FlashKAT: Understanding and Addressing Performance Bottlenecks in the Kolmogorov-Arnold Transformer

Matthew Raffel, Lizhong Chen

School of Electrical Engineering and Computer Science
Oregon State University, USA
{raffelm, chenliz}@oregonstate.edu

Abstract

The Kolmogorov-Arnold Network (KAN) has been gaining popularity as an alternative to the multilayer perceptron (MLP) due to its greater expressiveness and interpretability. Even so, KAN suffers from training instability and being orders of magnitude slower due to its increased computational cost, limiting its applicability to large-scale tasks. Recently, the Kolmogorov-Arnold Transformer (KAT) has been proposed, achieving FLOPs comparable to traditional Transformer models with MLPs by leveraging Group-Rational KAN (GR-KAN). Unfortunately, despite the comparable FLOPs, our testing shows that KAT remains 123x slower during training, indicating that there are other performance bottlenecks beyond FLOPs. In this paper, we conduct a series of experiments to understand the root cause of the slowdown in KAT. We uncover that the slowdown can be isolated to memory stalls, linked more specifically to inefficient gradient accumulations in the backward pass of GR-KAN. To address this memory bottleneck, we propose FlashKAT, which minimizes accesses to slow memory and the usage of atomic adds through a restructured kernel. Evaluations show that FlashKAT achieves up to an 86.5x training speedup over state-of-the-art KAT while reducing rounding errors in gradient computation.

Code — https://github.com/OSU-STARLAB/FlashKAT
Extended version — https://arxiv.org/abs/2505.13813

1 Introduction

The Kolmogorov-Arnold Network (KAN) has emerged as a popular alternative to the multilayer perceptron (MLP) (Liu et al. 2024). Rather than learning a fixed weight per edge, KAN instead learns a flexible nonlinearity. In doing so, KAN achieves greater expressivity and interpretability than MLP, which has led to impressive capabilities in scientific and equation modeling tasks (Li et al. 2025; Coffman and Chen 2025; Liu et al. 2024; Koenig, Kim, and Deng 2024; Wang et al. 2024). However, even with the superior performance of KAN for these tasks, it has fallen short on larger-scale tasks in NLP and computer vision due to substantially increased parameters and computational cost, lack of GPU hardware optimization (e.g., for basis functions), and training instability (Yu, Yu, and Wang 2024; Le et al. 2024; Chen, Gundavarapu, and DI 2024). For example, while computing an edge for an MLP only requires 2 FLOPS (1 for multiply and 1 for add), computing an edge for a KAN may require up to 204 FLOPs, due to the nonlinearity (Yang and Wang 2024). This shortcoming necessitates significant research to reduce the orders of magnitude performance gap between KANs and MLPs.

Recently, the Kolmogorov-Arnold Transformer (KAT) has overcome some of the obstacles that prevent KAN from being deployed to large-scale tasks (Yang and Wang 2024). It achieves this by replacing each MLP in the Transformer with a Group-Rational KAN (GR-KAN), which augments the input to each linear layer with the safe Pade Activation Unit (PAU) (Molina, Schramowski, and Kersting 2019) in a grouped fashion (more details in Section 2). In doing so, each GR-KAN learns a weighted summation of learned nonlinear functions. Unlike KAN, GR-KAN is more computationally efficient, reducing FLOPs to only slightly more than MLPs, and is designed to better map to GPUs. By modifying the Transformer with GR-KAN, the resulting KAT achieves impressive results on computer vision tasks.

However, even with the comparable number of FLOPs (and similar *expected* speed) offered by KAT, in our characterization, KAT is significantly slower than its MLP counterpart. For instance, our experiments reveal that a KAT (using GR-KAN) is 123x slower during training than an identically sized Vision Transformer (ViT, using MLP) (Dosovitskiy et al. 2020) on ImageNet-1K (Russakovsky et al. 2015). This realization motivated an in-depth investigation to understand the KAT's performance bottleneck. Unlike previous approaches focused primarily on FLOPs, we re-examine the performance issue from a memory-centric perspective, a novel angle not previously explored (Yang and Wang 2024), which uncovers that the slowdown is primarily due to a memory bottleneck rather than excessive computation. Specifically, we identify the inefficient gradient accumulation method for the PAU coefficients, involving atomic adds, as the key contributor (Molina, Schramowski, and Kersting 2019). This insight not only challenges assumptions that computation is the primary constraint in KAT but also offers a new direction for optimizing memory access patterns in similar architectures.

To address the memory bottleneck, we propose FlashKAT, a *substantially* faster KAT that solves GR-KAN's (and by extension PAU's) shortcomings by not only avoiding accu-

mulating gradients with atomic adds but also the unnecessary accesses to high-bandwidth memory (HBM). The method restructures data accesses and enables more effective use of shared memory for gradient accumulations. By implementing our method as a GPU kernel, FlashKAT achieves a training speedup of up to 86.5x compared with state-of-the-art KAT. Furthermore, we demonstrate that our new kernel reduces rounding errors in the PAU coefficient gradient computations by nearly an order of magnitude, thereby improving training stability.

As our method directly optimizes PAU, a learnable rational activation, the techniques and analysis also apply to alternative learnable rational activations, a complementary area of research to KAT (Molina, Schramowski, and Kersting 2019; Delfosse et al. 2020, 2021; Biswas, Banerjee, and Pandey 2021). Despite the clear need for efficient implementations, researchers in both the KAT and rational activation communities had not identified a way to accelerate these activations. FlashKAT provides this essential optimization, representing a nontrivial breakthrough that not only advances the state of the art but also inspires future research in the field.

The main contributions of this paper are:

1. Conducting a detailed investigation to understand the computational inefficiencies of KAT, revealing that the primary performance bottleneck arises not from FLOPs, but from suboptimal memory accesses caused by inefficient gradient accumulation.
2. Proposing FlashKAT, which addresses the identified memory bottleneck by leveraging a substantially more efficient gradient accumulation method to speed up training.
3. Performing an extensive evaluation of FlashKAT, demonstrating significantly reduced training times and rounding errors from the proposed method.

2 Background and Related Works

Kolmogorov-Arnold Network

Recently, KAN has been popularized as an alternative to MLP (Liu et al. 2024). It is based on the Kolmogorov-Arnold Theorem (Kolmogorov 1961), which asserts that a multivariate continuous function on a bounded domain $f : [0, 1]^n \to \mathbb{R}$ can be rewritten as a finite summation of univariate continuous functions, $\phi_{q,p} : [0, 1] \to \mathbb{R}$ and $\phi_q : \mathbb{R} \to \mathbb{R}$ as expressed in Equation 1:

$$f(x_1, ..., x_n) = \sum_{q=1}^{2n+1} \phi_q(\sum_{p=1}^{n} \phi_{q,p}(x_p)). \tag{1}$$

Although Equation 1 presents a theoretical form of the Kolmogorov-Arnold Theorem, (Liu et al. 2024) demonstrated its effectiveness in a more generalized form by scaling the width and depth arbitrarily for a defined task. Suppose n_l is the number of input nodes for layer l, where each input is represented by $x_{l,i}$. Each edge connection to layer $l + 1$ has a learned activation $\phi_{l,j,i}(\cdot)$, where $i \in [1, n_l]$ and $j \in [1, n_{l+1}]$. Then, we can represent each output node as

$$x_{l+1,j} = \sum_{i=1}^{n_l} \phi_{l,j,i}(x_{l,i}). \tag{2}$$

In a vectorized format, we can represent Equation 2 with

$$\mathbf{x}_{l+1} = \Phi \circ \mathbf{x}_l = \begin{bmatrix} \phi_{l,1,1}(\cdot) & \cdots & \phi_{l,1,n_l}(\cdot) \\ \vdots & \ddots & \vdots \\ \phi_{l,n_{l+1},1}(\cdot) & \cdots & \phi_{l,n_{l+1},n_l}(\cdot) \end{bmatrix} \mathbf{x}_l. \tag{3}$$

Thus, an L layer KAN can be represented as

$$\text{KAN}(\mathbf{x}) = (\Phi_L \circ \Phi_{L-1} \circ ... \circ \Phi_1)\mathbf{x}. \tag{4}$$

The most widely adopted method for representing $\phi_{l,j,i}(\cdot)$ has been with B-splines (Liu et al. 2024), and although B-spline-based KANs demonstrate promising results, they still have a host of shortcomings (Yang and Wang 2024; Yu, Yu, and Wang 2024). These shortcomings consist of (1) increased parameter and computational cost compared to MLP (e.g., increased FLOPs as summarized in Table 1), (2) B-splines not being GPU optimizable (e.g., requiring a recursive algorithm), and (3) training instability in large networks (Yang and Wang 2024). Such shortcomings limit KAN's compatibility with the Transformer (Vaswani et al. 2017), restricting its applicability to large-scale tasks like computer vision.

Kolmogorov-Arnold Transformer

Unlike its predecessors, littered with shortcomings, KAT succeeds in merging KAN into a Transformer. It does so by developing a new KAN variation called GR-KAN, which (1) shares coefficients across edge groups, reducing FLOPs, (2) leverages safe PAU activation functions, eliminating recursion, and (3) applies a variance-preserving initialization, increasing training stability (Yang and Wang 2024). In KAT, each MLP is replaced with a GR-KAN, causing each MLP weight to be augmented with a nonlinear learned activation function (i.e., a nonlinear edge in KAN).

First, rather than learning $d_{in} \cdot d_{out}$ activations, $\phi_{l,j,i}(\cdot)$, like KAN, GR-KAN groups inputs and only learns $\lfloor d_{in}/d_g \rfloor$ activations, $F(\cdot)$, where d_g is the number of activations that share the same set of learned coefficients in a group. Each output $x_{l+1,j}$ is then a weighted summation of $F_{\lfloor i/d_g \rfloor}(x_{l,i})$ using learnable weights, $\mathbf{W} \in \mathbb{R}^{d_{out} \times d_{in}}$. GR-KAN is expressed in Equation 5:

$$\begin{aligned} \text{GR-KAN}(\mathbf{x}) &= \mathbf{W}\mathbf{F}(\mathbf{x}) \\ &= \begin{bmatrix} w_{11} & \cdots & w_{1,d_{in}} \\ \vdots & \ddots & \vdots \\ w_{d_{out},1} & \cdots & w_{d_{out},d_{in}} \end{bmatrix} \begin{bmatrix} F_{\lfloor 1/d_g \rfloor}(x_1) \\ \vdots \\ F_{\lfloor d_{in}/d_g \rfloor}(x_{d_{in}}) \end{bmatrix}. \end{aligned} \tag{5}$$

Second, each GR-KAN activation is a safe PAU (Molina, Schramowski, and Kersting 2019), which we will refer to as a *group-wise rational function*, composed of learnable coefficients a_i and b_j as follows:

$$F(x) = \frac{P(x)}{Q(x)} = \frac{a_0 + a_1 x + a_2 x^2 + ... + a_m x^m}{1 + |b_1 x + b_2 x^2 + ... + b_n x^n|}. \tag{6}$$

Third, a variance-preserving initialization is applied for stable learning (Yang and Wang 2024). The procedure for applying variance preserving weights requires (1) initializing

Name	Number of parameters	FLOPs
MLP (ViT)	$d_{\text{in}} \times d_{\text{out}}$	$\text{FuncFLOPs} \times d_{\text{out}} + 2 \times d_{\text{in}} \times d_{\text{out}}$
KAN	$d_{\text{in}} \times d_{\text{out}} \times (G + K + 3)$	$\text{FuncFLOPs} \times d_{\text{in}} + d_{\text{in}} \times d_{\text{out}} \times \left[9K \times (G + 1.5K) + 2G - 2.5K + 3\right]$
GR-KAN (KAT)	$d_{\text{in}} \times d_{\text{out}} + (m + n \times g + 1)$	$(2m + 2n + 3) \times d_{\text{in}} + 2 \times d_{\text{in}} \times d_{\text{out}}$

Table 1: Comparison of parameter counts and FLOPs for different layers. "FuncFLOPs" denotes the activation FLOPs. In KAN, K is the spline order and G the number of intervals; in GR-KAN, m, n are the polynomial degrees and g the number of groups.

the coefficients of $F(\cdot)$ to mimic a known activation function (ie. Swish) and (2) initializing the weights in $\mathbf{W}$ according to $\mathcal{N}(0, \frac{\alpha}{d_{in}})$ whereby $\alpha = \frac{\mathbb{E}[\mathbf{F}(\mathbf{x})^2]}{Var[\mathbf{x}]}$ assuming $\mathbf{x} \sim \mathcal{N}(0, 1)$.

Although KAT is effective at vision tasks and is more computationally efficient than a version of ViT with KAN (a result of overcoming shortcomings (1), (2), and (3)), they still far short in terms of training speed compared to an identical ViT using MLP, a shortcoming which has slowed further research. We demonstrate such results in the next Section.

3 Beyond FLOPs: Understanding Performance Bottlenecks of KAT

Insight 1: KAT Is Significantly Slower Than ViT During Training

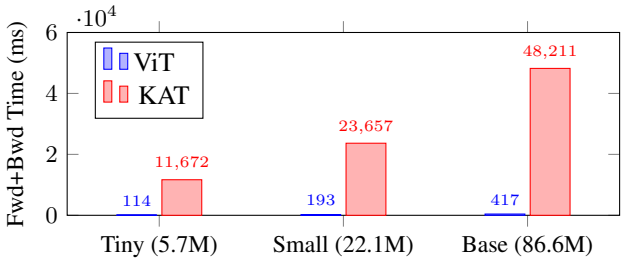

Figure 1: Comparison of training time (Fwd+Bwd) for ViT and KAT.

Although KAT presents a promising alternative to ViT, it still lags behind ViT in training time. We illustrate this by comparing the combined forward and backward times for KAT-T (5.7M parameters), KAT-S (22.1M parameters), and KAT-B (86.6M parameters) against identically sized ViT-T, ViT-S, and ViT-B models, using the ImageNet-1k dataset for training (Russakovsky et al. 2015). For the experiment, each KAT group-wise rational function contains 8 groups comprising of 6 numerator and 4 denominator coefficients, and all measurements are taken using an H200 GPU. The results, depicted in Figure 1, show the average time of 100 forward and backward passes, following 5 warmup passes. The data reveals that, for these forward/backward passes, KAT-T is 102x slower than ViT-T, KAT-S is 123x slower than ViT-S, and KAT-B is 116x slower than ViT-B.

Insight 2: FLOPs Is Not the Performance Bottleneck

It is natural to assume that such a drop-off in training speed between ViT and KAT results from the difference in FLOPs, as KAT has more FLOPs than ViT. However, by comparing the estimated FLOPs for GR-KAN and MLP (Table 1), the only component in ViT and KAT where they differ, we can dismiss this assumption.

As can be seen, between the FLOPs expressions of ViT and KAT, the main difference resides in the MLP activation requiring $\text{FuncFLOPs} \times d_{out}$ FLOPs and the GR-KAN learnable activation requiring $(2m + 2n + 3) \times d_{in}$ FLOPs, both of which are negligible compared with the $2 \times d_{in} \times d_{out}$ term. This suggests that the slight increase in FLOPs in KAT may not be the cause of the slowdown. To further verify this, we conduct an experiment that artificially increases the FLOPs inside the group-wise rational function of KAT by placing an additional loop around the floating-point computations. This would examine whether FLOPs bottleneck the performance. We measure performance metrics using Nvidia Nsight Compute[1] on an RTX 4060 Ti for the forward and backward pass. The dimension configuration is: input tensor, $\mathbf{X} \in \mathbb{R}^{1024 \times 197 \times 768}$, numerator coefficients, $\mathbf{A} \in \mathbb{R}^{8 \times 6}$, denominator coefficients, $\mathbf{B} \in \mathbb{R}^{8 \times 4}$, and the upstream gradient tensor, $\mathbf{dO} \in \mathbb{R}^{1024 \times 197 \times 768}$. We report these results in Table 2.

From Table 2, we can immediately see that, even if the FLOPs scale to 8x of the original amount, the overall required "Cycles" or "Time" to complete the computation does not scale in either the forward or backward pass. Such results clearly demonstrate that the forward and backward pass kernels in KAT are not compute-bound, and we must look beyond FLOPs to identify performance bottlenecks.

Insight 3: The Backward Pass Dominates Execution Time

Results in Table 2 also demonstrate that the group-wise rational function dedicates a far greater amount of compute time to the backward pass than the forward pass. For instance, the forward pass takes 4.96 ms, whereas the backward pass requires 1.03 s, which is 207.7x as long. Therefore, to reduce KAT's training time, performance issues in the backward pass must be identified and addressed. The backward pass can be broken down into three components: (1) loading values to compute the gradients, (2) computing the gradients, and (3) storing the gradients. Since we know the backward pass is not compute-bound from Insight 2, this means that (1) and (3) are to blame for the slowdown, both of which are memory issues.

[1]The documentation for Nvidia Nsight Compute can be found at https://docs.nvidia.com/nsight-compute/

Forward Pass							
Loops	FLOPs	Cycles	Time	SM Thp. (%)	L1 Thp. (%)	L2 Thp. (%)	HBM Thp. (%)
1	2.9T	11.3M	4.89 ms	29.14	28.25	18.11	89.40
2	5.9T	11.3M	4.91 ms	30.34	28.46	18.08	89.25
4	11.8T	11.3M	4.89 ms	41.02	29.08	19.78	89.37
8	23.6T	11.3M	4.91 ms	62.20	28.00	18.40	89.34
Backward Pass							
Loops	FLOPs	Cycles	Time	SM Thp. (%)	L1 Thp. (%)	L2 Thp. (%)	HBM Thp. (%)
1	11.2T	2.4T	1.03 s	1.97	4.38	5.24	1.01
2	22.3T	2.4T	1.03 s	1.97	4.38	5.22	1.01
4	44.6T	2.4T	1.03 s	1.97	4.38	5.22	1.01
8	89.2T	2.4T	1.03 s	1.97	4.38	5.22	1.01

Table 2: Performance breakdown for forward and backward passes of the group-wise rational function. "Loops" is the total instruction loops; "Cycles" is the cycle count; "Time" is wall-clock time; "FLOPs" is the floating point operation count; "SM Thp.", "L1 Thp.", "L2 Thp.", and "HBM Thp." report the percentage of the device's peak arithmetic and memory throughput.

Insight 4: Memory (and Memory Stall in Particular) Is the Culprit

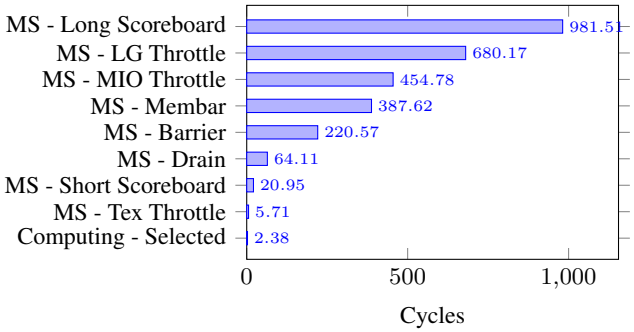

Figure 2: The warp-state statistics for the KAT group-wise rational function backward pass. "Computing - Selected" is where the warp does useful computation; all the others are memory stall (MS) states.

Focusing on the memory side for backward passes, we note that Table 2 reports very low L1, L2, and HBM throughput. At first glance, this is counterintuitive because a memory-bound process would typically imply a saturated memory bandwidth. However, further reasoning about the underlying hardware can provide insights. A GPU consists of multiple streaming multiprocessors (SMs). Typically, 32 threads are grouped in a *warp* and executed synchronously. Multiple warps can be scheduled for execution on an SM in a context-switching fashion. When one warp is waiting for a memory transfer, the SM can switch to doing computation for a different "ready" warp (i.e., not waiting for memory), thereby hiding memory latency. However, suppose there are not enough ready warps. Then, the warp scheduler cannot entirely hide long latencies or high memory contention, like with atomic operations that create sequential memory accesses. If an SM does not make adequate use of the memory hierarchy consisting of registers, shared memory, and global memory (a mapped memory composed of L1 cache, L2 cache, and HBM), the latency for each transfer becomes even more severe. For reference, on an A100, shared memory, L1, L2, and HBM have latencies of 29.0, 37.9, 261.5, and 466.3 clock cycles, respectively (Luo et al. 2024). Assuming that the group-wise rational function evaluation fails to hide latency, it makes sense that we would witness low memory utilization yet still experience a memory bottleneck. To validate this assumption, we examine the warp state statistics for the group-wise rational function backward pass using Nvidia Nsight Compute with the same configuration as before. These statistics report the average number of cycles each warp remains in a given state for each instruction the kernel executes. We report the warp state statistics in Figure 2.

Figure 2 demonstrates that each warp spends the majority of its time waiting for various memory stalls to conclude. Each reported stall corresponds to a memory transfer, excluding the "Selected" state, in which the warp is selected to issue an instruction (i.e., compute). For instance, the "Stall Long Scoreboard", a stall on a memory transfer from global memory, accounts for 412x the time spent in the "Selected" state. As such, it becomes clear that the group-wise rational function slowdown is primarily due to memory transfer stalls. This calls for a new method that is aware of the memory hierarchy and avoids memory operations that create contention, limiting the warp scheduler's latency-hiding capabilities.

4 FlashKAT

In the previous section, we identified that KAT has a memory bottleneck in the backward pass of the group-wise rational function, a finding that prior approaches had not recognized (Yang and Wang 2024; Molina, Schramowski, and Kersting 2019; Delfosse et al. 2020, 2021). In this section, we first outline the computations required for this operation, then provide an analysis of the standard group-wise rational function used in KAT to pin down the source of the bottleneck, and finally present FlashKAT, our proposed Kolmogorov-Arnold Transformer that builds on an optimized group-wise rational function that efficiently leverages gradient accumulation and memory hierarchy to minimize global memory accesses.

Gradient Computations

Since group-wise rational function, expressed in Equation 6 as $F(\cdot)$, is an element-wise operation composed of learnable parameters a_i and b_j for the numerator and the denominator, the backward pass requires us to compute the gradients $\frac{\partial F}{\partial a_i}$, $\frac{\partial F}{\partial b_j}$, and $\frac{\partial F}{\partial x}$. Suppose that $A(x) = b_1 x + ..., + b_n x^n$, $\frac{\partial P(x)}{\partial x} = a_1 + 2a_2 x + ... + m a_m x^{m-1}$, and $\frac{\partial Q(x)}{\partial x} = \frac{\partial (1+|A(x)|)}{\partial A(x)} \cdot \frac{\partial A(x)}{\partial x} = \frac{A(x)}{|A(x)|}(b_1 + 2b_2 x + ... + n b_n x^{n-1})$ then the gradients for each are

$$\frac{\partial F}{\partial a_i} = \frac{x^i}{Q(x)}, \tag{7}$$

$$\frac{\partial F}{\partial b_j} = -x^j \frac{A(x)}{|A(x)|} \cdot \frac{P(x)}{Q(x)^2}, \tag{8}$$

$$\frac{\partial F}{\partial x} = \frac{\partial P(x)}{\partial x} \cdot \frac{1}{Q(x)} - \frac{\partial Q(x)}{\partial x} \cdot \frac{P(x)}{Q^2(x)}. \tag{9}$$

Equations 7, 8, and 9 provides the local element-wise gradients; however, since each batch of size B, each element in the sequence of size N, and each group of size d_g share an identical set of coefficients, the contributions of each must be accumulated on $\frac{\partial F}{\partial a_i}$ and $\frac{\partial F}{\partial b_j}$. Let us denote $\frac{\partial F}{\partial a_{g,i}}$ and $\frac{\partial F}{\partial b_{g,j}}$ as the gradient for coefficient i and j of group g and $\frac{\partial \mathcal{L}}{\partial F}$ the upstream gradient. Then we can express this accumulation in Equations 10 and 11:

$$\frac{\partial \mathcal{L}}{\partial a_{g,i}} = \sum_{i'=1}^{B} \sum_{j'=1}^{N} \sum_{k'=1}^{d_g} \frac{\partial \mathcal{L}}{\partial F^{i',j'}_{g \cdot d_g + k'}} \cdot \frac{\partial F^{i',j'}_{g \cdot d_g + k'}}{\partial a_{g,i}}, \tag{10}$$

$$\frac{\partial \mathcal{L}}{\partial b_{g,j}} = \sum_{i'=1}^{B} \sum_{j'=1}^{N} \sum_{k'=1}^{d_g} \frac{\partial \mathcal{L}}{\partial F^{i',j'}_{g \cdot d_g + k'}} \cdot \frac{\partial F^{i',j'}_{g \cdot d_g + k'}}{\partial b_{g,j}}. \tag{11}$$

There are multiple different methods for accumulating these gradients in a kernel, and identifying the optimal accumulation procedure is crucial for a fast training algorithm. We will now demonstrate where the standard group-wise rational function method falls short in this regard.

Analysis of KAT Group-Wise Rational Function

Provided an input $\mathbf{X} \in \mathbb{R}^{B \times N \times d}$, an upstream gradient $\mathbf{dO} \in \mathbb{R}^{B \times N \times d}$ and coefficients $\mathbf{A} \in \mathbb{R}^{n_g \times m+1}$ and $\mathbf{B} \in \mathbb{R}^{n_g \times n}$, we want to compute the gradients $\mathbf{dA} \in \mathbb{R}^{n_g \times m+1}$ ($\frac{\partial \mathcal{L}}{\partial \mathbf{A}}$), $\mathbf{dB} \in \mathbb{R}^{n_g \times n}$ ($\frac{\partial \mathcal{L}}{\partial \mathbf{B}}$), and $\mathbf{dX} \in \mathbb{R}^{B \times N \times d}$ ($\frac{\partial \mathcal{L}}{\partial \mathbf{X}}$). In Algorithm 1, we provide the method for computing each of these gradients in KAT. The algorithm is structured sequentially, and all on-chip computation is performed within subroutine calls, with the upstream gradients applied to Equations 7, 8, and 9 for readability.

At a high level, Algorithm 1 iterates over the T blocks in the grid, where each block has S_{block} threads that compute the associated gradients for $\mathbf{dA}$, $\mathbf{dB}$, $\mathbf{dX}$. In the case of $\mathbf{dA}$ and $\mathbf{dB}$, for each of these computations, the thread performs an atomic add, which requires reading from global memory, accumulating the value, and then storing the result back to global memory as an indivisible unit of work. However, since

Algorithm 1: The KAT group-wise rational function backward pass.

Require: $\mathbf{X}, \mathbf{dO} \in \mathbb{R}^{B \times N \times d}$, $\mathbf{A} \in \mathbb{R}^{n_g \times m+1}$, $\mathbf{B} \in \mathbb{R}^{n_g \times n}$ in global memory.
Set block size to S_{block}.
Divide $\mathbf{X}$ and $\mathbf{dO}$ into $T = \lceil (B \cdot N \cdot d / S_{\text{block}} \rceil$ blocks $\mathbf{X}_1, ..., \mathbf{X}_T$ and $\mathbf{dO}_1, ..., \mathbf{dO}_T$ each of size S_{block}.
Initialize $\mathbf{dA} \leftarrow 0 \in \mathbb{R}^{n_g \times m+1}$, $\mathbf{dB} \leftarrow 0 \in \mathbb{R}^{n_g \times n}$, and $\mathbf{dX} \leftarrow 0 \in \mathbb{R}^{B \times N \times d}$.
for $1 \le i \le T$ **do**
  Load $\mathbf{X}_i$ and $\mathbf{dO}_i$ from global memory
  for $1 \le j \le S_{\text{block}}$ **do**
    $k \leftarrow \lfloor (((i-1) \cdot S_{\text{block}} + j) \mod d) / d_g \rfloor$
    Load $\mathbf{A}_{k,:}$ and $\mathbf{B}_{k,:}$ from global memory
    $\mathbf{dX}_{i,j} \leftarrow$ **Comp_dX**$(\mathbf{A}_{k,:}, \mathbf{B}_{k,:}, \mathbf{X}_{i,j}, \mathbf{dO}_{i,j})$
    $\mathbf{dA}'_{k,:} \leftarrow$ **Comp_dA**$(\mathbf{A}_{k,:}, \mathbf{B}_{k,:}, \mathbf{X}_{i,j}, \mathbf{dO}_{i,j})$
    $\mathbf{dB}'_{k,:} \leftarrow$ **Comp_dB**$(\mathbf{A}_{k,:}, \mathbf{B}_{k,:}, \mathbf{X}_{i,j}, \mathbf{dO}_{i,j})$
    Load $\mathbf{dA}_{k,:}$ from global memory, $\mathbf{dA}_{k,:} \leftarrow \mathbf{dA}_{k,:} + \mathbf{dA}'_{k,:}$, write back $\mathbf{dA}_{k,:}$ to global memory ▷ atomic add
    Load $\mathbf{dB}_{k,:}$ from global memory, $\mathbf{dB}_{k,:} \leftarrow \mathbf{dB}_{k,:} + \mathbf{dB}'_{k,:}$, write back $\mathbf{dB}_{k,:}$ to global memory ▷ atomic add
  end for
  Write back $\mathbf{dX}_i$ to global memory
end for
return $\mathbf{dA}, \mathbf{dB}, \mathbf{dX}$

each of these threads must perform an atomic add for each of the $m + n + 1$ coefficients, the atomic adds must occur sequentially. As such, for each of these atomic adds, a stall occurs. Since multiple threads within the same block and across different blocks may write to the same location, there will be severe resource contention. Such resource contention inevitably leads to high latency and stalls without utilizing the available bandwidth, as we demonstrated in Section 3.

Let us analyze the total number of global memory accesses incurred by Algorithm 1. We can see that each $\mathbf{X}_i$ and $\mathbf{dO}_i$ requires S_{block} reads from global memory. Then writing $\mathbf{dX}_i$ back to global memory requires S_{block} writes to global memory. Since these 3 memory actions occur T times they require $3TS_{\text{block}} = 3\frac{B \cdot N \cdot d}{S_{\text{block}}} \cdot S_{\text{block}} = 3 \cdot B \cdot N \cdot d$ global memory accesses. Regarding the coefficients, each element in the output dX requires loading $m+n+1$ coefficients from global memory. It also requires the aforementioned atomic add, which includes an additional $m + n + 1$ reads and writes to global memory. Thus, since there are T blocks of S_{block} threads, there are $TS_{\text{block}} \cdot 3(m + n + 1) = 3(m + n + 1) \cdot B \cdot N \cdot d$ accesses. As a result, there are $3(m+n+2) \cdot B \cdot N \cdot d$ global memory accesses in total. To reiterate, memory accesses for atomic add operations for coefficient accumulations are much worse due to additional contention from other threads.

FlashKAT Group-Wise Rational Function

We now present an optimized method for computing the backward pass of the group-wise rational function in Algorithm 2. To reduce global memory accesses, we restructure the grid from 1D to 2D. Such a restructuring mirrors the format of Equation 10, facilitating a more straightforward mapping of it to the algorithm. The first dimension of our reconstructed grid is allocated for resolving the B and N summations, whereas

Algorithm 2: The FlashKAT group-wise rational function backward pass.

Require: $\mathbf{X}, \mathbf{dO} \in \mathbb{R}^{B\times N\times d}$, $\mathbf{A} \in \mathbb{R}^{n_g\times m+1}$, $\mathbf{B} \in \mathbb{R}^{n_g\times n}$ in global memory.
Set block sizes to S_{block} and $d_g = d/n_g$.
Divide $\mathbf{X}$ and $\mathbf{dO}$ into $\lceil (B\cdot N\cdot d/(S_{\text{block}}\cdot d_g)\rceil$ blocks $\mathbf{X}_{i,j} \in \mathbb{R}^{S_{\text{block}}\times d_g}$ and $\mathbf{dO}_{i,j} \in \mathbb{R}^{S_{\text{block}}\times d_g}$ of a $T \times n_g$ grid where $T = \lceil (B\cdot N/S_{\text{block}}\rceil$.
Divide $\mathbf{A}$ and $\mathbf{B}$ into n_g blocks $\mathbf{A}_j \in \mathbb{R}^{m+1}$ and $\mathbf{B}_j \in \mathbb{R}^n$.
Initialize $\mathbf{dA} \leftarrow 0 \in \mathbb{R}^{n_g\times m+1}$, $\mathbf{dB} \leftarrow 0 \in \mathbb{R}^{n_g\times n}$, and $\mathbf{dX} \leftarrow 0 \in \mathbb{R}^{B\times N\times d}$.
for $1 \le i \le T$ **do**
for $1 \le j \le n_g$ **do**
  Load $\mathbf{A}_j$, $\mathbf{B}_j$, $\mathbf{X}_{i,j}$, and $\mathbf{dO}_{i,j}$ from global memory
  Initialize $\mathbf{dA}'_j \leftarrow 0$ and $\mathbf{dB}'_j \leftarrow 0$
  for $1 \le k \le S_{\text{block}}$ **do**
  for $1 \le l \le d_g$ **do**
    $\mathbf{dX}_{i,j,k,l} \leftarrow$ **Comp_dX**$(\mathbf{A}_j, \mathbf{B}_j, \mathbf{X}_{i,j,k,l}, \mathbf{dO}_{i,j,k,l})$
    $\mathbf{dA}'_j \leftarrow \mathbf{dA}'_j +$ **Comp_dA**$(\mathbf{A}_j, \mathbf{B}_j, \mathbf{X}_{i,j,k,l}, \mathbf{dO}_{i,j,k,l})$
    $\mathbf{dB}'_j \leftarrow \mathbf{dB}'_j +$ **Comp_dB**$(\mathbf{A}_j, \mathbf{B}_j, \mathbf{X}_{i,j,k,l}, \mathbf{dO}_{i,j,k,l})$
  end for
  end for
  Load $\mathbf{dA}_j$ from global memory, $\mathbf{dA}_j \leftarrow \mathbf{dA}_j + \mathbf{dA}'_j$, write back $\mathbf{dA}_j$ to global memory ▷ atomic add
  Load $\mathbf{dB}_j$ from global memory, $\mathbf{dB}_j \leftarrow \mathbf{dB}_j + \mathbf{dB}'_j$, write back $\mathbf{dB}_j$ to global memory ▷ atomic add
  Write back $\mathbf{dX}_{i,j}$ to global memory
end for
end for
return $\mathbf{dA}, \mathbf{dB}, \mathbf{dX}$

the second dimension is responsible for the d_g summation. The size of this first dimension in the grid is $B\cdot N/S_{\text{block}}$, whereas the size of the second dimension is n_g (the total number of groups in GR-KAN).

Algorithm 2 does not change the memory accesses for reads and writes for $\mathbf{dX}, \mathbf{X}$ and $\mathbf{dO}$; however, it heavily influences them for $\mathbf{A}$, $\mathbf{B}$, $\mathbf{dA}$, and $\mathbf{dB}$. At a high level, we can see that Algorithm 2 iterates over the grid of $T\cdot n_g$ blocks. For each of these blocks, it loads a single group of coefficients for $\mathbf{A}_j$ and $\mathbf{B}_j$, which it then reuses for all computations in the block. As a result, rather than each thread performing an atomic add for each value in $\mathbf{dA}_j$ and $\mathbf{dB}_j$, the algorithm can instead accumulate all the contributions for $\mathbf{dA}_j$ and $\mathbf{dB}_j$ in the block before doing an atomic add. As a result, Algorithm 2 reduces the total amount of memory contention and global memory accesses.

We now analyze the total global memory accesses for Algorithm 2. Each block first loads one set of coefficients from the global memory associated with the group to which the block is assigned. Since there is only 1 set of coefficients per block, this operation requires only $m+n+1$ accesses. Then the algorithm loads the associated $\mathbf{X}_{i,j}$ and $\mathbf{dO}_{i,j}$ from global memory, each of size $S_{\text{block}}\cdot d_g$. After computing $\mathbf{dX}_{i,j}$, its $S_{\text{block}}\cdot d_g$ values are written back to global memory. Then, an atomic add loads $m+n+1$ values from $\mathbf{dA}_j$ and $\mathbf{dB_j}$, accumulates them with $\mathbf{dA}'_j$ and $\mathbf{dB}'_j$, and writes them back to global memory. Since all these operations occur across the $T\cdot n_g$ blocks, there are $T\cdot n_g(3S_{\text{block}}\cdot d_g + 3(m+n+1)) = 3(\frac{m+n+1}{S_{\text{block}}\cdot d_g}+1)\cdot B\cdot N\cdot d$ global memory access. Compared to Algorithm 1, we can see that Algorithm 2 reduces the number of global memory accesses and atomic adds by a factor of $\frac{1}{S_{\text{block}}\cdot d_g}$.

5 Results

Revisiting Speed, Utilization and Stalls

Using our experimental configurations from Section 3, we verify the efficacy of Algorithm 2 compared to Algorithm 1. We implement Algorithm 2 in Triton (Tillet, Kung, and Cox 2019) and run our experiments on a 4060 Ti. First, we compare the backward pass of the FlashKAT group-wise rational function with that of the KAT group-wise rational function by measuring cycles, time, and throughput utilization for SMs and memory. As shown in Table 3, the FlashKAT group-wise rational function achieves a speedup of 140.5x over the KAT group-wise rational function, with improvements across SM, cache, and memory utilization, reducing backward pass time.

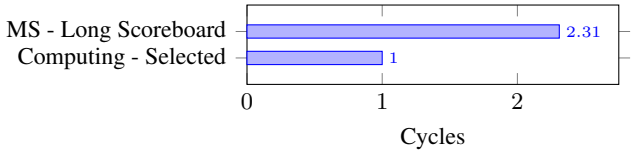

Figure 3: The warp-state statistics for the FlashKAT group-wise rational function backward pass. "Computing - Selected" is where the warp does useful computation; all the others are memory stall (MS) states.

Secondly, we again conduct the warp state statistics analysis on the backward pass of the FlashKAT group-wise rational function. From observing the warp state statistics in Figure 3, we can see that only the "Stall Long Scoreboard" is still longer than the "Selected" compute state; all the other 8 memory stalls are shorter than compute. Furthermore, the "Stall Long Scoreboard" has reduced from 981.51 cycles to 2.31 cycles. This result indicates the new kernel spends more time performing computations than waiting for memory transfers.

Image Recognition Training Speed and Accuracy

We validate that FlashKAT's training capabilities are consistent with KAT by training it on ImageNet-1k (Russakovsky et al. 2015). During training, we measure FlashKAT's training throughput to compare it with KAT, ViT, and DeiT. The data loader time is excluded from this measurement because it depends on external factors (e.g., CPU, main memory, etc.). We measure throughput in images/second as the average of 100 samples, and provide 95% confidence intervals.

Our training procedure follows the same curriculum as (Yang and Wang 2024), using the hyperparameters of DeiT (Touvron et al. 2021). As such, we train using a batch size of 1024 with the AdamW optimizer (Loshchilov and Hutter 2017). For the models, we apply Mimetic initialization to the attention layers (Trockman and Kolter 2023) and set the patch size to 16. In GR-KAN, each of the 8 groups shares its 6 numerator coefficients and has 4 unique denominator coefficients. The first layer of GR-KAN has its group-wise rational

Model	Cycles	Time	SM Thp. (%)	L1 Thp. (%)	L2 Thp. (%)	HBM Thp. (%)
KAT	2.4T	1.03 s	1.97	4.38	5.24	1.01
FlashKAT	16.9M	7.33 ms	32.24	34.14	44.76	92.05

Table 3: A comparison between the group-wise rational function backward pass for KAT and FlashKAT using FLOPs, cycles, time, SM, and memory throughput.

Model	#Param.	Top-1	Train Thp. (images/s)
ViT-T	5.7 M	72.7	8954.97 ($\pm$ 17.36)
DeiT-T	5.7 M	72.2	8954.97 ($\pm$ 17.36)
KAT-T	5.7 M	74.6	87.73 ($\pm$0.01)
FlashKAT-T	5.7 M	74.6	6317.90 ($\pm$2.65)
ViT-S	22.1 M	78.8	5311.71 ($\pm$ 6.58)
DeiT-S	22.1 M	79.8	5311.71 ($\pm$ 6.58)
KAT-S	22.1 M	81.2	43.28 ($\pm$ 0.01)
FlashKAT-S	22.1 M	81.4	3741.91 ($\pm$ 0.08)
ViT-B	86.6 M	79.1	2457.15 ($\pm$ 2.08)
DeiT-B	86.6 M	81.8	2457.15 ($\pm$ 2.08)
KAT-B	86.6 M	82.3	21.24 ($\pm$ 0.02)
FlashKAT-B	86.6 M	82.2	1801.75 ($\pm$ 0.683)

Table 4: Comparative analysis of model performance and computational efficiency on ImageNet-1K.

KAT		
Gradient	Mean Absolute Error	Variance
$\mathbf{dA}$	8.84×10^{-2} ($\pm 3.40 \times 10^{-3}$)	1.45×10^{-2}
$\mathbf{dB}$	9.63×10^{-2} ($\pm 9.87 \times 10^{-3}$)	8.11×10^{-2}
FlashKAT		
Gradient	Mean Absolute Error	Variance
$\mathbf{dA}$	8.42×10^{-4} ($\pm 3.28 \times 10^{-5}$)	1.35×10^{-6}
$\mathbf{dB}$	9.81×10^{-4} ($\pm 1.15 \times 10^{-4}$)	1.11×10^{-5}

Table 5: A comparison of mean absolute error (with 95% confidence intervals) and variances.

function initialized to the identity function, and the second layer is initialized to a Swish function (Ramachandran, Zoph, and Le 2017). For data augmentation and regularization techniques, we apply RandAugment (Cubuk et al. 2020), Mixup (Zhang et al. 2017), CutMix (Yun et al. 2019), Random Erasing (Zhong et al. 2020), weight decay, Label Smoothing (Szegedy et al. 2016) and Stochastic Depth (Huang et al. 2016). We conduct all experiments using an H200 GPU. The Appendix provides the complete set of model configurations and training hyperparameters.

We provide the image recognition accuracy of FlashKAT-T, FlashKAT-S, and FlashKAT-B models on the ImageNet-1k dataset in Table 4. Importantly, FlashKAT-T, FlashKAT-S, and FlashKAT-B all match their original counterparts and continue to outperform their ViT and DeiT alternatives in image classification. We also report the respective training throughput in Table 4. Our results demonstrate that the proposed FlashKAT-T, FlashKAT-S, and FlashKAT-B are 72.0x, 86.5x, and 84.5x faster than their KAT counterparts. This significantly reduces the performance gap to ViT, making KAT-based models a viable option for vision tasks.

Reduced Gradient Rounding Error

In addition to a significant speedup, the proposed FlashKAT also reduces rounding errors when computing gradients for the group-wise rational function coefficients. This is a byproduct of our method, accumulating gradients in each block through a reduction sum rather than atomic adds. Table 5 demonstrates that the mean absolute error in gradient outputs in FlashKAT is orders of magnitude smaller than in KAT. As such, FlashKAT is less susceptible to rounding errors that may contribute toward unstable gradient updates (e.g., low-precision training). Due to space limitations, we provide more details in the Appendix.

6 Conclusion

Although KAN has achieved success, it has yet to see widespread adoption in large-scale NLP and computer vision tasks due to its increased parameter count, higher computational cost, lack of GPU hardware optimization, and training instability. KAT has addressed many of KAN's shortcomings with the introduction of GR-KAN, but it still suffers from slow training. In this work, we conduct experiments to understand the source of the slowdown and identify that slow training speeds are a byproduct of memory stalls in the backward pass, a discovery that eluded prior work (Yang and Wang 2024; Molina, Schramowski, and Kersting 2019; Delfosse et al. 2020, 2021). To address this bottleneck, we propose a novel restructuring of the group-wise rational function that allows efficient gradient accumulation by minimizing atomic add usage. The resulting FlashKAT achieves an 86.5x speedup over KAT, while reducing gradient rounding errors. The application of our work both enables wider adoption and expedites research on KAT.

Limitations. A main drawback preventing the use of KAN in the Transformer is KAN's increased computational cost. With GR-KAN, KAT successfully reduced FLOPs, allowing it to nearly match the speed of a standard Transformer on the forward pass. Still, it remains orders of magnitude slower on the backward pass, making training computationally burdensome. Our proposed FlashKAT, with a significant speedup, has largely closed the performance gap, but is still about 25% slower than a standard Transformer. Even so, our work is an important step toward closing the gap between the KAT and standard Transformer in training speed, motivating future work to narrow it further.

References

Biswas, K.; Banerjee, S.; and Pandey, A. K. 2021. Orthogonal-Padé Activation Functions: Trainable Activation functions for smooth and faster convergence in deep networks. *arXiv preprint arXiv:2106.09693*.

Chen, Z.; Gundavarapu; and DI, W. 2024. Vision-KAN: Exploring the Possibility of KAN Replacing MLP in Vision Transformer. https://github.com/chenziwenhaoshuai/Vision-KAN.git.

Coffman, C.; and Chen, L. 2025. MatrixKAN: Parallelized Kolmogorov-Arnold Network. *arXiv preprint arXiv:2502.07176*.

Cubuk, E. D.; Zoph, B.; Shlens, J.; and Le, Q. V. 2020. Randaugment: Practical automated data augmentation with a reduced search space. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition workshops*, 702–703.

Delfosse, Q.; Schramowski, P.; Molina, A.; Beck, N.; Hsu, T.-Y.; Kashef, Y.; Rüling-Cachay, S.; and Zimmermann, J. 2020. Rational Activation functions. https://github.com/ml-research/rational_activations.

Delfosse, Q.; Schramowski, P.; Molina, A.; and Kersting, K. 2021. Recurrent Rational Networks. *arXiv preprint arXiv:2102.09407*.

Dosovitskiy, A.; Beyer, L.; Kolesnikov, A.; Weissenborn, D.; Zhai, X.; Unterthiner, T.; Dehghani, M.; Minderer, M.; Heigold, G.; Gelly, S.; et al. 2020. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*.

Huang, G.; Sun, Y.; Liu, Z.; Sedra, D.; and Weinberger, K. Q. 2016. Deep networks with stochastic depth. In *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part IV 14*, 646–661. Springer.

Koenig, B. C.; Kim, S.; and Deng, S. 2024. KAN-ODEs: Kolmogorov–Arnold network ordinary differential equations for learning dynamical systems and hidden physics. *Computer Methods in Applied Mechanics and Engineering*, 432: 117397.

Kolmogorov, A. N. 1961. *On the representation of continuous functions of several variables by superpositions of continuous functions of a smaller number of variables*. American Mathematical Society.

Le, T. X. H.; Tran, T. D.; Pham, H. L.; Le, V. T. D.; Vu, T. H.; Nguyen, V. T.; Nakashima, Y.; et al. 2024. Exploring the limitations of kolmogorov-arnold networks in classification: Insights to software training and hardware implementation. In *2024 Twelfth International Symposium on Computing and Networking Workshops (CANDARW)*, 110–116. IEEE.

Li, C.; Liu, X.; Li, W.; Wang, C.; Liu, H.; Liu, Y.; Chen, Z.; and Yuan, Y. 2025. U-kan makes strong backbone for medical image segmentation and generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, 4652–4660.

Liu, Z.; Wang, Y.; Vaidya, S.; Ruehle, F.; Halverson, J.; Soljačić, M.; Hou, T. Y.; and Tegmark, M. 2024. Kan: Kolmogorov-arnold networks. *arXiv preprint arXiv:2404.19756*.

Loshchilov, I.; and Hutter, F. 2017. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*.

Luo, W.; Fan, R.; Li, Z.; Du, D.; Wang, Q.; and Chu, X. 2024. Benchmarking and dissecting the nvidia hopper gpu architecture. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 656–667. IEEE.

Molina, A.; Schramowski, P.; and Kersting, K. 2019. Padé activation units: End-to-end learning of flexible activation functions in deep networks. *arXiv preprint arXiv:1907.06732*.

Ramachandran, P.; Zoph, B.; and Le, Q. V. 2017. Searching for activation functions. *arXiv preprint arXiv:1710.05941*.

Russakovsky, O.; Deng, J.; Su, H.; Krause, J.; Satheesh, S.; Ma, S.; Huang, Z.; Karpathy, A.; Khosla, A.; Bernstein, M.; et al. 2015. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115: 211–252.

Szegedy, C.; Vanhoucke, V.; Ioffe, S.; Shlens, J.; and Wojna, Z. 2016. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2818–2826.

Tillet, P.; Kung, H. T.; and Cox, D. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2019, 10–19. New York, NY, USA: Association for Computing Machinery. ISBN 9781450367196.

Touvron, H.; Cord, M.; Douze, M.; Massa, F.; Sablayrolles, A.; and Jégou, H. 2021. Training data-efficient image transformers & distillation through attention. In *International conference on machine learning*, 10347–10357. PMLR.

Trockman, A.; and Kolter, J. Z. 2023. Mimetic initialization of self-attention layers. In *International Conference on Machine Learning*, 34456–34468. PMLR.

Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, Ł.; and Polosukhin, I. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

Wang, Y.; Sun, J.; Bai, J.; Anitescu, C.; Eshaghi, M. S.; Zhuang, X.; Rabczuk, T.; and Liu, Y. 2024. Kolmogorov Arnold Informed neural network: A physics-informed deep learning framework for solving forward and inverse problems based on Kolmogorov Arnold Networks. *arXiv preprint arXiv:2406.11045*.

Yang, X.; and Wang, X. 2024. Kolmogorov-arnold transformer. *arXiv preprint arXiv:2409.10594*.

Yu, R.; Yu, W.; and Wang, X. 2024. Kan or mlp: A fairer comparison. *arXiv preprint arXiv:2407.16674*.

Yun, S.; Han, D.; Oh, S. J.; Chun, S.; Choe, J.; and Yoo, Y. 2019. Cutmix: Regularization strategy to train strong classifiers with localizable features. In *Proceedings of the IEEE/CVF international conference on computer vision*, 6023–6032.

Zhang, H.; Cisse, M.; Dauphin, Y. N.; and Lopez-Paz, D. 2017. mixup: Beyond empirical risk minimization. *arXiv preprint arXiv:1710.09412*.

Zhong, Z.; Zheng, L.; Kang, G.; Li, S.; and Yang, Y. 2020. Random erasing data augmentation. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, 13001–13008.