FAST AND SIMPLEX: 2-SIMPLICIAL ATTENTION IN TRITON

Anonymous authors
Paper under double-blind review

ABSTRACT

Recent work has shown that training loss scales as a power law with both model size and the number of tokens, and that achieving compute-optimal models requires scaling model size and token count together. However, these scaling laws assume an infinite supply of data and apply primarily in compute-bound settings. As modern large language models increasingly rely on massive internet-scale datasets, the assumption that they are compute-bound is becoming less valid. This shift highlights the need for architectures that prioritize token efficiency.

In this work, we investigate the use of the 2-simplicial Transformer, an architecture that generalizes standard dot-product attention to trilinear functions through an efficient Triton kernel implementation. We demonstrate that the 2-simplicial Transformer achieves better token efficiency than standard Transformers: for a fixed token budget, similarly sized models outperform their dot-product counterparts on tasks involving mathematics, coding, reasoning, and logic. We quantify these gains by demonstrating that 2-simplicial attention changes the exponent in the scaling laws for knowledge and reasoning tasks compared to dot product attention.

1 INTRODUCTION

Large language models (LLMs) based on the Transformer architecture (Vaswani et al., 2017) have become foundational to many state-of-the-art artificial intelligence systems, including GPT-3 (Brown et al., 2020), GPT-4 (Achiam et al., 2023), Gemini (Team et al., 2023), and Llama (Touvron et al., 2023). The remarkable progress in scaling these models has been guided by neural scaling laws (Hestness et al., 2017; Kaplan et al., 2020; Hoffmann et al., 2022), which empirically establish a power-law relationship between training loss, number of model parameters, and size of training data.

A key insight from this body of work is that optimal model performance is achieved not simply by increasing model size, but by scaling both the number of parameters and the amount of training data in tandem. Notably, Hoffmann et al. (2022) demonstrate that compute-optimal models require a balanced scaling approach. Their findings show that the Chinchilla model, with 70 billion parameters, outperforms the much larger Gopher model (280 billion parameters) by being trained on four times as much data. This result underscores the importance of data scaling alongside model scaling for achieving superior performance in large language models.

As artificial intelligence (AI) continues to advance, a significant emerging challenge is the availability of sufficiently high-quality tokens. As we approach this critical juncture, it becomes imperative to explore novel methods and architectures that can scale more efficiently than traditional Transformers under a limited token budget. However, most architectural and optimizer improvements merely shift the error but do not meaningfully change the exponent of the power law (Everett, 2025). The work of Kaplan et al. (2020); Shen et al. (2024) showed that most architectural modifications do not change the exponent, while Hestness et al. (2017) show a similar result for optimizers. The only positive result has been on data due to the works of Sorscher et al. (2022); Bahri et al. (2024); Brandfonbrener et al. (2024) who show that changing the data distribution can affect the exponent in the scaling laws.

In this context we revisit an old work Clift et al. (2020) which generalizes the dot product attention of Transformers to trilinear forms as the 2-simplicial Transformer. We explore generalizations of RoPE (Su et al., 2024) to trilinear functions and present a rotation invariant trilinear form that we prove is as expressive as 2-simplicial attention. We further show that the 2-simplicial Transformer scales better than the Transformer under a limited token budget: for a fixed number of tokens, a similar sized 2-simplicial Transformer out-performs the Transformer on math, coding and reasoning tasks. Furthermore, our experiments also reveal that the 2-simplicial Transformer has a more favorable scaling exponent corresponding to the number of parameters than the Transformer (Vaswani et al., 2017). This suggests that, unlike Chinchilla scaling (Hoffmann et al., 2022), it is possible to increase tokens at a slower rate than the parameters for the 2-simplicial Transformer. Our findings imply that, when operating under token constraints, the 2-simplicial Transformer can more effectively approach the irreducible entropy of natural language compared to dot product attention Transformers.

2 Related work

Several generalizations of attention have been proposed since the seminal work of Vaswani et al. (2017). A line of work that started immediately after was to reduce the quadratic complexity of attention with sequence length. In particular, the work of Parmar et al. (2018) proposed local attention in the context of image generation and several other works subsequently used it in conjunction with other methods for language modeling (Zaheer et al., 2020; Roy et al., 2021). Other work has proposed doing away with softmax attention altogether - e.g., Katharopoulos et al. (2020) show that replacing the softmax with an exponential without normalization leads to linear time Transformers using the associativity of matrix multiplication. Other linear time attention work are state space models such as Mamba (Gu & Dao, 2023); however these linear time attention methods have received less widespread adoption due to their worse quality compared to Transformers in practice. According to Allen (2025), the key factor contributing to Mamba's success in practical applications is the utilization of the $\mathrm{conv1d}$ operator; see also So et al. (2021) and Roy et al. (2022) for similar proposals to the Transformer architecture.

The other end of the spectrum is going from quadratic to higher order attention. The first work in this direction to the best of our knowledge was 2-simplicial attention proposed by Clift et al. (2020) which showed that it is a good proxy for logical problems in the context of deep reinforcement learning. A similar generalization of Transformers was proposed in Bergen et al. (2021) which proposed the *Edge Transformer* where the authors proposed *triangular attention*. The AlphaFold (Jumper et al., 2021) paper also used an attention mechanism similar to the *Edge Transformer* which the authors called *triangle self-attention* induced by the $2D$ geometry of proteins. Higher order interactions were also explored in Wang et al. (2021) in the context of recommender systems. Simplical attention was also explored in the context of Hopfield networks in Burns & Fukai (2023). Recent work by Sanford et al. (2023) shows that the class of problems solved by an n-layer 2-simplicial Transformer is strictly larger than the class of problems solved by dot product attention Transformers. In particular, the authors define a class of problems referred to as `Match3` and show that dot product attention requires exponentially many layers in the sequence length to solve this task. Follow up work by Kozachinskiy et al. (2025) propose a scalable approximation to 2-simplicial attention and prove lowerbounds between Strassen attention and dot product attention on tasks that require more complex reasoning using VC dimension (Vapnik, 1968) arguments.

Also related is work on looping Transformer layers (Dehghani et al., 2018) as in Universal Transformers; see also Yang et al. (2023); Saunshi et al. (2025) for a more recent treatment of the same idea. Both higher order attention and looping serve a similar purpose: compute a more expressive function per parameter. It has been established in these works that looped Transformers are better at logical reasoning tasks. A key challenge in scaling looped Transformers to larger models is their trainability. Specifically, looping k times increases the model depth by a factor of k, which can significantly exacerbate the difficulties associated with training deeper models. As a result, it remains unclear how well large looped Transformers can be trained, and further research is needed to address this concern.

Notation. We use small and bold letters to denote vectors, capital letters to denote matrices and tensors and small letters to denote scalars. We denote $\langle \mathbf{a}, \mathbf{b} \rangle$ to denote dot product between two vectors $\mathbf{a}$ and $\mathbf{b}$. Similarly, the trilinear dot product is denoted as follows:$\langle \mathbf{a}, \mathbf{b}, \mathbf{c} \rangle = \sum_{i=1}^{d} \langle \mathbf{a}_i, \mathbf{b}_i, \mathbf{c}_i \rangle$. We use @ to highlight a matrix multiplication, for e.g., $(AB)@C$, for matrices A, B, C. To denote array slicing, we use $\mathbf{a}[l : l + m] = (a_l, \dots, a_{l+m-1})$ with zero-based indexing. Some tensor operations are described using Einstein summation notation as used in the Numpy library (Harris et al., 2020). We use $FLOPs$ to denote floating point operations. Column stacking of arrays are denoted by $[\mathbf{a}, \mathbf{b}, \mathbf{c}]$. We use $\det$ to denote determinant of a square matrix.

3 Overview of neural scaling laws

In this section we provide a brief overview of neural scaling laws as introduced in Kaplan et al. (2020). We will adopt the approach outlined by Hoffmann et al. (2022), which proposes that the loss $L(N, D)$ decays as a power law in the total number of model parameters N and the number of tokens D:

$$L(N, D) = E + \frac{A}{N^{\alpha}} + \frac{B}{D^{\beta}}. \tag{1}$$

The first term E is often described as the *irreducible loss* which corresponds to the entropy of natural text. The second term captures the fact that a model with N parameters underperforms this ideal generative process. The third term corresponds to the fact that we train on only a finite sample of the data and do not train the model to convergence. Theoretically, as $N \to \infty$ and $D \to \infty$ a large language model should approach the irreducible loss E of the underlying text distribution.

For a given compute budget C where $FLOPs(N, D) = C$, one can express the optimal number of parameters as $N_{opt} \propto C^{a}$ and the optimal dataset size as $D_{opt} \propto C^{b}$. The authors of Hoffmann et al. (2022) perform several experiments and fit parametric functions to the loss to estimate the exponents a and b: multiple different approaches confirm that roughly $a \sim 0.49$ while $b \sim 0.5$. This leads to the central thesis of Hoffmann et al. (2022): one must scale the number of tokens proportionally to the model size.

However, as discussed in Section 1, the quantity of sufficiently high-quality tokens is an emerging bottleneck in pre-training scaling, necessitating an exploration of alternative training algorithms and architectures. On the other hand recent studies have shown that most modeling and optimization techniques proposed in the literature merely shift the error (offset E) and do not fundamentally change exponent in the power law. We refer the readers to the excellent discussion in Everett (2025).

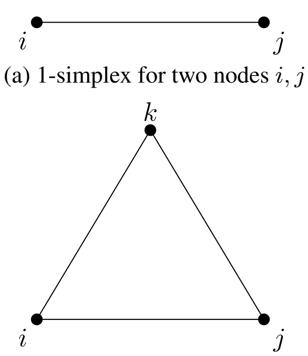
4 The 2-simplicial Transformer

The 2-simplicial Transformer was introduced in Clift et al. (2020) where the authors extended the dot product attention from bilinear to trilinear forms, or equivalently from the 1-simplex to the 2-simplex. Let us recall the attention mechanism in a standard Transformer (Vaswani et al., 2017). Given a sequence $X \in \mathbb{R}^{n \times d}$ we have three projection matrices $W_Q, W_K, W_V \in \mathbb{R}^{d \times d}$ which we refer to as the query, key and value projections respectively. These projection matrices are used to infer the query $Q = XW_Q$, key $K = XW_K$ and value $V = XW_V$ respectively. This is then used to construct the *attention logits*:

$$A = QK^{\top}/\sqrt{d} \in \mathbb{R}^{n \times n}, \tag{2}$$

where each entry is a dot product $A_{ij} = \langle \boldsymbol{q}_i, \mathbf{k}_j \rangle / \sqrt{d}$ which are both entries in $\mathbb{R}^d$. The attention scores (logits) are then transformed into probability weights by using a row-wise softmax operation:

$$S_{ij} = \exp\left(A_{ij}\right) / \sum_{j=1}^{n} \exp\left(A_{ij}\right). \tag{3}$$



(a) 1-simplex for two nodes i, j

(b) 2-simplex between three nodes i, j, k

Figure 1: Geometry of dot product attention and 2-simplical attention.

The final output of the attention layer is then a linear combination of the values according to these attention scores:

$$\tilde{\boldsymbol{v}}_i = \sum_{j=1}^{n} A_{ij} \boldsymbol{v}_j \tag{4}$$

The 2-simplicial Transformer paper Clift et al. (2020) generalizes this to trilinear products where we have two additional key and value projection matrices $W_{K'}$ and $W_{V'}$, which give us $K' = XW_{K'}$ and $V' = XW_{V'}$. The attention logits for 2-simplicial Transformer are then given by the trilinear product between Q, K and K', resulting in the following third-order tensor:

$$A_{ijk}^{(2s)} = \frac{\langle \boldsymbol{q}_i, \mathbf{k}_j, \mathbf{k}'_k \rangle}{\sqrt{d}} = \frac{1}{\sqrt{d}} \sum_{l=1}^{d} Q_{il} K_{jl} K'_{kl}, \tag{5}$$

Algorithm 1 Pseudocode for the forward pass of 2-simplicial attention

```
1: procedure 2-SIMPLICIAL ATTENTION(Q, K, V, K', V')
2:     logits ← einsum("btnh, bsnh, brnh → bntsr", Q, K, K')
3:     attention ← softmax(logits + causal-mask, axis = [−1, −2])
4:     output ← einsum("bntsr, bsnh, brnh → btnh", attention, V, V')
5:     return output
6: end procedure
```

so that the attention tensor becomes:

$$S_{ijk}^{(2s)} = \exp(A_{ijk}^{(2s)}) / \sum_{j,k} \exp(A_{ijk}^{(2s)}), \tag{6}$$

with the final output of the attention operation being defined as

$$\tilde{\boldsymbol{v}}^{(2s)}(i) = \sum_{j,k=1}^{n} S_{ijk}^{(2s)} \left(\boldsymbol{v}_j \circ \boldsymbol{v}'_k \right), \tag{7}$$

where $\boldsymbol{v}_j \circ \boldsymbol{v}'_k$ represents the element wise Hadamard product between two vectors in $\mathbb{R}^d$. The pseudo-code for 2-simplicial attention is depicted in Algorithm 1. Note that Equation 5 does not incorporate any position encoding such as RoPE (Su et al., 2024); we discuss this in the next section.

5 Determinant based Trilinear Forms

RoPE (Su et al., 2024) was proposed as a way to capture the positional information in a sequence for Transformer language models. RoPE applies a position dependent rotation to the queries $\boldsymbol{q}_i$ and the key $\mathbf{k}_j$ so that the dot product $\langle \boldsymbol{q}_i, \mathbf{k}_j \rangle$ is a function of the relative distance $i - j$. In particular, note that the dot product is invariant to orthogonal transformations $R \in \mathbb{R}^{d \times d}$:

$$\langle \boldsymbol{q}_i, \mathbf{k}_j \rangle = \langle R\boldsymbol{q}_i, R\mathbf{k}_j \rangle.$$

This is important for RoPE to work as for a query $\boldsymbol{q}_i$ and key $\mathbf{k}_i$ at the same position i, we expect its dot product to be unchanged by the application of position based rotations: $\langle \boldsymbol{q}_i, \mathbf{k}_i \rangle = \langle R\boldsymbol{q}_i, R\mathbf{k}_i \rangle$.

Note that the trilinear form defined in Equation 5 is not invariant to rotation and the application of the same rotation to $\boldsymbol{q}_i, \mathbf{k}_i$ and $\mathbf{k}'_i$ no longer preserves the inner product: $\langle \boldsymbol{q}_i, \mathbf{k}_i, \mathbf{k}'_i \rangle = \sum_{l=1}^{d} \boldsymbol{q}_{il} \mathbf{k}_{il} \mathbf{k}'_{il} \neq \langle R\boldsymbol{q}_i, R\mathbf{k}_i, R\mathbf{k}'_i \rangle$. Therefore, to generalize RoPE to 2-simplicial attention, it is important to explore alternative bilinear and trilinear forms that are rotation invariant.

We note that the following functions are also invariant to rotations:

$$\begin{aligned}
\hat{f}_2(\mathbf{a}, \mathbf{b}) &= \det \begin{pmatrix} a_1 & a_2 \\ b_1 & b_2 \end{pmatrix} = a_1 b_2 - a_2 b_1, \\
\hat{f}_3(\mathbf{a}, \mathbf{b}, \mathbf{c}) &= \det \begin{pmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ c_1 & c_2 & c_3 \end{pmatrix}, \\
&= a_1 b_2 c_3 + a_2 b_3 c_1 + a_3 b_1 c_2 - a_1 b_3 c_2 - a_2 b_1 c_3 - a_3 b_2 c_1 \\
&= \langle (a_1, a_2, a_3), (b_2, b_3, b_1), (c_3, c_1, c_2) \rangle - \langle (a_1, a_2, a_3), (b_3, b_1, b_2), (c_2, c_3, c_1) \rangle,
\end{aligned} \tag{8}$$

the rearrangement in the last equality is popularly called Sarrus rule (Strang, 2022). Here, $\hat{f}_2$ is a bilinear form in $\mathbf{a} = (a_1, a_2)$ and $\mathbf{b} = (b_1, b_2)$ and $\hat{f}_3$ is a trilinear form in $\mathbf{a} = (a_1, a_2, a_3)$, $\mathbf{b} = (b_1, b_2, b_3)$, $\mathbf{c} = (c_1, c_2, c_3)$. Geometrically, $|\hat{f}_2(\mathbf{a}, \mathbf{b})|$ measures the area of the parallelogram spanned by $\mathbf{a}$ and $\mathbf{b}$, and similarly, $|\hat{f}_2(\mathbf{a}, \mathbf{b}, \mathbf{c})|$ measures the volume of the parallelotope spanned by $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$. We use the signed determinant operation $\hat{f}_3$ to compute $A^{(\det)} \in \mathbb{R}^{n \times n \times n}$. For any vector $\boldsymbol{q}$, let $\boldsymbol{q}^{(l)} = \boldsymbol{q} = \boldsymbol{q}[3(l-1) : 3l]$ be its lth chunk of size 3. The logits are defined as:

$$A_{ij_1j_2}^{(\det)} = \sum_{l=1}^{p} \det([\boldsymbol{q}_i^{(l)}, \mathbf{k}_{j_1}^{(l)}, {\mathbf{k}'_{j_2}}^{(l)}]). \tag{9}$$

Since Equation 8 has 2 dot product terms due to Sarrus rule, it would modify Algorithm 1 to use 2 einsums instead of 1 in line 2. The final attention weights S are computed by applying a softmax function on the logits above, similar to Equation 6. The output for token i is then the weighted sum of value vectors as in Equation 7.

Theorem 5.1. *For any input size n and input range $m = n^{O(1)}$, there exists a transformer architecture with a single head of attention with logits computed as in (9), with attention head dimension $d = 7$, such that for all $X \in [M]^N$, the transformer's output for element x_i is 1 if $\exists j_1, j_2$ s.t. $x_i + x_{j_1} + x_{j_2} = 0 \pmod M$, and 0 otherwise.*

We provide the proof in Appendix A. Since the sum-of-determinants trilinear function of Equation 9 involves 6 terms compared to the simpler trilinear form of Equation 5, in the following sections where we compute the backwards function for 2-simplicial attention, we will use the simpler trilinear form of Equation 5 without loss of generality.

6 Trace based Trilinear forms and 2-D RoPE

In this section we present an additional rotation invariant trilinear form using the trace operator on matrices. Given a vector $\boldsymbol{x} \in \mathbb{R}^d$, we denote by $\mathrm{mat}\,(\boldsymbol{x})$ the $\sqrt{d} \times \sqrt{d}$ matrix obtained by reshaping the vector $\boldsymbol{x}$. Therefore, equivalently we have the following identity $\mathrm{mat}(\boldsymbol{x}) = \boldsymbol{x}$. Then we note that the following function is also invariant to rotation and is equivalent to dot product attention in the 2-D case:

$$\langle \boldsymbol{q}_i, \mathbf{k}_j \rangle = \mathrm{tr}\left(\mathrm{mat}(\boldsymbol{q}_i)^\top \mathrm{mat}(\mathbf{k}_j)\right), \tag{10}$$

where $\mathrm{mat}(\boldsymbol{q}_i)\,\mathrm{mat}(\mathbf{k}_j)$ corresponds to matrix multiplication of the $\sqrt{d} \times \sqrt{d}$ matrices and tr is the trace operator $\mathrm{tr}(A) = \sum_i A_{ii}$ which sums over the diagonal elements of a matrix A. An alternate (and elegant) formulation of the trace of a matrix is that it is the sum of its eigenvalues. Note that equation 10 follows from the identity that for any two square matrices A, B we have $\mathrm{tr}(A^\top B) = \langle \mathrm{vec}(A), \mathrm{vec}(B) \rangle$. We note that this bilinear form is also invariant to orthogonal transformations (rotations) R as

$$\mathrm{tr}((RAR^\top)^\top RBR^\top) = \mathrm{tr}(RA^\top BR^\top) = \mathrm{tr}(A^\top B). \tag{11}$$

This follows since the eigenvalues of a matrix are invariant to orthogonal transformations. This leads us to the trilinear generalization $\langle \boldsymbol{q}_i, \mathbf{k}_{j1}, \mathbf{k}'_{j2} \rangle_{\mathrm{tr}}$:

$$\langle \boldsymbol{q}_i, \mathbf{k}_{j1}, \mathbf{k}_{j2} \rangle_{\mathrm{tr}} = \mathrm{tr}(\mathrm{mat}(\boldsymbol{q}_i)\,\mathrm{mat}(\mathbf{k}_{j1})\,\mathrm{mat}(\mathbf{k}'_{j2})), \tag{12}$$

which is no longer equivalent to the standard trilinear product, $\langle \boldsymbol{q}_i, \mathbf{k}_{j1}, \mathbf{k}_{j2} \rangle_{\mathrm{tr}} \neq \langle \boldsymbol{q}_i, \mathbf{k}_{j1}, \mathbf{k}_{j2} \rangle$. The logits of the trace based trilinear attention is then defined as:

$$A^{(\mathrm{tr})}_{ij_1j_2} = \mathrm{tr}(\mathrm{mat}(\boldsymbol{q}_i)\,\mathrm{mat}(\mathbf{k}_{j_1})\,\mathrm{mat}(\mathbf{k}'_{j_2})). \tag{13}$$

While the trilinear form of Equation 13 is arguably simpler than the determinant based form and also preserves rotational invariance, the requirement of reshaping the matrix dimensions to $\sqrt{d}$ makes it more challenging to integrate with Flash attention (Dao et al., 2022). Hence we do not present any experimental results with this formulation.

7 Model design

Since 2-simplicial attention scales as $\mathcal{O}(n^3)$ in the sequence length n, it is impractical to apply it over the entire sequence. Instead, we parametrize it as $\mathcal{O}(n \times w_1 \times w_2)$, where w_1 and w_2 define the dimensions of a sliding window over the sequence. Each query vector Q_i attends to a localized region of w_1 K keys and w_2 K' keys, thereby reducing the computational burden. We systematically evaluate various configurations of w_1 and w_2 to identify optimal trade-offs between computational efficiency and model performance (see Table 1).

For causal dot product attention, the complexity for a sequence of length n is given by:

$$O(A) = \frac{1}{2} \cdot 2 \cdot 2n^2 = 2n^2,$$

!t

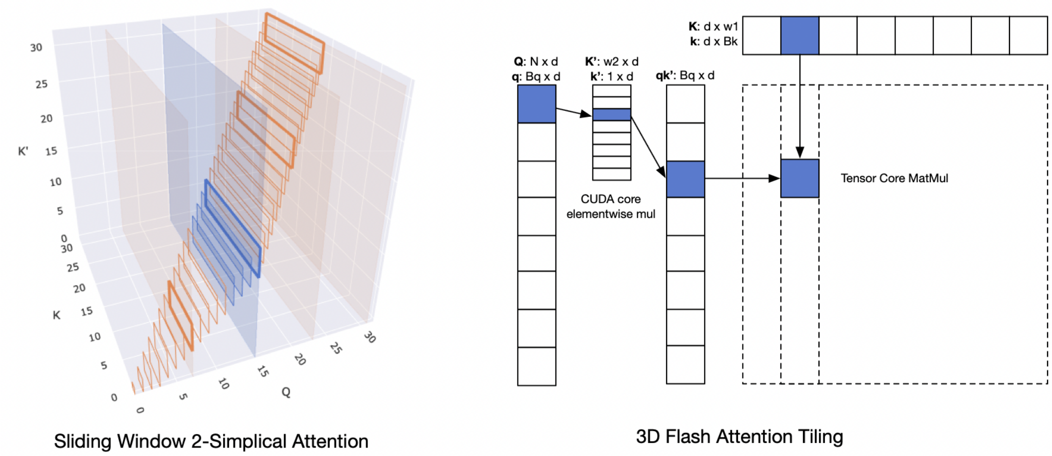


Figure 2: **Left:** Visualization of sliding window 2-simplical attention. Each Q_i attends to a $[w1, w2]$ shaped rectangle of K, K'. **Right:** Tiling to reduce 2-simplicial einsum QKK' to elementwise mul QK' on CUDA core and tiled matmul $(QK')@K$ on tensor core.

where n is the sequence length. This involves two matrix multiplications: one for $Q@K$, one for $P@V$, each requiring two floating-point operations per element. The causal mask allows us to skip $\frac{1}{2}$ of these computations.

In contrast, the complexity of 2-simplical attention, parameterized by w_1 and w_2, is expressed as:

$$O(A^{(2s)}) = 3 \cdot 2nw_1w_2 = 6nw_1w_2$$

This increase in complexity arises from the trilinear einsum operation, which necessitates an additional multiplication compared to standard dot product attention.

We choose a window size of (512, 32), balancing latency and quality. With this configuration, the computational complexity of 2-simplical attention is comparable to dot product attention at $48k$ context length.

A naive sliding window 2-simplicial attention implementation has each Q_i vector attending to $w_1 + w_2 - 1$ different KK' vectors, as illustrated in Figure 2. Thus, tiling queries Q like in flash attention leads to poor compute throughput. Inspired by Native Sparse Attention (Yuan et al., 2025), we adopt a model architecture leveraging a high Grouped Query Attention GQA (Ainslie et al., 2023) ratio of 64 . This approach enabled efficient tiling along query heads, ensuring dense computation and eliminating the need for costly element-wise masking.

$w_1 \times w_2$	w_1	w_2	**Latency (ms)**
$32k$	1024	32	104.1 ms
$32k$	512	64	110.7 ms
$16k$	128	128	59.2 ms
$16k$	256	64	55.8 ms
$16k$	512	32	55.1 ms
$16k$	1024	16	55.1 ms
$8k$	256	32	28.3 ms

Table 1: Latency for different combinations of w_1, w_2

8 KERNEL OPTIMIZATION

We introduce a series of kernel optimizations tailored for 2-simplical attention, building off of Flash Attention (Dao et al., 2022) using online softmax. For the trilinear operations, we perform 2d tiling by merging one of the inputs via elementwise multiplication and executing matmul on the product as illustrated in Figure 2. This allows us to overlap both QK and VV' on CUDA Core with $(QK)@K'$ and $P@(VV')$ on Tensor Core. Implementing this in Triton, we achieve 520 TFLOPS, rivaling the fastest FAv3 Triton implementations. Further optimization could be achieved with a lower-level language like CUTLASS for finer grained tuning and optimizations. Despite this, we achieve competitive performance compared to CUTLASS FAv3 for large sequence lengths, as shown in Figure 3.

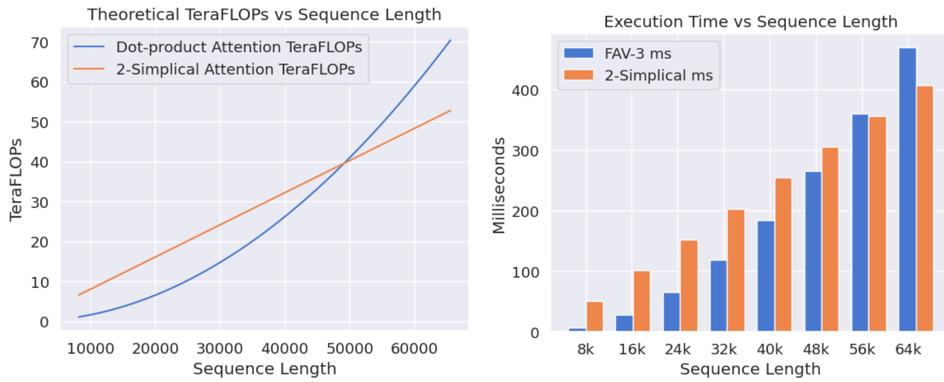


Figure 3: FLOPs and Latencies of FAv3 vs 2-simplical attention

For the backwards pass, aggregations across three different dimension orderings introduces significant overhead from atomic operations. (Exact computation needed provided in Appendix B.) To mitigate this, we decompose the backward pass into two distinct kernels: one for computing dK and dV, and another for dK', dV', and dQ. Although this approach incurs additional overhead from recomputing O and dS, we find it is better than the extra overhead from atomics needed for a single fused kernel. We note this may be a limitation of Triton's coarser grained pipeline control making it difficult to hide the overhead from atomics.

For small w_2, we employ a two-stage approach to compute dQ jointly with dK', dV' without atomics as detailed in Algorithm 2. We divide Q along the sequence dimension into

$$[w_2, dim]$$

sized tiles. First we iterate over even tiles, storing dQ, dK, dK', and dV, dV'. Then we iterate over odd tiles, storing dQ, and adding to dK, dK' and dV, dV'.

Algorithm 2 Backward pass for 2-simplicial attention

procedure 2-SIMPLICIAL FLASH ATTENTION BWD(Q, K, V, K', V', w_1, w_2)
  for stage in [0, 1] **do**
    for q_start in range(stage * w_2, seq_len, w_2 * 2) **do**
      q_end ← q_start + w_2
      for kv1_start in range(q_start - w_1, q_end) **do**
        q_tile ← Q[q_start : q_end]
        ...
        k2_tile ← K′[kv1_start : q_end]
        dQ += dQ(q_tile, k2_tile, ...)
        dV' += dV'(q_tile, k2_tile, ...)
        dK' += dK'(q_tile, k2_tile, ...)
      end for
      if stage == 1 **then**
        dK' += **load** dK'
        dV' += **load** dV'
      end if
      store dQ, ..., dK'
    end for
  end for
end procedure

9 Experiments & Results

We train a series of MoE models (Jordan & Jacobs, 1994; Shazeer et al., 2017) ranging from 1 billion active parameters and 57 billion total parameters to 3.5 billion active parameters and 176 billion total parameters. We use interleaved sliding-window 2-simplicial attention, where every fourth layer is a 2-simplicial attention layer. The choice of this particular ordering is to distribute the load in attention computation when using pipeline parallelism (Huang et al., 2019; Narayanan et al., 2019), since 2-simplicial attention and global attention are the most compute intensive operations in a single pipeline stage and have comparable FLOPs.

We use the AdamW optimizer (Loshchilov et al., 2017) with a peak learning rate of 4×10^{-3} and weight decay of 0.0125. We use a warmup of 4000 steps and use a cosine decay learning schedule decreasing the learning rate to $0.01\times$ of the peak learning rate. We report the negative log-likelihood on GSM8k (Cobbe et al., 2021), MMLU (Hendrycks et al., 2020), MMLU-pro (Wang et al., 2024) and MBPP (Austin et al., 2021), since these benchmarks most strongly test math, reasoning and coding skills in pre-training.

Model	Active Params	Total Params	GSM8k	MMLU	MMLU-pro	MBPP
Transformer	1B	57B	0.3277	0.6411	0.8718	0.2690
2-simplicial	1B	57B	0.3302	0.6423	0.8718	0.2714
$\Delta(\%)$			+0.79%	+0.19%	-0.01%	+0.88%
Transformer	2B	100B	0.2987	0.5932	0.8193	0.2435
2-simplicial	2B	100B	0.2942	0.5862	0.8135	0.2411
$\Delta(\%)$			-1.51%	-1.19%	-0.71%	-1%
Transformer	3.5B	176B	0.2781	0.5543	0.7858	0.2203
2-simplicial	3.5B	176B	0.2718	0.5484	0.7689	0.2193
$\Delta(\%)$			-2.27%	-1.06%	-2.15%	-0.45%

Table 2: Negative log-likelihood of Transformer (Vaswani et al., 2017) versus 2-simplicial attention. For MMLU (Hendrycks et al., 2020) and MMLU-pro (Wang et al., 2024) we measure the negative log-likelihood of the choice together with the entire answer. For GSM8k (Cobbe et al., 2021) we use 5-shots for the results.

We see that the decrease (Δ) in negative log-likelihood scaling from a 1.0 billion (active) parameter model increases going to a 3.5 billion (active) parameter model. Furthermore, on models smaller than 2.0 billion (active) parameters, we see no gains from using 2-simplicial attention. From Table 2 we can estimate how the power law coefficients for the 2-simplicial attention differ from dot product attention. Recall from Section 3 that the loss can be expressed as:

$$L(N, D) = E + \frac{A}{N^{\alpha}} + \frac{B}{D^{\beta}}. \tag{14}$$

Since we train both the models on the same fixed number of tokens, we may ignore the third term and simply write the loss as $L(N) = E' + A/N^{\alpha}$ with $E' = E + B/D^{\beta}$. We can approximate for curve fitting as follows:

$$-\log L(N) \approx \alpha \log N + \beta, \tag{15}$$

where we used $\log(a + b) = \log(1 + a/b) + \log(b)$ to separate out the two terms and $\beta = -\log E'' - \log A$ with the $1 + a/b$ term hidden in E'' along with. Now we estimate α, β for both sets of models from the losses in Table 2 where we use for N the active parameters in each model. We estimate the slope α and the intercept β for both the Transformer as well as the 2-simplicial Transformer in Table 3. We see that 2-simplicial attention has a steeper slope α, i.e. a higher exponent in its scaling law compared to dot product attention Transformer (Vaswani et al., 2017).

9.1 Ablation on Trilinear Forms

We conduct an ablation study to evaluate the effectiveness of different trilinear functions within the 2-simplicial attention mechanism. The goal is to determine which mathematical construction offers the best inductive bias for the logical and reasoning tasks our model is evaluated on. All experiments are performed on a 125M parameter model, and we report the negative log-likelihood on several downstream benchmarks.

Model	GSM8k		MMLU		MMLU-pro		MBPP	
	α	β	α	β	α	β	α	β
Transformer	0.1420	-1.8280	0.1256	-2.1606	0.0901	-1.7289	0.1720	-2.2569
2-simplicial	0.1683	-2.3939	0.1364	-2.3960	0.1083	-2.1181	0.1837	-2.5201
Δ(%)	18.5%		8.5%		20.2%		6.8%	

Table 3: Estimates of the power law coefficients α and β for the Transformer (Vaswani et al., 2017) and 2-simplicial attention.

Model	GSM8k		MMLU		MMLU-pro		MBPP	
	R^2	residual	R^2	residual	R^2	residual	R^2	residual
Transformer	0.9998	2.8×10^{-6}	0.9995	4.7×10^{-6}	0.9972	1.5×10^{-5}	0.9962	7.5×10^{-5}
2-simplicial	0.9974	4.9×10^{-5}	0.9989	1.3×10^{-5}	0.9999	4.6×10^{-8}	0.9999	1.5×10^{-6}

Table 4: R^2 and residuals measuring goodness of fit for Table 3.

The results, summarized in Table 5, show a clear advantage for using a determinant-based trilinear form. This approach consistently outperforms both the standard dot-product attention baseline and the *unsigned scalar triple product* proposed by Clift et al. (2020) across most reasoning and knowledge-intensive benchmarks, including MBPP, MMLU Pro, MMLU, and ARC.

Table 5: Ablation study on different trilinear forms for 2-simplicial attention in a 125M LLaMA model. Lower values indicate better performance (negative log-likelihood).

Experiment	MBPP	GSM8K	MMLU Pro	MMLU	ARC
Trilinear Product	0.3352	0.4363	1.0400	0.8028	0.6740
Unsigned Scalar Triple Product	0.3412	**0.4294**	1.0437	0.8065	0.6845
Determinant	**0.3296**	0.4329	**1.0323**	**0.7982**	**0.6700**
Dot Product (Baseline)	0.3377	0.4426	1.0477	0.8089	0.6901

This performance gain is noteworthy given the computational trade-offs. While the simple trilinear product is the most efficient, the determinant-based form requires approximately twice the floating-point operations (FLOPs). The unsigned scalar triple product, as defined by its polynomial form in Clift et al. (2020), is 3 times more computationally intensive.

10 Conclusion

We show that a similar sized 2-simplicial attention (Clift et al., 2020) improves on dot product attention of Vaswani et al. (2017) by improving the negative log likelihood on reasoning, math and coding problems (see Table 2). We quantify this explicitly in Table 3 by demonstrating that 2-simplicial attention changes the exponent corresponding to parameters in the scaling law of Equation 15: in particular it has a higher α for reasoning and coding tasks compared to the Transformer (Vaswani et al., 2017) which leads to more favorable scaling under token constraints. Furthermore, the percentage increase in the scaling exponent α is higher for less saturated and more challenging benchmarks such as MMLU-pro and GSM8k.

While 2-simplicial attention improves the exponent in the scaling laws, we should caveat that the technique maybe more useful when we are in the regime when token efficiency becomes more important. Our Triton kernel while efficient for prototyping is still far away from being used in production. More work in co-designing the implementation of 2-simplicial attention tailored to the specific hardware accelerator is needed in the future.

We hope that scaling 2-simplicial Transformers could unlock significant improvements in downstream performance on reasoning-heavy tasks, helping to overcome the current limitations of pre- training scalability. Furthermore, we believe that developing a specialized and efficient implementation is key to fully unlocking the potential of this architecture.

REFERENCES

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Joshua Ainslie, James Lee-Thorp, Michiel De Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv preprint arXiv:2305.13245*, 2023.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Yasaman Bahri, Ethan Dyer, Jared Kaplan, Jaehoon Lee, and Utkarsh Sharma. Explaining neural scaling laws. *Proceedings of the National Academy of Sciences*, 121(27):e2311878121, 2024.

Leon Bergen, Timothy O'Donnell, and Dzmitry Bahdanau. Systematic generalization with edge transformers. *Advances in Neural Information Processing Systems*, 34:1390–1402, 2021.

David Brandfonbrener, Nikhil Anand, Nikhil Vyas, Eran Malach, and Sham Kakade. Loss-to-loss prediction: Scaling laws for all datasets. *arXiv preprint arXiv:2411.12925*, 2024.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Thomas F Burns and Tomoki Fukai. Simplicial hopfield networks. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=_QLsH8gatwx.

James Clift, Dmitry Doryn, Daniel Murfet, and James Wallbridge. Logic and the 2-simplicial transformer, 2020. URL https://openreview.net/forum?id=rkecJ6VFvr.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

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

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. Universal transformers. *arXiv preprint arXiv:1807.03819*, 2018.

Katie Everett. Observation on scaling laws, May 2025. URL https://x.com/_katieeverett/status/1925665335727808651. [Tweet].

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

Charles R Harris, K Jarrod Millman, Stéfan J Van Der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. Array programming with numpy. *Nature*, 585(7825):357–362, 2020.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300*, 2020.

Joel Hestness, Sharan Narang, Newsha Ardalani, Gregory Diamos, Heewoo Jun, Hassan Kianinejad, Md Mostofa Ali Patwary, Yang Yang, and Yanqi Zhou. Deep learning scaling is predictable, empirically. *arXiv preprint arXiv:1712.00409*, 2017.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022.

Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

Michael I Jordan and Robert A Jacobs. Hierarchical mixtures of experts and the em algorithm. *Neural computation*, 6(2):181–214, 1994.

John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, et al. Highly accurate protein structure prediction with alphafold. *nature*, 596(7873):583–589, 2021.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and Francois Fleuret. Transformers are rnns: fast autoregressive transformers with linear attention. In *Proceedings of the 37th International Conference on Machine Learning*, ICML'20. JMLR.org, 2020.

Alexander Kozachinskiy, Felipe Urrutia, Hector Jimenez, Tomasz Steifer, Germán Pizarro, Matías Fuentes, Francisco Meza, Cristian B Calderon, and Cristóbal Rojas. Strassen attention: Unlocking compositional abilities in transformers based on a new lower bound method. *arXiv preprint arXiv:2501.19215*, 2025.

Ilya Loshchilov, Frank Hutter, et al. Fixing weight decay regularization in adam. *arXiv preprint arXiv:1711.05101*, 5:5, 2017.

Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pp. 1–15, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368735. doi: 10.1145/3341301.3359646. URL https://doi.org/10.1145/3341301.3359646.

Niki Parmar, Ashish Vaswani, Jakob Uszkoreit, Lukasz Kaiser, Noam Shazeer, Alexander Ku, and Dustin Tran. Image transformer. In *International conference on machine learning*, pp. 4055–4064. PMLR, 2018.

Aurko Roy, Mohammad Saffar, Ashish Vaswani, and David Grangier. Efficient content-based sparse attention with routing transformers. *Transactions of the Association for Computational Linguistics*, 9:53–68, 2021.

Aurko Roy, Rohan Anil, Guangda Lai, Benjamin Lee, Jeffrey Zhao, Shuyuan Zhang, Shibo Wang, Ye Zhang, Shen Wu, Rigel Swavely, et al. N-grammer: Augmenting transformers with latent n-grams. *arXiv preprint arXiv:2207.06366*, 2022.

Clayton Sanford, Daniel J Hsu, and Matus Telgarsky. Representational strengths and limitations of transformers. *Advances in Neural Information Processing Systems*, 36:36677–36707, 2023.

Nikunj Saunshi, Nishanth Dikkala, Zhiyuan Li, Sanjiv Kumar, and Sashank J Reddi. Reasoning with latent thoughts: On the power of looped transformers. *arXiv preprint arXiv:2502.17416*, 2025.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

Xuyang Shen, Dong Li, Ruitao Leng, Zhen Qin, Weigao Sun, and Yiran Zhong. Scaling laws for linear complexity language models. *arXiv preprint arXiv:2406.16690*, 2024.

David So, Wojciech Mańke, Hanxiao Liu, Zihang Dai, Noam Shazeer, and Quoc V Le. Searching for efficient transformers for language modeling. *Advances in neural information processing systems*, 34:6010–6022, 2021.

Ben Sorscher, Robert Geirhos, Shashank Shekhar, Surya Ganguli, and Ari Morcos. Beyond neural scaling laws: beating power law scaling via data pruning. *Advances in Neural Information Processing Systems*, 35:19523–19536, 2022.

Gilbert Strang. *Introduction to linear algebra*. SIAM, 2022.

Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.

Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

Vladimir Vapnik. On the uniform convergence of relative frequencies of events to their probabilities. In *Doklady Akademii Nauk USSR*, volume 181, pp. 781–787, 1968.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Ruoxi Wang, Rakesh Shivanna, Derek Cheng, Sagar Jain, Dong Lin, Lichan Hong, and Ed Chi. Dcn v2: Improved deep & cross network and practical lessons for web-scale learning to rank systems. In *Proceedings of the web conference 2021*, pp. 1785–1797, 2021.

Yubo Wang, Xueguang Ma, Ge Zhang, Yuansheng Ni, Abhranil Chandra, Shiguang Guo, Weiming Ren, Aaran Arulraj, Xuan He, Ziyan Jiang, et al. Mmlu-pro: A more robust and challenging multi-task language understanding benchmark. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2024.

Liu Yang, Kangwook Lee, Robert Nowak, and Dimitris Papailiopoulos. Looped transformers are better at learning learning algorithms. *arXiv preprint arXiv:2311.12424*, 2023.

Jingyang Yuan, Huazuo Gao, Damai Dai, Junyu Luo, Liang Zhao, Zhengyan Zhang, Zhenda Xie, YX Wei, Lean Wang, Zhiping Xiao, et al. Native sparse attention: Hardware-aligned and natively trainable sparse attention. *arXiv preprint arXiv:2502.11089*, 2025.

Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. Big bird: Transformers for longer sequences. *Advances in neural information processing systems*, 33:17283–17297, 2020.

A ROTATION INVARIANT TRILINEAR FORMS

A.1 PROOF FOR THEOREM 5.1

We define the embedding functions for the Query and Key vectors such that their interaction within the Sum-of-Determinants attention mechanism computes the `Match3` function. To handle cases where no match exists, we use a 7-dimensional embedding where the 7th dimension acts as a selector for a "blank pair" option, a technique adapted from `Match2` construction in Sanford et al. (2023).

The construction for regular token pairs is based on the mathematical identity:

$$\cos(\theta_1 + \theta_2 + \theta_3) = \det(M_1) + \det(-M_2), \tag{16}$$

where the matrices $M_1, M_2 \in \mathbb{R}^{3\times 3}$ are defined as:

$$M_1 = \begin{pmatrix} \cos(\theta_1) & \sin(\theta_1) & 0 \\ \sin(\theta_2) & \cos(\theta_2) & 0 \\ 0 & 0 & \cos(\theta_3) \end{pmatrix}, \quad -M_2 = \begin{pmatrix} -\sin(\theta_1) & \cos(\theta_1) & 0 \\ -\sin(\theta_2) & -\cos(\theta_2) & 0 \\ 0 & 0 & -\sin(\theta_3) \end{pmatrix}$$

Let $\theta_k = \frac{2\pi x_k}{M}$. We define the 7-dimensional query vector $\boldsymbol{q}_i$ and key vectors $\mathbf{k}_{j_1}, \mathbf{k}'_{j_2}$ via an input MLP ϕ and matrices Q, K, K'. Let c be a large scaling constant.

The 7-dimensional query vector $q_i = Q\phi(x_i)$ is defined as:

$$\boldsymbol{q}_i = (c\cos(\theta_i), c\sin(\theta_i), 0, -c\sin(\theta_i), c\cos(\theta_i), 0, c)$$

The key vectors $\mathbf{k}_{j_1} = K\phi(x_{j_1})$ and $\mathbf{k}'_{j_2} = K'\phi(x_{j_2})$ for regular tokens are defined as:

$$\mathbf{k}_{j_1} = (\sin(\theta_{j_1}), \cos(\theta_{j_1}), 0, -\sin(\theta_{j_1}), -\cos(\theta_{j_1}), 0, 0)$$

$$\mathbf{k}'_{j_2} = (0, 0, \cos(\theta_{j_2}), 0, 0, -\sin(\theta_{j_2}), 0)$$

The attention score is computed via a hybrid mechanism:

1. **For regular pairs** (j_1, j_2), the score is the sum of determinants of two 3D chunks formed from the first 6 dimensions of the vectors. The 7th dimension of the keys is 0, so it is ignored in this term.

$$\begin{aligned} A_{i,j_1,j_2} &= \det(\boldsymbol{q}_i[:3], k_{j_1}[:3], \mathbf{k}'_{j_2}[:3]) + \det(\boldsymbol{q}_i[3:6], k_{j_1}[3:6], \mathbf{k}'_{j_2}[3:6]) \\ &= c \cdot (\det(M_1) + \det(-M_2)) \quad \text{(from (16))} \\ &= c \cdot \cos\left(\frac{2\pi(x_i + x_{j_1} + x_{j_2})}{M}\right) \quad (\text{since } \theta_i = 2\pi x_k/M), \end{aligned}$$

   where $\boldsymbol{q}_i[l : l+m] = \{(\boldsymbol{q}_i)_l, \ldots, (\boldsymbol{q}_i)_{l+m-1}\}$, denotes array slicing.

2. **For the blank pair**, the score is computed using the 7th dimension. It is the dot product of the query vector $\boldsymbol{q}_i$ and a fixed key vector $\mathbf{k}_{\text{blank}} = (0, 0, 0, 0, 0, 0, 1)$:

$$A_{i,\text{blank}} = \boldsymbol{q}_i \cdot \mathbf{k}_{\text{blank}} = c$$

As a result, the attention score is maximized to a value of c if and only if $x_i + x_{j_1} + x_{j_2} = 0 \pmod{M}$. The blank pair also receives a score of c. For any non-matching triple, the score is strictly less than c.

The value vectors are defined by matrices V and V'.

- For any **regular token** x_j, we set its value embeddings to be $V\phi(x_j) = 1$ and $V'\phi(x_j) = 1$. The resulting value for the pair (j_1, j_2) in the final value matrix is their Kronecker product, which is 1.
- For the **blank pair**, the corresponding value is 0.

Let β_i be the number of pairs (j_1, j_2) that form a match with x_i. The softmax function distributes the attention weight almost exclusively among the entries with a score of c.

- If no match exists ($\beta_i = 0$), the blank pair receives all the attention, and the output is ≈ 0 since its value is 0.

- If at least one match exists ($\beta_i \geq 1$), the attention is distributed among the β_i matching pairs and the 1 blank pair. The output of the attention layer will be approximately $\frac{\beta_i \cdot (1) + 1 \cdot (0)}{\beta_i + 1} = \frac{\beta_i}{\beta_i + 1}$.

The final step is to design an output MLP ψ such that $\psi(z) = 1$ if $z \geq 1/2$ and $\psi(z) = 0$ otherwise, which is straightforward to implement.

B Backward Pass Computation

For completeness we provide the backwards pass terms explicitly. Note that each compuation would need aggregation over three different dimension orderings.

$$dV_{jd} = \sum_{i,k} (A_{ijk} \cdot dO_{id} \cdot V'_{kd}) \tag{17}$$

$$dV'_{kd} = \sum_{i,j} (A_{ijk} \cdot dO_{id} \cdot V_{jd}) \tag{18}$$

$$dP_{ijk} = \sum_{d} (dO_{id} \cdot V_{jd} \cdot V'_{kd}) \tag{19}$$

$$dS = \mathrm{dsoftmax}_{jk}(dP) \tag{20}$$

$$dK_{jd} = \sum_{i,k} (Q_{id} \cdot dS_{ijk} \cdot K'_{kd}) \tag{21}$$

$$dK'_{kd} = \sum_{i,k} (Q_{id} \cdot dS_{ijk} \cdot K_{jd}) \tag{22}$$

$$dQ_{id} = \sum_{j,k} (dS_{ijk} \cdot K_{jd} \cdot K'_{kd}) \tag{23}$$

C Triton Kernels

We document here the forward and backward passes for the 2-simplicial attention mechanism. We will release the complete kernel on Github upon acceptance.

```
@triton.autotune(
    configs=[
        Config(
            {
                "BLOCK_SIZE_Q": 64,
                "BLOCK_SIZE_KV": 32,
                "num_stages": 1,
            },
            num_warps=4,
        )
    ],
    key=["HEAD_DIM"],
)
@triton.jit
def two_simplicial_attn_fwd_kernel(
    Q_ptr,  # [b, s, k, h]
    K1_ptr,  # [b, s, k, h]
    K2_ptr,  # [b, s, k, h]
    V1_ptr,  # [b, s, k, h]
    V2_ptr,  # [b, s, k, h]
    O_ptr,  # [b, s, k, h]
    M_ptr,  # [b, k, s]
    bs,
    seq_len,
```

```
    num_heads,
    head_dim,
    w1: tl.constexpr,
    w2: tl.constexpr,
    q_stride_b,
    q_stride_s,
    q_stride_k,
    q_stride_h,
    k1_stride_b,
    k1_stride_s,
    k1_stride_k,
    k1_stride_h,
    k2_stride_b,
    k2_stride_s,
    k2_stride_k,
    k2_stride_h,
    v1_stride_b,
    v1_stride_s,
    v1_stride_k,
    v1_stride_h,
    v2_stride_b,
    v2_stride_s,
    v2_stride_k,
    v2_stride_h,
    out_stride_b,
    out_stride_s,
    out_stride_k,
    out_stride_h,
    m_stride_b,
    m_stride_k,
    m_stride_s,
    BLOCK_SIZE_Q: tl.constexpr,
    BLOCK_SIZE_KV: tl.constexpr,
    HEAD_DIM: tl.constexpr,
    INPUT_PRECISION: tl.constexpr,
    SM_SCALE: tl.constexpr,
    K2_BIAS: tl.constexpr,
    V2_BIAS: tl.constexpr,
    num_stages: tl.constexpr,
):
    data_dtype = tl.bfloat16
    compute_dtype = tl.float32
    gemm_dtype = tl.bfloat16

    q_start = tl.program_id(0) * BLOCK_SIZE_Q
    q_end = q_start + BLOCK_SIZE_Q
    bk = tl.program_id(1)
    offs_b = bk // num_heads
    offs_k = bk % num_heads

    qkv_offs_bk = offs_b * q_stride_b + offs_k * q_stride_k

    Q_ptr += qkv_offs_bk
    K1_ptr += qkv_offs_bk
    K2_ptr += qkv_offs_bk
    V1_ptr += qkv_offs_bk
    V2_ptr += qkv_offs_bk
    O_ptr += qkv_offs_bk
    M_ptr += offs_b * m_stride_b + offs_k * m_stride_k

    m_i = tl.zeros((BLOCK_SIZE_Q,), dtype=compute_dtype) - float("inf")
    l_i = tl.zeros((BLOCK_SIZE_Q,), dtype=compute_dtype)
    acc = tl.zeros((BLOCK_SIZE_Q, HEAD_DIM), dtype=compute_dtype)

    q_offs_s = q_start + tl.arange(0, BLOCK_SIZE_Q)
```

```
    qkv_offs_h = tl.arange(0, HEAD_DIM)
    q_mask_s = q_offs_s < seq_len
    qkv_mask_h = qkv_offs_h < head_dim
    q_offs = q_offs_s[:, None] * q_stride_s + qkv_offs_h[None, :] *
        q_stride_h
    q_mask = q_mask_s[:, None] & (qkv_mask_h[None, :])

    q_tile = tl.load(Q_ptr + q_offs, mask=q_mask).to(
        compute_dtype
    )  # [BLOCK_SIZE_Q, HEAD_DIM]
    softmax_scale = tl.cast(SM_SCALE, gemm_dtype)

    for kv1_idx in tl.range(tl.maximum(0, q_start - w1), tl.minimum(
        seq_len, q_end)):
        k1_offs = kv1_idx * k1_stride_s + qkv_offs_h * k1_stride_h
        k1_tile = (tl.load(K1_ptr + k1_offs, mask=qkv_mask_h).to(
            compute_dtype))[
            None, :
        ]  # [1, HEAD_DIM]
        qk1 = q_tile * k1_tile  # [BLOCK_SIZE_Q, HEAD_DIM]
        qk1 = qk1.to(gemm_dtype)

        v1_offs = kv1_idx * v1_stride_s + qkv_offs_h * v1_stride_h
        v1_tile = (tl.load(V1_ptr + v1_offs, mask=qkv_mask_h).to(
            compute_dtype))[
            None, :
        ]  # [1, HEAD_DIM]

        for kv2_idx in tl.range(
            tl.maximum(0, q_start - w2),
            tl.minimum(seq_len, q_end),
            BLOCK_SIZE_KV,
            num_stages=num_stages,
        ):
            kv2_offs_s = kv2_idx + tl.arange(0, BLOCK_SIZE_KV)
            kv2_mask_s = kv2_offs_s < seq_len
            k2t_mask = kv2_mask_s[None, :] & qkv_mask_h[:, None]
            v2_mask = kv2_mask_s[:, None] & qkv_mask_h[None, :]
            k2_offs = (
                kv2_offs_s[None, :] * k2_stride_s + qkv_offs_h[:, None] *
                     k2_stride_h
            )
            v2_offs = (
                kv2_offs_s[:, None] * v2_stride_s + qkv_offs_h[None, :] *
                     v2_stride_h
            )
            k2t_tile = tl.load(K2_ptr + k2_offs, mask=k2t_mask).to(
                compute_dtype
            )  # [HEAD_DIM, BLOCK_SIZE_KV]
            v2_tile = tl.load(V2_ptr + v2_offs, mask=v2_mask).to(
                compute_dtype
            )  # [BLOCK_SIZE_KV, HEAD_DIM]
            k2t_tile += K2_BIAS
            v2_tile += V2_BIAS
            k2t_tile = k2t_tile.to(gemm_dtype)
            v2_tile = v2_tile.to(compute_dtype)

            qk = tl.dot(
                qk1 * softmax_scale,
                k2t_tile,
                input_precision="tf32",  # INPUT_PRECISION,
                out_dtype=tl.float32,
            )  # [BLOCK_SIZE_Q, BLOCK_SIZE_KV]

            qk_mask = q_mask_s[:, None] & kv2_mask_s[None, :]
```

```
            # Mask for q_idx - w1 < kv1_idx <= q_idx
            # and q_idx - w2 < kv2_offs_s <= q_idx
            kv1_local_mask = ((q_offs_s[:, None] - w1) < kv1_idx) & (
                kv1_idx <= q_offs_s[:, None]
            )
            kv2_local_mask = ((q_offs_s[:, None] - w2) < kv2_offs_s[None,
                 :]) & (
                kv2_offs_s[None, :] <= q_offs_s[:, None]
            )
            qk_mask &= kv1_local_mask & kv2_local_mask
            qk += tl.where(qk_mask, 0, -1.0e38)

            m_ij = tl.maximum(m_i, tl.max(qk, 1))
            p = tl.math.exp(qk - m_ij[:, None])
            l_ij = tl.sum(p, 1)
            alpha = tl.math.exp(m_i - m_ij)
            l_i = l_i * alpha + l_ij
            acc = acc * alpha[:, None]

            v12_tile = v1_tile * v2_tile  # [BLOCK_SIZE_KV, HEAD_DIM]
            acc += tl.dot(
                p.to(gemm_dtype),
                v12_tile.to(gemm_dtype),
                input_precision="ieee",  # INPUT_PRECISION,
                out_dtype=tl.float32,
            )

            m_i = m_ij
    acc = acc / l_i[:, None]

    acc = tl.where(q_mask, acc, 0.0)
    acc = acc.to(data_dtype)
    out_offs = q_offs_s[:, None] * out_stride_s + qkv_offs_h[None, :] *
         out_stride_h
    tl.store(O_ptr + out_offs, acc, mask=q_mask)

    m = m_i + tl.log(l_i)

    m_offs = q_offs_s * m_stride_s
    m_mask = q_offs_s < seq_len
    tl.store(M_ptr + m_offs, m, mask=m_mask)
```

Listing 1: Forward pass for 2-simplicial attention.

D Triton Kernel: Backward pass for 2-simplicial Attention

```
@triton.jit
def two_simplicial_attn_bwd_kv1_kernel(
    Q_ptr,  # [b, s, k, h]
    K1_ptr,  # [b, s, k, h]
    K2_ptr,  # [b, s, k, h]
    V1_ptr,  # [b, s, k, h]
    V2_ptr,  # [b, s, k, h]
    dO_ptr,  # [b, s, k, h]
    M_ptr,  # [b, k, s]
    D_ptr,  # [b, k, s]
    dQ_ptr,  # [b, s, k, h]
    dK1_ptr,  # [b, s, k, h]
    dV1_ptr,  # [b, s, k, h]
    # Skip writing dk2, dv2 for now.
    bs,
    seq_len,
    num_heads,
```

```
    head_dim,
    w1,  # Q[i]: KV1(i-w1,i]
    w2,  # Q[i]: KV2(i-w2,i]
    q_stride_b,
    q_stride_s,
    q_stride_k,
    q_stride_h,
    k1_stride_b,
    k1_stride_s,
    k1_stride_k,
    k1_stride_h,
    k2_stride_b,
    k2_stride_s,
    k2_stride_k,
    k2_stride_h,
    v1_stride_b,
    v1_stride_s,
    v1_stride_k,
    v1_stride_h,
    v2_stride_b,
    v2_stride_s,
    v2_stride_k,
    v2_stride_h,
    dO_stride_b,
    dO_stride_s,
    dO_stride_k,
    dO_stride_h,
    m_stride_b,
    m_stride_k,
    m_stride_s,
    d_stride_b,
    d_stride_k,
    d_stride_s,
    dq_stride_b,
    dq_stride_s,
    dq_stride_k,
    dq_stride_h,
    dk1_stride_b,
    dk1_stride_s,
    dk1_stride_k,
    dk1_stride_h,
    dv1_stride_b,
    dv1_stride_s,
    dv1_stride_k,
    dv1_stride_h,
    BLOCK_SIZE_Q: tl.constexpr,
    BLOCK_SIZE_KV: tl.constexpr,
    HEAD_DIM: tl.constexpr,
    SM_SCALE: tl.constexpr,
    K2_BIAS: tl.constexpr,
    V2_BIAS: tl.constexpr,
    COMPUTE_DQ: tl.constexpr,
    num_stages: tl.constexpr,
    is_flipped: tl.constexpr,
):
    data_dtype = tl.bfloat16
    compute_dtype = tl.float32
    gemm_dtype = tl.bfloat16

    kv1_start = tl.program_id(0) * BLOCK_SIZE_KV
    kv1_end = kv1_start + BLOCK_SIZE_KV
    bk = tl.program_id(1)
    offs_b = bk // num_heads
    offs_k = bk % num_heads

```

```
    qkv_offs_bk = offs_b * q_stride_b + offs_k * q_stride_k
    Q_ptr += qkv_offs_bk
    K1_ptr += qkv_offs_bk
    K2_ptr += qkv_offs_bk
    V1_ptr += qkv_offs_bk
    V2_ptr += qkv_offs_bk

    dO_ptr += offs_b * dO_stride_b + offs_k * dO_stride_k
    M_ptr += offs_b * m_stride_b + offs_k * m_stride_k
    D_ptr += offs_b * d_stride_b + offs_k * d_stride_k
    dK1_ptr += offs_b * dk1_stride_b + offs_k * dk1_stride_k
    dV1_ptr += offs_b * dv1_stride_b + offs_k * dv1_stride_k
    if COMPUTE_DQ:
        dQ_ptr += offs_b * dq_stride_b + offs_k * dq_stride_k

    softmax_scale = tl.cast(SM_SCALE, gemm_dtype)
    qkv_offs_h = tl.arange(0, HEAD_DIM)
    qkv_mask_h = qkv_offs_h < head_dim

    kv1_offs_s = kv1_start + tl.arange(0, BLOCK_SIZE_KV)

    k1_offs = kv1_offs_s[:, None] * k1_stride_s + qkv_offs_h[None, :] *
        k1_stride_h
    kv1_mask_s = kv1_offs_s < seq_len
    kv1_mask = kv1_mask_s[:, None] & qkv_mask_h[None, :]
    k1_tile = tl.load(K1_ptr + k1_offs, mask=kv1_mask).to(
        compute_dtype
    )  # [BLOCK_SIZE_KV, HEAD_DIM]
    v1_offs = kv1_offs_s[:, None] * v1_stride_s + qkv_offs_h[None, :] *
        v1_stride_h
    v1_tile = tl.load(V1_ptr + v1_offs, mask=kv1_mask).to(
        compute_dtype
    )  # [BLOCK_SIZE_KV, HEAD_DIM]
    if is_flipped:
        k1_tile += K2_BIAS
        v1_tile += V2_BIAS
    dv1 = tl.zeros((BLOCK_SIZE_KV, HEAD_DIM), compute_dtype)
    dk1 = tl.zeros((BLOCK_SIZE_KV, HEAD_DIM), compute_dtype)
    # for kv2_idx in tl.range(0, seq_len):
    # kv1 - w2 < kv2 <= kv1 + w1
    for kv2_idx in tl.range(
        tl.maximum(0, kv1_start - w2), tl.minimum(seq_len, kv1_end + w1)
    ):
        k2_offs = kv2_idx * k2_stride_s + qkv_offs_h * k2_stride_h
        k2_tile = (tl.load(K2_ptr + k2_offs, mask=qkv_mask_h).to(
            compute_dtype))[
            None, :
        ]  # [1, HEAD_DIM]
        v2_offs = kv2_idx * v2_stride_s + qkv_offs_h * v2_stride_h
        v2_tile = (tl.load(V2_ptr + v2_offs, mask=qkv_mask_h).to(
            compute_dtype))[
            None, :
        ]  # [1, HEAD_DIM]
        if not is_flipped:
            k2_tile += K2_BIAS
            v2_tile += V2_BIAS
        k1k2 = k1_tile * k2_tile  # [BLOCK_SIZE_KV, HEAD_DIM]
        v1v2 = v1_tile * v2_tile  # [BLOCK_SIZE_KV, HEAD_DIM]
        k1k2 = k1k2.to(gemm_dtype)
        v1v2 = v1v2.to(gemm_dtype)
        # kv1 <= q < kv1 + w1
        # kv2 <= q < kv2 + w2
        q_start = tl.maximum(kv1_start, kv2_idx)
        q_end = tl.minimum(seq_len, tl.minimum(kv1_end + w1, kv2_idx + w2
            ))
```

```
        for q_idx in tl.range(q_start, q_end, BLOCK_SIZE_Q):
            # Load qt, m, d, dO
            q_offs_s = q_idx + tl.arange(0, BLOCK_SIZE_Q)
            q_offs = q_offs_s[None, :] * q_stride_s + qkv_offs_h[:, None]
                 * q_stride_h
            q_mask_s = q_offs_s < seq_len
            qt_mask = q_mask_s[None, :] & qkv_mask_h[:, None]
            qt_tile = tl.load(Q_ptr + q_offs, mask=qt_mask).to(
                gemm_dtype
            )  # [HEAD_DIM, BLOCK_SIZE_Q]
            m_offs = q_offs_s * m_stride_s
            m_tile = tl.load(M_ptr + m_offs, mask=q_mask_s).to(
                compute_dtype)[
                None, :
            ]  # [1, BLOCK_SIZE_Q]
            d_offs = q_offs_s * d_stride_s
            d_tile = tl.load(D_ptr + d_offs, mask=q_mask_s).to(
                compute_dtype)[
                None, :
            ]  # [1, BLOCK_SIZE_Q]
            dO_offs = (
                q_offs_s[:, None] * dO_stride_s + qkv_offs_h[None, :] *
                    dO_stride_h
            )
            dO_tile = tl.load(
                dO_ptr + dO_offs, mask=q_mask_s[:, None] & qkv_mask_h[
                    None, :]
            ).to(compute_dtype)  # [BLOCK_SIZE_Q, HEAD_DIM]
            if COMPUTE_DQ:
                dq = tl.zeros((BLOCK_SIZE_Q, HEAD_DIM), tl.float32)
            # Compute dv1.
            # [KV, D] @ [D, Q] => [KV, Q]
            qkkT = tl.dot(
                k1k2, qt_tile * softmax_scale, out_dtype=tl.float32
            )  # [BLOCK_SIZE_KV, BLOCK_SIZE_Q]

            # Mask qkkT to -inf.
            kv1_local_mask = ((q_offs_s[None, :] - w1) < kv1_offs_s[:,
                None]) & (
                kv1_offs_s[:, None] <= q_offs_s[None, :]
            )
            kv2_local_mask = ((q_offs_s - w2) < kv2_idx) & (kv2_idx <=
                q_offs_s)
            local_mask = (
                kv1_local_mask & kv2_local_mask[None, :]
            )  # [BLOCK_SIZE_KV, BLOCK_SIZE_Q]
            qkkT = tl.where(local_mask, qkkT, -1.0e38)

            pT = tl.exp(qkkT - m_tile)  # [BLOCK_SIZE_KV, BLOCK_SIZE_Q]
            pT = tl.where(local_mask, pT, 0.0)
            dOv2 = dO_tile * v2_tile  # [BLOCK_SIZE_Q, HEAD_DIM]
            dv1 += tl.dot(
                pT.to(gemm_dtype), dOv2.to(gemm_dtype), out_dtype=tl.
                    float32
            )  # [BLOCK_SIZE_KV, HEAD_DIM]

            dpT = tl.dot(
                v1v2, tl.trans(dO_tile.to(gemm_dtype)), out_dtype=tl.
                    float32
            )  # [BLOCK_SIZE_KV, BLOCK_SIZE_Q]
            dsT = pT * (dpT - d_tile)  # [BLOCK_SIZE_KV, BLOCK_SIZE_Q]
            dsT = tl.where(local_mask, dsT, 0.0)
            dsT = dsT.to(gemm_dtype)

            dk1 += (
```

```
                tl.dot(dsT, tl.trans(qt_tile), out_dtype=tl.float32)
                * k2_tile.to(tl.float32)
                * softmax_scale
            )
            if COMPUTE_DQ:
                # dq[q, d] = dsT.T[q, kv1] @ k1k2[kv1, d]
                dq += (
                    tl.dot(tl.trans(dsT), k1k2, out_dtype=tl.float32) *
                        softmax_scale
                )  # [BLOCK_SIZE_Q, HEAD_DIM]
                dq_offs = (
                    q_offs_s[:, None] * dq_stride_s + qkv_offs_h[None, :]
                         * dq_stride_h
                )
                tl.atomic_add(
                    dQ_ptr + dq_offs, dq, mask=q_mask_s[:, None] &
                        qkv_mask_h[None, :]
                )
    dv1_offs = kv1_offs_s[:, None] * dv1_stride_s + qkv_offs_h[None, :] *
         dv1_stride_h
    dk1_offs = kv1_offs_s[:, None] * dk1_stride_s + qkv_offs_h[None, :] *
         dk1_stride_h
    tl.store(dV1_ptr + dv1_offs, dv1.to(data_dtype), mask=kv1_mask)
    tl.store(dK1_ptr + dk1_offs, dk1.to(data_dtype), mask=kv1_mask)
```

Listing 2: Backward pass for 2-simplicial attention.

```
@triton.autotune(
    configs=[
        Config(
            {
                "BLOCK_SIZE_Q": 32,
                "BLOCK_SIZE_KV2": 64,
                "num_stages": 1,
            },
            num_warps=4,
        )
    ],
    key=["HEAD_DIM"],
)
@triton.jit
def two_simplicial_attn_bwd_kv2q_kernel(
    Q_ptr,  # [b, s, k, h]
    K1_ptr,  # [b, s, k, h]
    K2_ptr,  # [b, s, k, h]
    V1_ptr,  # [b, s, k, h]
    V2_ptr,  # [b, s, k, h]
    dO_ptr,  # [b, s, k, h]
    M_ptr,  # [b, k, s]
    D_ptr,  # [b, k, s]
    dQ_ptr,  # [b, s, k, h]
    dK2_ptr,  # [b, s, k, h]
    dV2_ptr,  # [b, s, k, h]
    bs,
    seq_len,
    num_heads,
    head_dim,
    w1,  # Q[i]: KV1(i-w1,i]
    w2,  # Q[i]: KV2(i-w2,i]
    q_stride_b,
    q_stride_s,
    q_stride_k,
    q_stride_h,
    k1_stride_b,
    k1_stride_s,
```

```
    k1_stride_k,
    k1_stride_h,
    k2_stride_b,
    k2_stride_s,
    k2_stride_k,
    k2_stride_h,
    v1_stride_b,
    v1_stride_s,
    v1_stride_k,
    v1_stride_h,
    v2_stride_b,
    v2_stride_s,
    v2_stride_k,
    v2_stride_h,
    dO_stride_b,
    dO_stride_s,
    dO_stride_k,
    dO_stride_h,
    m_stride_b,
    m_stride_k,
    m_stride_s,
    d_stride_b,
    d_stride_k,
    d_stride_s,
    dq_stride_b,
    dq_stride_s,
    dq_stride_k,
    dq_stride_h,
    dk2_stride_b,
    dk2_stride_s,
    dk2_stride_k,
    dk2_stride_h,
    dv2_stride_b,
    dv2_stride_s,
    dv2_stride_k,
    dv2_stride_h,
    BLOCK_SIZE_Q: tl.constexpr,
    BLOCK_SIZE_KV2: tl.constexpr,
    HEAD_DIM: tl.constexpr,
    SM_SCALE: tl.constexpr,
    K2_BIAS: tl.constexpr,
    V2_BIAS: tl.constexpr,
    num_stages: tl.constexpr,
    IS_SECOND_PASS: tl.constexpr,
):
    assert BLOCK_SIZE_KV2 == BLOCK_SIZE_Q + w2
    data_dtype = tl.bfloat16
    compute_dtype = tl.float32
    gemm_dtype = tl.bfloat16

    # First pass does even tiles, second pass does odd tiles.
    q_start = tl.program_id(0) * BLOCK_SIZE_KV2
    if IS_SECOND_PASS:
        q_start += BLOCK_SIZE_Q
    q_end = q_start + BLOCK_SIZE_Q
    kv2_start = q_start - w2

    bk = tl.program_id(1)
    offs_b = bk // num_heads
    offs_k = bk % num_heads

    qkv_offs_bk = offs_b * q_stride_b + offs_k * q_stride_k
    Q_ptr += qkv_offs_bk
    K1_ptr += qkv_offs_bk
    K2_ptr += qkv_offs_bk
```

```
    V1_ptr += qkv_offs_bk
    V2_ptr += qkv_offs_bk

    dO_ptr += offs_b * dO_stride_b + offs_k * dO_stride_k
    M_ptr += offs_b * m_stride_b + offs_k * m_stride_k
    D_ptr += offs_b * d_stride_b + offs_k * d_stride_k
    dQ_ptr += offs_b * dq_stride_b + offs_k * dq_stride_k
    dK2_ptr += offs_b * dk2_stride_b + offs_k * dk2_stride_k
    dV2_ptr += offs_b * dv2_stride_b + offs_k * dv2_stride_k

    softmax_scale = tl.cast(SM_SCALE, gemm_dtype)
    qkv_offs_h = tl.arange(0, HEAD_DIM)
    qkv_mask_h = qkv_offs_h < head_dim

    q_offs_s = q_start + tl.arange(0, BLOCK_SIZE_Q)
    kv2_offs_s = kv2_start + tl.arange(0, BLOCK_SIZE_KV2)
    q_offs = q_offs_s[:, None] * q_stride_s + qkv_offs_h[None, :] *
        q_stride_h
    kv2_offs = kv2_offs_s[:, None] * k2_stride_s + qkv_offs_h[None, :] *
        k2_stride_h
    m_offs = q_offs_s * m_stride_s
    d_offs = q_offs_s * d_stride_s
    dO_offs = q_offs_s[:, None] * dO_stride_s + qkv_offs_h[None, :] *
        dO_stride_h
    q_mask_s = q_offs_s < seq_len
    q_mask = q_mask_s[:, None] & qkv_mask_h[None, :]
    kv2_mask_s = 0 <= kv2_offs_s and kv2_offs_s < seq_len
    kv2_mask = kv2_mask_s[:, None] & qkv_mask_h[None, :]


    q_tile = tl.load(Q_ptr + q_offs, mask=q_mask).to(
        compute_dtype
    )  # [BLOCK_SIZE_Q, HEAD_DIM]
    k2_tile = tl.load(K2_ptr + kv2_offs, mask=kv2_mask).to(gemm_dtype) #
        [KV2, HEAD_DIM]
    v2_tile = tl.load(V2_ptr + kv2_offs, mask=kv2_mask).to(gemm_dtype) #
        [KV2, HEAD_DIM]
    m_tile = tl.load(M_ptr + m_offs, mask=q_mask_s).to(compute_dtype) # [
        BLOCK_SIZE_Q]
    d_tile = tl.load(D_ptr + d_offs, mask=q_mask_s).to(compute_dtype) # [
        BLOCK_SIZE_Q]
    dO_tile = tl.load(dO_ptr + dO_offs, mask=q_mask).to(
        gemm_dtype
    )  # [BLOCK_SIZE_Q, HEAD_DIM]

    # Apply KV2 norm.
    k2_tile += K2_BIAS
    v2_tile += V2_BIAS
    k2_tile = k2_tile.to(gemm_dtype)
    v2_tile = v2_tile.to(gemm_dtype)

    dq = tl.zeros((BLOCK_SIZE_Q, HEAD_DIM), tl.float32)
    dk2 = tl.zeros((BLOCK_SIZE_KV2, HEAD_DIM), tl.float32)
    dv2 = tl.zeros((BLOCK_SIZE_KV2, HEAD_DIM), tl.float32)

    kv1_start = tl.maximum(0, q_start - w1)
    kv1_end = tl.minimum(seq_len, q_end)
    for kv1_idx in tl.range(kv1_start, kv1_end, num_stages=num_stages):
        k1_offs = kv1_idx * k1_stride_s + qkv_offs_h * k1_stride_h
        v1_offs = kv1_idx * v1_stride_s + qkv_offs_h * v1_stride_h
        k1_tile = tl.load(K1_ptr + k1_offs, mask=qkv_mask_h).to(
            compute_dtype
        )  # [HEAD_DIM]

        v1_tile = tl.load(V1_ptr + v1_offs, mask=qkv_mask_h).to(
```

```
            compute_dtype
        )  # [HEAD_DIM]

        qk1_s = q_tile * (k1_tile[None, :] * softmax_scale) # [Q, D]
        qk1_s = qk1_s.to(gemm_dtype)
        # k2[KV, Q] @ qk1_s.T[Q, D] => [KV2, Q]
        qkkT = tl.dot(k2_tile, qk1_s.T, out_dtype=tl.float32) # [KV2, Q]

        qkT_mask = kv2_mask_s[:, None] & q_mask_s[None, :]
        kv1_local_mask = ((q_offs_s[None, :] - w1) < kv1_idx) & (
            kv1_idx <= q_offs_s[None, :]
        )  # [KV2, Q]
        kv2_local_mask = ((q_offs_s[None, :] - w2) < kv2_offs_s[:, None])
             & (
            kv2_offs_s[:, None] <= q_offs_s[None, :]
        )  # [KV2, Q]
        local_mask = (
            kv1_local_mask & kv2_local_mask
        )  # [BLOCK_SIZE_KV, BLOCK_SIZE_Q]
        qkT_mask &= kv1_local_mask & kv2_local_mask

        pT = tl.exp(qkkT - m_tile[None, :]) # [KV2, Q]
        pT = tl.where(qkT_mask, pT, 0.0)

        qkkT = tl.where(local_mask, qkkT, -1.0e38)

        dOv1 = dO_tile * v1_tile[None, :] # [Q, D]
        dOv1 = dOv1.to(gemm_dtype)
        # pT[KV2, Q] @ dOv1[Q, D] => [KV2, D]
        dv2 += tl.dot(pT.to(gemm_dtype), dOv1, out_dtype=tl.float32)

        # v2[KV2, D] @ dOv1.T[D, Q] => dpT[KV2, Q]
        dpT = tl.dot(v2_tile, dOv1.T, out_dtype=tl.float32)
        dsT = pT * (dpT - d_tile[None, :]) # [KV2, Q]
        dsT = tl.where(qkT_mask, dsT, 0.0)
        dsT = dsT.to(gemm_dtype) # [KV2, Q]

        # dsT[KV2, Q] @ qk1[Q, D] => dk2[KV2, D]
        dk2 += tl.dot(dsT, qk1_s, out_dtype=tl.float32)

        k1k2 = k1_tile[None, :] * k2_tile # [KV2, D]
        k1k2 = k1k2.to(gemm_dtype)

        dq += tl.dot(dsT.T, k1k2) # * softmax scale at the end.

    # End. update derivatives.
    if IS_SECOND_PASS:
        #load, add.
        prev_dk2 = tl.load(dK2_ptr + kv2_offs, kv2_mask)
        prev_dv2 = tl.load(dV2_ptr + kv2_offs, kv2_mask)
        dk2 += prev_dk2
        dv2 += prev_dv2

    dq *= softmax_scale
    tl.store(dK2_ptr + kv2_offs, dk2, kv2_mask)
    tl.store(dV2_ptr + kv2_offs, dv2, kv2_mask)
    tl.store(dQ_ptr + q_offs, dq, q_mask)
```

Listing 3: Backward pass for 2-simplicial attention optimized for small w_2 avoiding atomic adds.

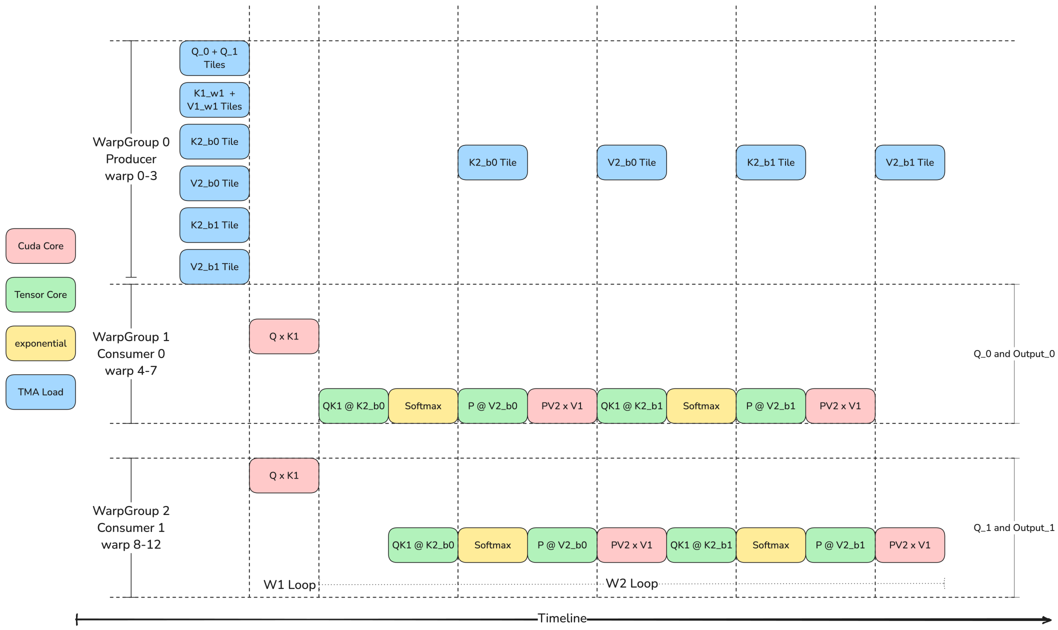


Figure 4: Scheduling Flow of TLX Kernel-3

E TLX (Triton Low-level Language Extensions): Forward Pass for 2-Simplicial Attention

Despite implementing all the kernel optimizations mentioned above, our Triton kernel implementation remained significantly below state-of-the-art performance. Our best forward attention kernel achieved only 336 Tensor Core TFLOPS with 34% Tensor Core utilization.

Analysis of the generated PTX code revealed that several important GPU optimization passes failed to work with the kernel, including software pipelining and warp specialization.

To rapidly integrate modern attention optimization techniques on Hopper, we rewrote the kernel using TLX (Triton Low-level Language Extensions), described in Figure 4. We developed three distinct versions:

- **Kernel-1**: Forward + Warp Specialization, described in Algorithm 3
- **Kernel-2**: Forward + Warp Specialization + Computation Pipelining
- **Kernel-3**: Forward + Warp Specialization + Pingpong Scheduling

The benchmark results show that the TLX kernel can achieve up to 588 TFLOPS peak performance with BFloat16 for the forward pass, with approximately 60% BFloat16 Tensor Core utilization, which is approximately 1.75× that of the Triton forward pass.

Algorithm 3 Fast 2-Simplicial Attention Forward Pass with Warp Specialization

Require: Tensors $Q_i \in \mathbb{R}^{B_r \times d}$ and $K_1, K_2, V_1, V_2 \in \mathbb{R}^{N \times d}$
Require: Output $O_i \in \mathbb{R}^{B_r \times d}$
Require: Sliding window size for K_1, V_1 is w_1 and for K_2, V_2 is w_2
Require: Number of circular K_2, V_2 SMEM buffers is num_buffers
procedure FAST2SIMPLICIALATTENTION(Q_i, K_1, K_2, V_1, V_2)
  Initialize $O_i \leftarrow \mathbf{0} \in \mathbb{R}^{B_r \times d}$, $\ell_i \leftarrow \mathbf{0} \in \mathbb{R}^{B_r}$, $m_i \leftarrow (-\infty) \in \mathbb{R}^{B_r}$
  For each CTA:
  if in producer warpgroup **then**
    Deallocate number of registers
    Load Q_i tile from GMEM to SMEM $\in \mathbb{R}^{B_r \times d}$
    Load $K_{1,i}$ tile and $V_{1,i}$ tile from GMEM to SMEM $\in \mathbb{R}^{w_1 \times d}$
    Set $\text{acc\_cnt} = 0$
    for $j \in \text{range}(w_1)$ **do**
      for $k \in (i - w_2 + 1, i]$ with step B_c **do**
        Wait for $\text{buffer\_id} = \text{acc\_cnt} \bmod \text{num\_buffers}$ to be released
        Issue the load of $K_{2,k}, V_{2,k}$ from HBM to SMEM
        Notify consumers of the load complete of $K_{2,k}$ and $V_{2,k}$
        $\text{acc\_cnt} = \text{acc\_cnt} + 1$
      end for
    end for
  else ▷ in consumer warpgroup
    Reallocate the number of registers
    Wait for Q_i tile to be loaded in SMEM
    Load Q_i from SMEM to RMEM
    Wait for $K_{1,i}, V_{1,i}$ tiles to be loaded in SMEM
    $\text{acc\_cnt} = 0$
    for $j \in (i - w_1 + 1, i]$ **do**
      Load $K_{1,j}, V_{1,j}$ from SMEM to RMEM $\in \mathbb{R}^{1 \times d}$
      Compute $QK_{1,ij} = Q_i \odot K_{1,j} \in \mathbb{R}^{B_r \times d}$
      for $k \in (i - w_2 + 1, i]$ with step B_c **do**
        Wait for $K_{2,k}$ to be loaded in SMEM $\in \mathbb{R}^{B_c \times d}$
        Compute $S_{ijk} = QK_{1,ij} K_{2,k}^T \in \mathbb{R}^{B_r \times B_c}$ ▷ RS-GEMM
        Store $m_i^{\text{old}} = m_i$
        Compute $m_i = \max(m_i^{\text{old}}, \text{rowmax}(S_{ijk}))$
        Compute $P_{ijk} = \exp(S_{ijk} - m_i) \in \mathbb{R}^{B_r \times B_c}$
        Compute $\ell_i^{\text{new}} = \exp(m_i^{\text{old}} - m_i)\ell_i + \text{rowsum}(P_{ijk})$
        Wait for $V_{2,k}$ to be loaded in SMEM $\in \mathbb{R}^{B_c \times d}$
        Compute $PV_{2,ijk} = P_{ijk} V_{2,k} \in \mathbb{R}^{B_r \times d}$ ▷ RS-GEMM
        Compute $PV_{12,ijk} = PV_{2,ijk} \odot V_{1,j} \in \mathbb{R}^{B_r \times d}$
        Update $O_i = \exp(m_i^{\text{old}} - m_i) O_i + P_{ijk} V_{12,jk}$
        $\text{buffer\_id} = \text{acc\_cnt} \bmod \text{num\_buffers}$
        Release buffer_id for $K_{2,k}$ and $V_{2,k}$
        Update $\ell_i = \ell_i^{\text{new}}$, $\text{acc\_cnt} = \text{acc\_cnt} + 1$
      end for
    end for
    Compute $O_i = O_i / \ell_i$, $L_i = m_i + \log(\ell_i)$
    Write O_i and L_i to HBM
  end if
  return O_i
end procedure