limit), and its best setting reaches 80.70/85.32 – below score-routed LapMerge at the *fixed* default (81.31/85.50), let alone its ceiling. Equal-mass barrier routing therefore contributes beyond what local smoothing around the kept positions can provide.

	per-anchor softmax width τ				
budget	0.05	0.3	0.7	1.5	3.0
10%	79.67	79.75	80.70	80.58	78.61
30%	85.11	85.32	85.26	84.41	80.92

Table 19: **Per-anchor smoothing control (Recipe-B)**: POPE F1 in the shared harness. τ=0.05 recovers hard selection exactly; the best column trails the default score-routed LapMerge of Table 20 at both budgets.

Uncertainty of the headline differences. Decoding is greedy and both arms answer the same 9,000 questions, so the difference to selection is paired. A nonparametric bootstrap over questions (10,000 resamples, percentile intervals) against our rerun of the released selection gives $\Delta\text{F1} = +1.64$ (95% CI $[+0.88, +2.40]$, $P(\Delta > 0) > 0.999$) at the 10% budget and $\Delta\text{F1} = +0.39$ ($[-0.08, +0.87]$, $P(\Delta > 0) = 0.94$) at 30%: decisive at the aggressive budget, positive but not separated from zero at the mild one.

E Datasets

We evaluate LapMerge on five groups of benchmarks that stress different properties of token merging: (i) gigapixel whole-slide images, where the input is a bag of thousands of tiles and most of the tissue is uninformative for the label; (ii) the Long Range Arena, which probes long-context reasoning on synthetic, textual and image-as-sequence inputs; (iii) POPE, which measures whether a vision-language model still "sees" the objects in an image after its visual tokens are compressed;

Table 20: **Width sweep of score-routed LapMerge** (POPE F1, frozen Qwen2.5-VL-7B; the same shared harness and metric as Table 2). "select" reruns the released selection pipeline under identical conditions and reproduces its published F1 (79.73/85.05) to within 0.06. The starred column is the dataset-agnostic default used in Table 2. Decoding is greedy, so each cell is deterministic.

		read-out width α_a (α_r=1)					routing width α_r (α_a=1)	
budget	select	0.3	0.7	1.0*	1.5	3.0	0.5	2.0
10%	79.67	**82.01**	81.67	81.31	80.69	79.56	80.97	80.70
30%	85.11	**85.87**	**85.87**	85.50	84.73	82.92	85.19	84.97

(iv) fine-grained image classification, where the discriminative evidence is concentrated in small object parts; and (v) long-document topic classification on arXiv, where relevant evidence is spread through the whole text. Table 21 summarises the statistics. Each dataset is described in detail below.

Table 21: Summary of the datasets used in our experiments.

Domain	Dataset	#Classes	Train	Test
Histopathology	TCGA-CRC (COAD+READ)	2	365 slides†	
	TCGA-STAD	2	303 slides†	
LRA	ListOps	10	96,000	2,000
	Text	2	25,000	25,000
	Retrieval	2	147,086	17,437
	Image	10	45,000	10,000
	Pathfinder	2	160,000	20,000
Long documents	arXiv topic classification	11	28,388	2,500
VLM hallucination	POPE	2	–	9,000 Q / 500 img
Fine-grained	Stanford Cars	196	8,144	8,041

†Patient-level splits; see Appendix E.1.

E.1 HISTOPATHOLOGY

Task. We consider slide-level prediction of microsatellite instability (MSI-high, MSI-H, vs. microsatellite stable, MSS) from routine haematoxylin-and-eosin (H&E) stained tissue. MSI-H is caused by a deficient DNA mismatch-repair system. It is a clinically actionable biomarker, because MSI-H tumours respond markedly better to immune checkpoint inhibitors. It is currently determined with dedicated immunohistochemistry or PCR-based genetic assays. Predicting it directly from H&E morphology has become a standard benchmark for computational pathology (Kather et al., 2019; Wagner et al., 2023). A single binary label is available per slide (inherited from the patient), while the morphological evidence, e.g. tumour-infiltrating lymphocytes, mucinous or medullary growth patterns, poor differentiation occupies only a fraction of the tissue.

Cohorts. We use two cohorts from The Cancer Genome Atlas (TCGA):

- **Colorectal cancer (TCGA-CRC):** colon adenocarcinoma (TCGA-COAD) and rectum adenocarcinoma (TCGA-READ), pooled into a single cohort of 365 diagnostic slides.
- **Stomach adenocarcinoma (TCGA-STAD):** 303 diagnostic slides.

All slides are formalin-fixed, paraffin-embedded (FFPE) diagnostic sections, as opposed to the frozen sections that TCGA also releases. FFPE slides preserve morphology substantially better and are the ones used in clinical practice. The slides were collected at many different contributing institutions and therefore exhibit strong variation in staining, section thickness and scanner, which makes the benchmark a realistic test of robustness.

Labels and splits. MSI status labels are taken from the TCGA pan-gastrointestinal molecular analysis of Liu et al. (2018), which assigns MSI status from sequencing-based assays. Tumours

classified as MSI-low are grouped with MSS, following common practice. We follow the published patient-level splits of Kather et al. (2019) and Wagner et al. (2023), so that all slides of one patient fall into the same partition and no patient information leaks between training and evaluation.

Features and protocol. Patch features are the released, gated `MahmoodLab/UNI2-h-features` (HuggingFace): 1536-dimensional UNI2-h embeddings in CLAM-style HDF5 files that also carry the patch-center coordinates; we download these precomputed features and do not re-extract them, so patching geometry and background filtering are inherited from that release. The patient-level splits contain $259/52/54$ slides for CRC (MSI-H fraction $17.4/15.4/16.7\%$) and $205/47/51$ for STAD ($18.5/23.4/15.7\%$); no patient spans two partitions (verified on both manifests; a few patients contribute more than one slide: 365 slides from 359 CRC patients, 303 from 285 STAD patients).

Slide lengths vary widely: CRC has median M=5,922 patches (range 328–31,521), STAD median M=7,329 (range 180–31,458); only 3.6% of CRC and 8.3% of STAD slides have $M \leq 1024$, for which the fixed budget N=1024 is not a compression; such slides go through the same code path, with the operator emitting N convex combinations of the $M \leq N$ inputs (the $N \geq M$ case is exercised in Table 6). Model selection takes the checkpoint with the best validation AUROC; the test AUROC is reported from that checkpoint. The random seed controls only the initialization and batch order – the splits are fixed files. The MIL baselines process the full bag of M patches, whereas LapMerge first reduces it to N=1024 tokens: the comparison matches inputs and splits, not downstream sequence length.

Perturbation test of the routing-mass map. To check that the learned mass is an attribution and not only a budget map, we delete $q \in \{5, 10, 20, 50\}\%$ of each test slide's patches – chosen by highest mass, lowest mass, or uniformly at random (masses always computed on the unperturbed slide) – and re-run the full pipeline on the survivors, for five of the six seed checkpoints per cohort (those with saved weights). Deleting the highest-mass patches degrades AUROC with growing q (CRC: $0.912 \rightarrow 0.894 \rightarrow 0.895 \rightarrow 0.868 \rightarrow 0.800$ at q=$0/5/10/20/50\%$; STAD: $0.959 \rightarrow 0.956 \rightarrow 0.941 \rightarrow 0.920 \rightarrow 0.849$), while deleting the lowest-mass or random patches leaves it essentially unchanged even at q=50% (CRC 0.902/0.895, STAD 0.962/0.962). The mass map therefore passes a deletion-style attribution test: the prediction rests on the patches the map points to.

What the scorer attends to. Figure 8 complements the quantitative test above with a qualitative view: for six CRC test slides (three MSI-H, three MSS, all correctly and confidently classified) it pairs the true-color thumbnail with the same thumbnail desaturated under the scorer-weight overlay, where opacity is tied to the weight so that low-weight patches fade into the background. The weight is visibly non-uniform and slide-specific, concentrated on compact sub-regions that differ from slide to slide – consistent with the scorer doing real, content-dependent work rather than acting as a near-uniform pass-through.

E.2 LRA

The Long Range Arena (LRA) (Tay et al., 2021) is a suite of sequence-classification tasks designed to evaluate efficient Transformers under long contexts (1K–16K tokens). It covers hierarchical/structured reasoning, byte-level text, document matching, and image-as-sequence inputs. In all tasks the model sees a flat sequence of discrete tokens with no built-in inductive bias for the underlying structure. We use the standard data, pre-processing and splits released with the benchmark. In our experiments the reducer and a four-layer Transformer backbone are trained jointly from scratch on every task with Adam (initial learning rate 10^{-3}, cosine decay, 80,000 steps, batch size 32); all budget-matched baselines share the backbone, optimizer, and training budget. The shared 10% budget sets N per task from its input length M: ListOps $2{,}048 \rightarrow 204$, Text $4{,}096 \rightarrow 410$, and Image/Pathfinder $1{,}024 \rightarrow 102$ tokens, i.e. roughly $100\times$ fewer attention pairs in the reduced backbone.

ListOps. Each input is a nested prefix-notation expression (Nangia & Bowman, 2018) over single digits (0–9) with the operators `MAX`, `MIN`, `MED` (median) and `SM` (sum modulo 10), e.g. `[MAX 4 3 [MIN 2 3 ] 1 0 [MED 1 5 8 9 2 ]]`. The target is the value of the expression, i.e.

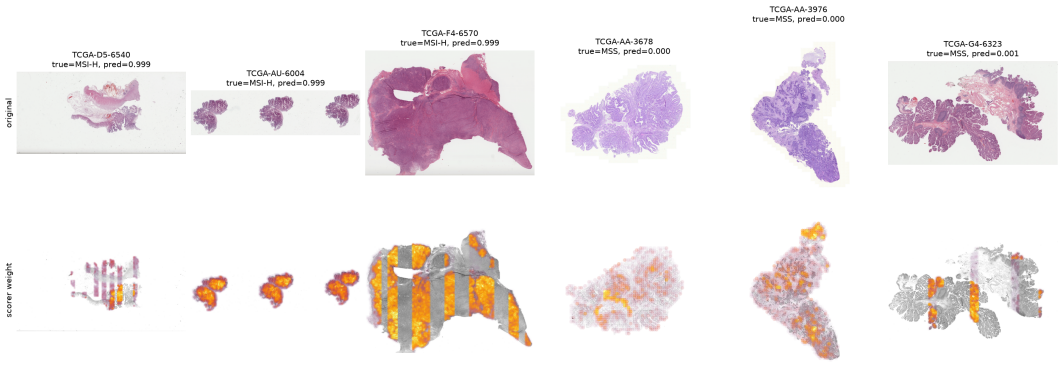

Figure 8: Per-patch scorer weight of LapMerge for six TCGA-CRC test slides (three MSI-H, three MSS), one column per slide: true-color thumbnail with true label and predicted $p(\text{MSI-H})$ (top) vs. the desaturated scorer-weight overlay (bottom; opacity tied to weight, so only the regions the scorer relies on stand out). *The banded columns (TCGA-D5-6540, TCGA-F4-6570) are gaps in patch coverage between several narrow tissue fragments on those slides – confirmed from the raw patch coordinates – not a scorer artefact.*

a 10-way classification. Sequences are up to 2,048 tokens long and the vocabulary consists of the digits, operators and brackets. Solving the task requires modelling the hierarchical structure over the whole sequence. A single token at the very beginning can change the result. Split: 96,000 / 2,000 / 2,000 (train / val / test).

Text. Binary sentiment classification on the IMDb movie-review dataset (Maas et al., 2011), operating at the *byte* (character) level rather than on word tokens, which turns a moderate-length document task into a long-sequence one and forces the model to compose characters into words and phrases. Reviews are truncated or padded to 4,096 bytes. Split: 25,000 training and 25,000 test reviews, balanced between positive and negative.

Retrieval. Byte-level document matching on the ACL Anthology Network (AAN) corpus (Radev et al., 2013). Given two scientific papers, the model predicts whether one cites the other (binary). Each document is encoded separately at byte level with up to 4,000 tokens (8,000 tokens per pair); following the LRA protocol the two document representations are combined only at the classifier (a two-tower setting), so the task tests how well a fixed-size representation compresses a long document. Split: 147,086 / 18,090 / 17,437.

Image. CIFAR-10 (Krizhevsky, 2009) images converted to 8-bit grayscale and flattened in raster-scan order into sequences of $32 \times 32 = 1{,}024$ pixel intensities (vocabulary of 256), with 10 object classes. Because no 2D structure is given explicitly, the model must recover spatial relations between pixels that are 32 positions apart in the sequence. Split: 45,000 / 5,000 / 10,000.

Pathfinder. A synthetic visual reasoning task (Linsley et al., 2018). Each 32×32 grayscale image contains two highlighted endpoints and a number of dashed curves, and the model must decide whether the two endpoints are connected by a single path (binary). Distractor paths make local cues uninformative, so the task requires integrating evidence along long, curved paths. Images are flattened to 1,024 tokens. Split: 160,000 / 20,000 / 20,000. Path-X, the 128×128 (16,384-token) version, is not included in our experiments.

E.3 arXiv Topic Classification

The `ccdv/arxiv-classification` dataset assigns each of 33,388 arXiv papers one of 11 subject classes. We use the released public split (28,388 / 2,500 / 2,500 train/validation/test) and truncate each document to $M{=}1024$ tokens of the frozen `allenai/longformer-base-4096`

tokenizer. Relevant evidence is spread through the whole text, which makes the task a deliberate stress test for order-based merging; the full training protocol and results are in Appendix C.3.

E.4 POPE

Task. The Polling-based Object Probing Evaluation (POPE) (Li et al., 2023) measures object hallucination in large vision-language models (LVLMs), i.e. the tendency to claim that objects are present in an image when they are not. Instead of parsing free-form captions, POPE casts evaluation as a set of binary questions of the form *"Is there a <object> in the image?"*, which the model must answer with "yes" or "no". This makes it a direct probe of whether fine-grained visual evidence survives token compression. An aggressive merging scheme that discards small or peripheral objects will increase false negatives, whereas one that blends object and background tokens can increase false positives.

Data. We use the standard MSCOCO version of POPE, built on 500 images, each containing at least three annotated object categories. For every image, three positive questions are formed from ground-truth objects and three negative questions from objects absent from the image, giving a balanced set of 3,000 questions (50% "yes") per setting. Three settings differ only in how the negative objects are sampled:

- **Random:** negatives are drawn uniformly from the COCO object vocabulary among categories not present in the image.
- **Popular:** negatives are the most frequent COCO categories not present in the image, testing whether the model is biased towards generally common objects.
- **Adversarial:** negatives are the categories that most often *co-occur* with the image's ground-truth objects (e.g. "fork" for an image with a "dining table"), the hardest setting, since language priors strongly suggest their presence.

In total, the benchmark comprises 9,000 image–question pairs.

E.5 Fine-grained Vision: Stanford Cars

Stanford Cars. Stanford Cars (Krause et al., 2013) contains 16,185 images of 196 car classes, where a class is defined at the level of *make, model and year* (e.g. *2012 Tesla Model S* vs. *2012 BMW M3 coupe*). The official split has 8,144 training and 8,041 test images, i.e. roughly a 50–50 split with about 41 training images per class. Images come from the web and vary strongly in resolution, viewpoint (front, rear, side, three-quarter views) and background. Several classes differ only by model year and are distinguishable only through subtle details, making the dataset particularly demanding for methods that discard high-frequency information.