RONDO: INTERLEAVED DATAPATHS FOR DIVISION-BEARING NEURAL RECURRENCES

Anonymous authors
Paper under double-blind review

ABSTRACT

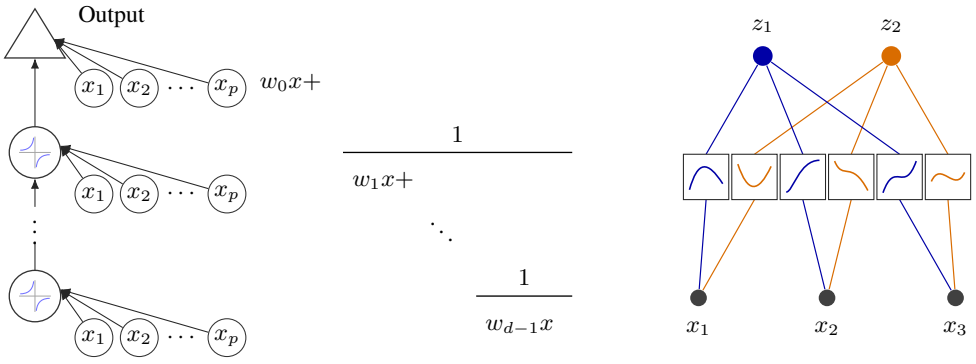
A neural network's parameter count governs the memory footprint and energy consumption of each deployment. Continued-fraction networks (CoFrNets) and Chebyshev Kolmogorov-Arnold networks (KAN) lower that count by replacing linear layers with learned rational and polynomial functions. Their functional form explains each prediction, suiting credit and fraud scoring under latency and energy budgets. Nevertheless, both architectures evaluate many second-order recurrences per layer, each step depending on the two before it. Existing hardware tends not to meet both latency and energy budgets. For instance, a flagship GPU needs 6.3 µs per input with launch, whereas a CPU core processes in 0.36 µs at far higher energy. We propose RONDO, an accelerator computing these recurrences with multiply-adds and at most one division, interleaving independent runs on one pipelined unit. We evaluate RONDO on a Kintex-7 FPGA and as an ASIC using the ASAP7 7 nm library, against RTX 5090 GPU, Ryzen 9960X CPU core, Ara vector processor, the PolyKAN library and SparseKAN accelerator. As a synthesized ASIC, RONDO answers 2.0 to 3.3× faster than the GPU with 6.2 to 10× less energy, and uses 35× less energy than the CPU core. On KAN layers, it answers 9.8 to 19× faster than PolyKAN with 54 to 135× less energy, and 38× faster than SparseKAN with 8.8× less energy. RONDO retains 99.7% of the ideal cycle reduction from two lanes to eight proving its scalability, compared to Ara's 44.5%.

1 INTRODUCTION

The parameter count of a neural network governs its deployment cost, setting the memory requirement and energy consumption for each inference. A conventional network alternates linear layers with fixed activation functions. In contrast, continued-fraction networks (CoFrNets) build each output from a learned continued fraction (Puri et al., 2021). Kolmogorov-Arnold networks (KANs) place a learned polynomial or spline on every connection (Liu et al., 2025). These learned functions reach the same accuracy with fewer parameters, lowering that cost. Continued-fraction language models perform competitively with GPT2-xl and Llama3 using one-half to two-thirds of their parameters (Dhurandhar et al., 2026), and a KAN outperforms a multilayer perceptron with 100× more parameters on a differential equation (Liu et al., 2025). In these networks, the learned function also explains each prediction, unlike post-hoc explanations of a trained network (Rudin, 2019).

These two families share a computation pattern that existing hardware handles poorly. A CoFrNet computes each output with a *ladder*, as illustrated in Figure 1a (Puri et al., 2021). Each of its d levels applies a learned linear function to the input and adds the reciprocal of the level below. A Chebyshev KAN places a learned polynomial on every edge and sums the edges at each output as shown in Figure 1b (Liu et al., 2025; SS et al., 2024). As Section 2.1 shows, both reduce to the same second-order recurrence, in which each step combines the two preceding steps using input-dependent coefficients. Existing hardware derives its throughput from independent multiply-accumulate operations (Kung, 1982; Jouppi et al., 2017). However, each step of this recurrence must wait for the one before it. As a result, neither common platform meets both latency and energy budgets as detailed in Section 5.2. A flagship GPU needs 6.3 µs per input, dominated by launch and serialization rather than arithmetic, while a single CPU core reaches 0.36 µs only by paying instruction and data-movement overheads that cost far more energy than the arithmetic itself. Consequently, these models cannot run where their compactness and interpretability matter most, such as latency- and energy-constrained edge and real-time systems.

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



(a) A CoFrNet ladder of depth d. Each stage applies a learned affine function of x and one reciprocal, evaluating the continued fraction beside it.

(b) A Chebyshev KAN layer. Each edge carries a learned polynomial, and each output sums its incoming edges.

Figure 1: The two architectures compute the same second-order recurrence, in which each step combines the two steps before it, weighted by coefficients that depend on the input.

Accelerating this recurrence raises three challenges. First, *division latency*: evaluating the nested fraction directly requires one division per level, and each division waits for the level below, which serializes d slow divisions per ladder. Second, *pipeline utilization*: hardware reaches high clock frequencies by splitting arithmetic into pipeline stages that work on different operations at once, whereas a recurrence step cannot start until the previous step leaves the pipeline. A five-stage unit therefore sits idle four cycles in five. Third, *coefficient generation*: the dot products that produce each coefficient perform about 98% of the arithmetic, and the direct ways to speed them up either lower the clock frequency or add input-selection logic that dominates the area. A design must also scale without losing efficiency per added unit, as a layer holds many independent ladders and a continued-fraction language model evaluates them for every token (Dhurandhar et al., 2026).

To address these challenges, we propose RONDO, an accelerator that converts serial dependence into independent work. RONDO computes each ladder through its continuants, the numerator and denominator that the fraction accumulates level by level (Jones & Thron, 1980). This replaces the d divisions with multiply-adds and at most one final reciprocal, clamped away from zero. Furthermore, RONDO interleaves the independent ladders of a layer on one pipelined unit. While one ladder waits for its previous step, the others fill the idle stages. This raises throughput and lowers latency and energy. Finally, RONDO scales almost linearly by replication, as shown in Section 5.3, with potential for server-level workloads. When scaled to occupy the same chip area as the GPU on equivalent memory, RONDO is estimated to achieve 4.9× the GPU's throughput (Appendix A.5).

In this paper, our main contributions, measured in Section 5, are:

- We introduce RONDO, an accelerator for input-dependent second-order recurrences, enabling CoFrNets and Chebyshev KANs to share one datapath of multiply-adds and at most one reciprocal.
- We develop an interleaving schedule that keeps more ladders in flight than pipeline stages, removing stalls without reading pending results, and scales across lanes that share no state.
- RONDO is evaluated on a Kintex-7 FPGA and in ASAP7 synthesis with three trained models, where, as synthesized, it runs 2.0 to 3.3× faster than an RTX 5090 GPU with 6.2 to 10× less energy, uses 35× less energy than a CPU core, and outperforms the PolyKAN library for GPUs and SparseKAN accelerators by 9.8 to 38× in latency and 8.8 to 135× in energy. Moreover, RONDO retains 99.7% of the ideal fourfold cycle reduction from two to eight lanes.

2 Background, Related Work and Motivation

2.1 Background

A CoFrNet is built from *ladders* (Puri et al., 2021), one of which Figure 1a draws. Each circle in that figure is a rung, and a ladder of depth d has d of them. Rung k of ladder j reads the whole input

$x \in \mathbb{R}^p$. Figure 1a labels its p components x_1 to x_p and draws them as the arrows entering every circle. Each rung applies its own affine map to them, $a_{j,k} = W_{j,k} \cdot x + b_{j,k}$. Here $W_{j,k} \in \mathbb{R}^p$ is a learned weight vector and $b_{j,k}$ a learned bias. Evaluation starts at the deepest rung $a_{j,d-1}$, drawn at the bottom, and each rung outward adds its affine term to the reciprocal of the value passed to it, the operation marked inside each circle. The ladder value is the reciprocal of the outermost rung's value, $z_j = 1/(a_{j,0} + 1/(a_{j,1} + \cdots + 1/a_{j,d-1}))$, written beside the circles in the figure. The triangle at the top adds one further affine term of the input to this value. A CoFrNet layer places L such ladders side by side on the same input and combines their outputs with a linear readout. Evaluated as written, each ladder divides once at every rung, and each rung waits for the deeper rung it reads. Writing each rung as a 2×2 matrix and multiplying these matrices produces the *continuants* of the fraction (Jones & Thron, 1980). The continuants follow the recurrence $K_k = a_{j,k}K_{k+1} + K_{k+2}$, started from $K_d = 1$ and $K_{d+1} = 0$, in which each continuant depends on the two deeper ones, and give $z_j = K_1/K_0$ with a single division, which Section 4.2 exploits. CoFrNets reach accuracy comparable to or better than other interpretable models on seven tabular, text and image datasets, and CoFrGeNet extends them to the attention and feed-forward blocks of GPT2-xl and Llama3, remaining competitive with the original models on downstream tasks with one half to two thirds of the parameters (Dhurandhar et al., 2026).

A Kolmogorov-Arnold network (KAN) replaces the weights of a fully connected layer with learnable univariate functions (Liu et al., 2025). As Figure 1b draws, the edge from input i to output o applies its own function to x_i, held in the box on that edge, and each output node at the top sums the edges reaching it. A Chebyshev KAN builds these functions from Chebyshev polynomials (SS et al., 2024). For each input, the layer squashes x_i to $\tilde{x}_i = \tanh(x_i)$ and climbs the polynomial degrees one at a time, $T_0 = 1$, $T_1 = \tilde{x}_i$ and $T_{n+1} = 2\tilde{x}_i T_n - T_{n-1}$, where T_n is the degree-n polynomial, up to a maximum degree D. Each edge weights these polynomials by its own learned coefficients $c_{i,o,n}$, giving $y_o = \sum_i \sum_{n=0}^{D} c_{i,o,n} T_n(\tilde{x}_i)$. This climb is the continuant recurrence with every coefficient set to $2\tilde{x}_i$ and the last sign reversed, with the same serial dependence as a ladder and no final division.

2.2 Related Work

Matrix-product accelerators. Systolic arrays, tensor processors and GPUs optimize multiply-accumulate operations (Kung, 1982; Jouppi et al., 2017). They lack an efficient element-wise reciprocal and draw throughput from mutually independent operations. The recurrence conflicts with both: the ladder needs a reciprocal, and each step depends on the previous two. In our experiments, a fused Triton kernel on an RTX 5090 reaches only 29% of the card's bf16 matrix-multiply rate (Appendix A.5).

KAN accelerators. Existing KAN accelerators remove the nonlinear evaluation from the datapath. KAN-SAs replaces recursive B-spline evaluation with a lookup table that feeds a systolic array (Errabii et al., 2026), and KANELÉ maps quantized splines to FPGA lookup tables (Hoang et al., 2026). Compute-in-memory designs store piecewise-linear segments (Huang et al., 2025; 2026), while BiKA replaces the splines with learned thresholds (Liu et al., 2026). On GPUs, PolyKAN interpolates the Chebyshev basis from a precomputed table (Yu et al., 2026). These approaches require a univariate function of a bounded input. A ladder, in contrast, depends on d affine combinations of all p inputs and diverges where its denominator approaches zero, which rules out tabulation. Its division also rules out reduction to a single matrix product. Among released designs, only SparseKAN computes a recurrence directly, evaluating Gram polynomials in floating point (Fuad & Chen, 2026).

Recurrent and rational operators. State-space models evaluate an associative linear recurrence that admits a parallel-prefix scan (Blelloch, 1990) and reduces to multiply-accumulate (Gu et al., 2022; Gu & Dao, 2023). The continuant recurrence is also associative, and Section 4.2 explains why RONDO does not evaluate it as a scan. Rational and Padé activations learn a ratio of two polynomials with input-independent coefficients and no recurrence (Molina et al., 2020; Boullé et al., 2020).

Table 1: Bottlenecks identified by the three steps, and the sections that address them.

	Bottleneck	Exposed by	Addressed in
P1	d serial divisions	step 1	Section 4.2
P2	scan performs $8\times$ the required multiply-adds	step 1	Section 4.2
P3	combinational floating point limits the clock	step 3	Section 4.3
P4	a recurrence cannot fill a pipeline	step 3	Section 4.3
P5	coefficient routing dominates area	step 2	Section 4.4
P6	coefficient generation performs 98% of the work	step 1	Section 4.5
P7	chained accumulators lose more clock than cycles	step 3	Section 4.5
P8	idle units toggle at full rate	step 2	Section 4.6

2.3 MOTIVATION

Existing hardware leaves this recurrence without an efficient evaluator. *First*, matrix-product accelerators assume independent operations and provide no efficient reciprocal. *Second*, KAN accelerators tabulate univariate functions. *Third*, vector processors such as Ara provide the floating-point division a ladder needs (Perotti et al., 2022), yet each added lane widens a single datapath controlled by one instruction stream, so all lanes stall together on each dependent step. In our experiments, Ara retains only 44.5% of ideal speedup from two to eight lanes, and the continuant form speeds Ara up by only 1.03×. On Ara, the dominant cost thus lies in the dependence between steps, not in division, which motivates RONDO's interleaving schedule.

3 BOTTLENECK LOCALIZATION

Section 1 introduced three challenges in accelerating second-order recurrences: division latency, pipeline utilization and coefficient generation. This section describes the procedure we used to localize and quantify these bottlenecks. We use a CoFrNet layer as the running example, as it contains both the recurrence and a final division. The layer is $L = 64$ independent copies of the ladder drawn in Figure 1a, each of depth $d = 7$ over $p = 40$ input features, matching the trained models of Section 5, and is implemented on a Xilinx Kintex-7 XC7K325T. All measurements are taken from placed and routed designs, with power estimated from simulated switching activity following the rules of Appendix A.1. The procedure applies three steps to every stage of the datapath.

Step 1: Count operations. For each stage, we count two quantities: the arithmetic operations required by the mathematical computation, and those performed by the hardware. For example, one step of the recurrence needs one fused multiply-add. A parallel-prefix scan removes the serial dependence, but each of its steps multiplies two 2×2 matrices, which needs up to eight. The per-stage totals also show where the work lies: the d coefficients of a ladder need $2dp$ operations and the recurrence needs $2(d-1)$, which places about 98% of the arithmetic in coefficient generation. Figure 1a shows the asymmetry that produces this ratio, as every rung of the ladder reads all p inputs, whereas the recurrence passes one value from each rung to the next.

Step 2: Measure cost per module. For each stage, we count two quantities: the arithmetic operations required by the mathematical computation, and those performed by the hardware. For example, one step of the recurrence needs one fused multiply-add. A parallel-prefix scan removes the serial dependence, but each of its steps multiplies two 2×2 matrices, which needs up to eight. The per-stage totals also show where the work lies: the d coefficients of a ladder need $2dp$ operations and the recurrence needs $2(d-1)$, which places about 98% of the arithmetic in coefficient generation. Figure 1a shows the asymmetry that produces this ratio, as every rung of the ladder reads all p inputs, whereas the recurrence passes one value from each rung to the next.

Step 3: Evaluate changes by throughput. Finally, we evaluate each design change by throughput, defined as clock frequency multiplied by the number of ladders completed per cycle. Neither factor alone suffices, since design changes routinely improve one at the expense of the other. Pipelining the floating-point unit improves its clock frequency by 3.5×, yet **worsens cycles per ladder fivefold**, as each step must wait for the previous result. Chaining multiply-accumulates in the

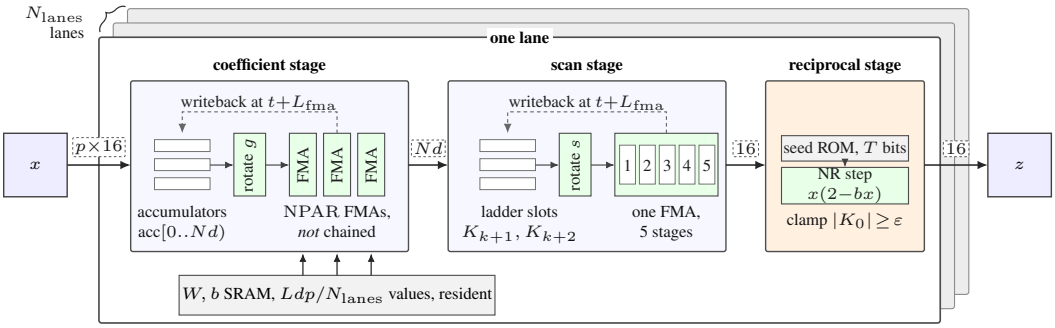


Figure 2: RONDO, one lane. Widths in bits; dashed arrows are the loop-carried dependencies that motivate the rotation.

coefficient generator reduces cycles per ladder by 5.3× while **lowering the clock frequency by 8×**. In both cases, the improved factor suggests a benefit that throughput does not confirm.

Table 1 lists the eight resulting bottlenecks, labeled P1 to P8, together with the step that exposed each and the section that addresses it. P1 and P2 constitute the first challenge of Section 1, P3 and P4 the second, and P5 to P7 the third. P8, idle units that consume power, is an additional bottleneck exposed by step 2.

4 Architecture

RONDO computes one layer of a model built on a second-order recurrence. We keep the CoFrNet layer of Section 3 as the running example. It takes an input $x \in \mathbb{R}^p$, combines it with weights held in on-chip memory, and produces L ladder outputs $z \in \mathbb{R}^L$. The design rests on the idea that interleaving many independent computations in a layer enhances throughput while each serial computation waits for its previous step. This section applies that idea to the bottlenecks P1 to P8 of Table 1 in order, starting each design step from the observation that motivates it.

4.1 Overview

We first outline the organization of RONDO before detailing each stage. Figure 2 shows one lane in its bf16 configuration; RONDO replicates the lane N_{lanes} times. The coefficient stage turns x and the stored weights into the Nd coefficients of the lane's N ladders, each of depth d. The scan stage, named for its sequential sweep over the levels rather than for a parallel-prefix scan, computes the continuants K_1 and K_0 of each ladder. The reciprocal stage performs the one division, K_1/K_0, to produce z. Each compute stage stores its work in progress in a register array, the *state file*: one accumulator per coefficient in the coefficient stage and one slot per ladder in the scan stage. Two round-robin pointers (g and s in Figure 2) feed these entries into shallow arithmetic. The coefficient stage holds a build-time number NPAR of parallel fused multiply-adds, and the scan stage holds one fused multiply-add of $L_{\text{fma}} = 5$ stages. The dashed arrows mark loop-carried dependencies, which Section 4.3 resolves by scheduling rather than by extra logic.

4.2 Continuant Datapath (Addressing Bottlenecks P1, P2)

Division is the slowest floating-point operation. Every horizontal bar in the continued fraction beside the ladder of Figure 1a stands for one division. Each division takes its divisor from the fraction nested below it, and the d divisions of a depth-d ladder therefore run strictly in sequence (P1). An efficient datapath requires an equivalent form that postpones division to the end. The continuants of Section 2.1 provide such a form. For ladder j, with coefficients $a_{j,k} = W_{j,k}{\cdot}x + b_{j,k}$, the continuants satisfy Equation 1.

$$K_k = a_k K_{k+1} + K_{k+2}, \qquad K_d = 1,\ K_{d+1} = 0, \qquad z = K_1/K_0. \tag{1}$$

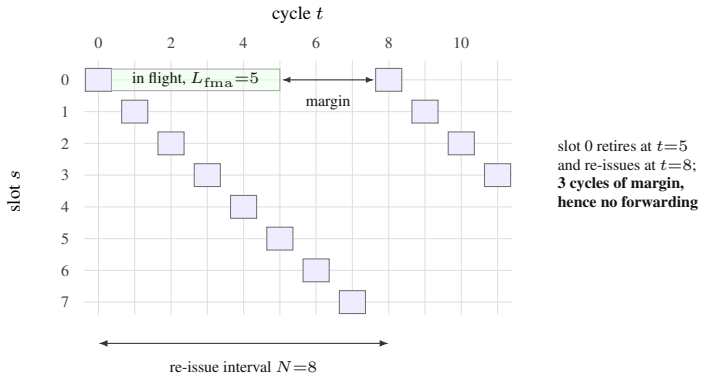


Figure 3: Slot issue over time, N=8, L_{fma}=5. Shaded cells are issues; the bar is slot 0 in flight.

Therefore, each step is now one fused multiply-add, and a single division remains at the end for any depth. The scan stage of Figure 2 therefore contains no divider. Dhurandhar et al. (2026) report the value of this identity in software; Appendix A.8 measures that value in hardware.

Parallelizing the recurrence itself appears intuitive, as the matrix product is associative and admits a parallel-prefix scan (P2). This approach, however, is ill-suited to a layer, for two reasons. A scan tree shortens the latency of a single ladder, whereas a layer already offers ample parallelism through its L independent ladders. A tree also carries the full 2×2 product and performs up to eight fused multiply-adds per step. Equation 1 instead needs only the first column of that product, which holds K_0 and K_1. The scan stage of Figure 2 is built on this reduction: each ladder slot holds the two continuants it carries forward, and one fused multiply-add serves every slot. Keeping only that column saves 27,553 LUTs and 14 DSP blocks against the tree form.

Furthermore, postponing the division makes it safer. The nested form can divide by a near-zero value at any of its d levels, producing an infinity or a NaN, and each level would need its own guard. Equation 1 divides once, and one guard therefore suffices. The reciprocal stage of Figure 2 implements that guarded division in two steps, without using a divider. It first clamps the denominator away from zero, $b = \mathrm{sgn}(K_0)\max(|K_0|, \varepsilon)$, where the *pole guard* ε is the smallest magnitude the divisor may take. It subsequently forms $1/b$ from a table lookup and and one Newton–Raphson step. The seed ROM is addressed by the leading bits of b and returns an initial estimate x_0, accurate to 12 bits at fp32 and 6 bits at bf16. One Newton–Raphson step, $x_1 = x_0(2 - b\,x_0)$, doubles the correct bits, covering the significand of either format. The stage returns $z = K_1 x_1$.

4.3 Pipelined Arithmetic and Slot Interleaving (P3, P4)

The standard way to improve clock frequency is to pipeline the arithmetic. Splitting the HardFloat fused multiply-add (Hauser, 2019) into five registered stages increases its clock frequency from 47.8 to 168.9 MHz. The LUT count barely increases, 1,995 against 1,985, and the only cost is 417 flip-flops (P3). A recurrence, however, cannot fill such a pipeline. Step k needs K_{k+1} from the previous step, therefore a five-stage unit idles four out of five cycles while it waits (P4).

The key observation is that the pipeline need not idle while one ladder waits. The layer holds many independent ladders, and any of them can use the idle stages. RONDO keeps N ladders in the state file as *slots*, and the pointer issues one fused multiply-add per cycle, round-robin. Each slot returns exactly N cycles later. When N exceeds the latency of the pipeline, **the rotation itself guarantees that each result is ready before its slot returns**, so no stalls occur and no forwarding or hazard logic is needed. In Figure 3, slot 0 issues at t=0 and completes at t=5, and the rotation does not return to it until t=8.

The slot count N follows from the pipeline timing. Let L_{fma} denote the latency of the fused multiply-add, five cycles for its five stages. An operation issued at cycle t writes its result at the end of cycle $t+L_{\mathrm{fma}}$, and the result is readable from cycle $t+L_{\mathrm{fma}}+1$. The number of slots must therefore satisfy $N \geq L_{\mathrm{fma}} + 1$. At $N = L_{\mathrm{fma}}$, the reissued slot reads the value from two steps earlier and silently produces an incorrect result.

4.4 On-Chip Coefficient Generation (P5)

Moving data can cost more than computing it. Supplying coefficients over one port shared by all L ladders makes the routing the dominant cost (P5). The multiplexer that routes each coefficient to its ladder takes 9,768 LUTs, 63% of the design, against 137 LUTs for the control logic. The remedy is to compute each coefficient where it is consumed. RONDO generates the coefficients inside each lane, and only x crosses the lane boundary. Each lane owns a contiguous range of ladders, so its results map onto z through fixed wiring, and the multiplexer disappears. The weights are stored on chip: Ldp words in total, split across the per-lane W, b memories of Figure 2. For the running example that is 17,920 words, 72 KB at fp32, well within the 16 MB of block RAM on the device.

4.5 Parallel Accumulation in the Coefficient Stage (P6, P7)

Once the recurrence is fast, the bottleneck moves to the coefficients. The coefficient stage performs about 98% of the arithmetic of the layer (P6). The conventional remedy is to chain NMAC multiply-accumulates so that one cycle covers NMAC inputs, but the chain lengthens the combinational path (P7). One combinational fused multiply-add runs at 47.8 MHz, whereas four chained units limit the design to 11.6 MHz on a critical path of 275 logic levels. Pipelining the chain does not help: each accumulation is itself loop-carried and would stall just as the recurrence did before interleaving.

The observation of Section 4.3 applies again: the remedy lies in independent work. **Every coefficient is an independent accumulation**, and a lane's N ladders at depth d contain Nd of them. The coefficient stage feeds NPAR parallel fused multiply-adds in round-robin order, each from its own group of accumulators, so each accumulator is revisited every Nd/NPAR cycles, which must be at least $L_{\mathrm{fma}} + 1$. The combinational path remains one fused multiply-add for any NPAR, and each coefficient still accumulates over its p inputs in order, which keeps results bit-identical to a sequential evaluation.

4.6 Duty-Cycle Data Gating (P8)

An arithmetic unit that produces a result only occasionally still wastes power if it switches on every cycle. The reciprocal stage produces one result every seven cycles, one per ladder of depth d=7, yet draws 54% of the dynamic power of a lane. The reason is that its internal fused multiply-adds toggle on idle data (P8). RONDO therefore loads the data registers of each pipeline stage only when the stage holds valid data, while the valid bits themselves always advance. A build with gating disabled reproduces the ungated design exactly, which makes the comparison in Section A.8 a controlled one.

4.7 One Recurrence, Two Bases

The scan stage is not specific to continued fractions. It computes a set of independent second-order linear recurrence with input-dependent coefficients, of which the continued fraction is one case. The Chebyshev recurrence $T_{n+1} = 2x\,T_n - T_{n-1}$ differs from Equation 1 only in the sign of the term K_{k+2}. A build parameter inverts that sign on a registered operand outside the critical path; if the inversion is instead a run-time option, it costs one exclusive-or gate. Setting every coefficient to $2x$ turns the scan into a Chebyshev basis evaluator, with $a_{d-1} = x$ producing polynomials of the first kind and $a_{d-1} = 2x$ those of the second kind. The scan built for the backward path retains the full sequence $K_0 \ldots K_d$, which is exactly the basis that a Chebyshev KAN weights. Bases whose recurrence coefficient varies with degree, such as Legendre and Hermite polynomials, need a second multiplication inside the dependency chain and are outside the scope of this design.

4.8 Backward Path

The backward path reuses the quantities the forward pass has already produced. As Appendix A.3 derives, the gradient with respect to the ladder coefficients admits a closed form in the same continuants. Evaluating it requires no second division and no per-step state saved from the forward pass. A single reciprocal stage serves both directions, as the pair (K_0, K_1) produces the forward output and the pair $(K_0, 1)$ produces the $1/K_0$ that the gradient requires. The resulting datapath is a feed-forward chain of three multiplications. It carries no loop-carried dependency and therefore pipelines without interleaving. On the Kintex-7 of Section 3, after placement and routing, this

Table 2: Latency, energy and scaling of RONDO against existing hardware, with RONDO measured on the FPGA at eight lanes and synthesized for ASAP7. Baseline values are measured. Advantage is baseline over RONDO for latency and energy and RONDO over baseline for scaling, and values below 1× favor the baseline.

				RONDO		RONDO advantage	
baseline	metric	unit	baseline	FPGA	ASIC	FPGA	ASIC
CPU core, fp32	single-input latency	µs	0.36	2.53	0.485	0.14×	0.74×
CPU core, fp32	energy per ladder	nJ	61.3	149.1	1.75	0.41×	35×
RTX 5090 GPU, bf16	single-input latency	µs	1.02	2.20	0.311	0.46×	3.3×
RTX 5090 GPU, bf16	energy per ladder	nJ	4.43	52.2	0.72	0.085×	6.2×
RTX 5090 GPU, fp32	single-input latency	µs	0.99	2.53	0.485	0.39×	2.0×
RTX 5090 GPU, fp32	energy per ladder	nJ	17.51	149.1	1.75	0.12×	10.0×
Ara, fp32	scaling efficiency, 2 to 8 lanes	%	44.5	99.7	—	2.2×	—

chain reaches 133.8 MHz, within 4% of the forward scan and against 13.8 MHz for a combinational implementation. Two further gradients, $\partial f/\partial W$ and $\partial f/\partial x$, are required for training. These are an outer product and a matrix-vector product, they account for about 98% of a backward pass, and RONDO does not implement them, as Appendix A.4 discusses.

5 Evaluation

5.1 Experimental Setup

We implement RONDO on a Kintex-7 XC7K325T FPGA and synthesize it for ASAP7, a predictive 7 nm process (Clark et al., 2016), as detailed in Appendix A.9 and Appendix A.6. The workloads are CoFrNets trained on Waveform (p=40), Credit Card (p=23) and MAGIC (p=10), each with L=64 ladders of depth d=7, and a Chebyshev KAN layer with 64 outputs and degree 7 on the same datasets. Clock frequencies are reported after placement and routing, and power is estimated from simulated switching activity, following Appendix A.1. The baselines are one CPU core, a fused Triton kernel on an RTX 5090 GPU, the Ara vector processor (Perotti et al., 2022), and the KAN accelerators PolyKAN (Yu et al., 2026) and SparseKAN (Fuad & Chen, 2026).

5.2 Baseline Comparison

Table 2 compares RONDO with one CPU core, an RTX 5090 GPU and the Ara vector processor, taking each CPU and GPU value from that platform's best configuration for the metric. On the 28 nm FPGA, RONDO is slower than both 4 nm-class baselines. As synthesized for ASAP7, RONDO runs 2.0 to 3.3× faster than the GPU and uses 6.2 to 10× less energy per ladder. One CPU core remains 1.35× faster at fp32, while RONDO uses 35× less energy. Appendix A.5 details each measurement and its energy accounting. Scaling efficiency is the reduction in cycles per ladder divided by the increase in lane count. Ara retains only 44.5% of it from two to eight lanes, against 99.7% for RONDO, as Figure 5 in Appendix A.5 shows.

KAN accelerators. Table 3 compares RONDO with PolyKAN library that run on GPU and SparseKAN accelerator. PolyKAN replaces the basis recurrence with a lookup table (Yu et al., 2026). Its code is unavailable, and we therefore measure our reimplementation of its published algorithm on the RTX 5090. SparseKAN evaluates a three-term recurrence and releases its code (Fuad & Chen, 2026). We run its own layer kernel on our FPGA and synthesize it with our ASAP7 flow, changing only the basis to Chebyshev. As synthesized, RONDO runs 9.8 to 19× faster than PolyKAN with 54 to 135× less energy per output. Inside PolyKAN's own kernel, the recurrence alone runs 3.6 to 6.4× faster than the table that replaces it, as Appendix A.7 details. Against SparseKAN, RONDO runs 16.9× faster with 1.41× less energy on the same FPGA, and 38× faster with 8.8× less energy as synthesized. SparseKAN occupies 2.9× less area, although a single RONDO lane still runs 3.9× faster on the FPGA.

Table 3: Latency and energy of RONDO against PolyKAN, measured on an RTX 5090, and SparseKAN, measured on the same FPGA and synthesized with the same ASAP7 flow as RONDO.

				RONDO		RONDO advantage	
baseline	metric	unit	baseline	FPGA	ASIC	FPGA	ASIC
PolyKAN, fp32	single-input latency	µs	6.66	4.59	0.683	1.5×	9.8×
PolyKAN, fp32	energy per output	nJ	156.0	219.3	2.92	0.71×	54×
PolyKAN, bf16	single-input latency	µs	6.64	2.49	0.351	2.7×	19×
PolyKAN, bf16	energy per output	nJ	139.3	69.4	1.03	2.0×	135×
SparseKAN, fp32	single-input latency	µs	77.5 / 25.96	4.59	0.683	16.9×	38×
SparseKAN, fp32	energy per output	nJ	309.0 / 25.8	219.3	2.92	1.41×	8.8×

Table 4: Lane scaling of RONDO, L=64, p=40; clock at a 5.5 ns target, power at 125 MHz.

fmt	lanes	cycles/ladder	$F_{\max}$ (MHz)	LUTs	M ladders/s	area eff.	nJ/ladder
fp32	1	46.28	161.0	19,157	3.48	100%	175.1
fp32	2	23.16	157.3	38,316	6.79	97.6%	159.9
fp32	4	11.59	155.3	76,635	13.40	96.3%	152.4
fp32	8	5.81	146.7	150,488	25.25	92.4%	149.1
bf16	4	11.59	170.9	31,827	14.75	100%	51.6
bf16	8	5.81	168.9	63,523	29.07	98.8%	52.2

RONDO is verified against a 64-bit floating-point reference in Appendix A.11. In its default fp32 format, RONDO preserves every decision of the three trained models on all 16,206 test inputs.

5.3 Scaling and Energy

Table 4 shows how the speed, area and energy of RONDO change as lanes are added. Doubling the lanes halves the cycles per ladder, from 46.28 at one lane to 5.81 at eight, as lanes share no state. From one to eight lanes, LUTs grow 7.9× and throughput, the clock frequency divided by cycles per ladder, grows 7.3×. This is an area efficiency of 92.4%, where area efficiency is throughput per LUT relative to the smallest build of the same format. The shortfall comes from the clock frequency, which falls from 161.0 to 146.7 MHz as routing grows harder, while energy per ladder falls from 175.1 to 149.1 nJ. The bf16 builds scale alike with 2.41× fewer LUTs and 2.85× less energy per ladder. Beyond L/N lanes, eight here, larger layers scale by replicating layer instances that share one weight stream. A CoFrGeNet-size layer, however, holds 8.3 MB of bf16 weights, which exceed this device's 2 MB of block RAM and must be streamed from off-chip memory.

5.4 Limitations

The ASIC results are pre-layout synthesis on a predictive 7 nm library, a node behind the 4 nm-class CPU and GPU. bf16 is not usable at CoFrGeNet size, where RONDO requires fp32. The output projection, a production weight-loading path and the weight and input gradients are not implemented, no build exceeds eight lanes and two instances, and the reported latency excludes host transfer.

6 Conclusion

In this paper, we present RONDO, an accelerator for the input-dependent second-order recurrences of CoFrNets and Chebyshev KANs. RONDO computes each recurrence with multiply-adds and at most one reciprocal, and interleaves independent ladders on one pipelined unit without stalls. As synthesized for ASAP7, RONDO runs 2.0 to 3.3× faster than an RTX 5090 GPU with 6.2 to 10× less energy, uses 35× less energy than a CPU core, and runs 9.8 to 38× faster than the PolyKAN and SparseKAN accelerators. It also retains 99.7% of the ideal fourfold cycle reduction from two to eight lanes. RONDO is therefore a promising foundation for deploying parameter-efficient, interpretable models under tight latency and energy budgets.

AI USE STATEMENT

A large-language-model coding assistant was used during this project for three tasks. First, drafting and iterating on RTL, testbench, and Python tooling code. The authors reviewed and executed that code, and verified it against an independent fp64 golden model, before any result in this paper was reported. Second, drafting figure-generation scripts and diagrams, which the authors checked for numerical and visual accuracy against the underlying data. Third, drafting and editing portions of this manuscript, which the authors reviewed, fact-checked against the measurements, and revised. Generative AI tools were not used to produce the reported experimental results. RTL simulation, synthesis, and benchmark numbers were produced by the stated toolchain. Citations were not generated or selected without verification. We take responsibility for the final content of this work, including all text, claims, and artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This is a hardware-efficiency and computer-architecture study. It does not involve human subjects, personal data, or a dataset release. The underlying CoFrNet model code is not released with this paper, as noted in the reproducibility statement. The paper describes only the accelerator architecture, and uses the publicly published CoFrNet formulation (Puri et al., 2021). We are not aware of a dual-use or safety concern specific to a division-operator accelerator, beyond those that apply to machine -learning accelerator research generally.

REPRODUCIBILITY STATEMENT

The algorithm and architecture are described precisely enough to reimplement without access to our RTL. Section 4 gives the continuant recurrence, the boundary conditions, and the pole guard. Appendix A.2 gives pseudocode for the folded forward scan and the closed-form backward pass. Appendix A.3 gives the closed-form gradient with its derivation. Table 15 lists every tool version and parameter used to produce the reported numbers. The underlying model code is IBM-confidential and is not released with this submission. No result here depends on that code being available. The reference computation is fully specified by the public CoFrNet formulation (Puri et al., 2021) together with the recurrence given in Section 4.2. The verification methodology is described in Section A.11, with per-dataset results in Table 17 and per-module synthesis in Appendix A.10.

REFERENCES

Guy E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, 1990.

Nicolas Boullé, Yuji Nakatsukasa, and Alex Townsend. Rational neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

James Cherry and William Scott. OpenSTA: Parallax static timing analyzer. `https://github.com/parallaxsw/OpenSTA`, 2025.

Lawrence T Clark, Vinay Vashishtha, Lucian Shifren, Aditya Gujja, Saurabh Sinha, Brian Cline, Chandarasekaran Ramamurthy, and Greg Yeric. Asap7: A 7-nm finfet predictive process design kit. *Microelectronics Journal*, 53:105–115, 2016.

Amit Dhurandhar, Vijil Chenthamarakshan, Dennis Wei, Tejaswini Pedapati, Karthikeyan Natesan Ramamurthy, and Rahul Nair. Cofrgenet: Continued fraction architectures for language generation. In *Forty-third International Conference on Machine Learning*, 2026. URL `https://openreview.net/forum?id=wjc6oz63WL`.

Sohaib Errabii, Olivier Sentieys, and Marcello Traiola. Kan-sas: Efficient acceleration of kolmogorov-arnold networks on systolic arrays. In *2026 Design, Automation & Test in Europe Conference (DATE)*, pp. 1–7. IEEE, 2026.

Kazi Ahmed Asif Fuad and Lizhong Chen. Sparsekan: Compressing kolmogorov–arnold networks across basis functions, neurons, and bits. *arXiv preprint arXiv:2608.00859*, 2026.

Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, et al. Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration. In *58th ACM/IEEE Design Automation Conference (DAC)*, pp. 769–774, 2021.

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

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

John R. Hauser. Berkeley HardFloat. `http://www.jhauser.us/arithmetic/HardFloat.html`, 2019. Accessed 2026.

Duc Hoang, Aarush Gupta, and Philip C Harris. Kanelé: Kolmogorov–arnold networks for efficient lut-based evaluation. In *Proceedings of the 2026 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 44–55, 2026.

Wei-Hsing Huang, Jianwei Jia, Yuyao Kong, Faaiq Waqar, Tai-Hao Wen, Meng-Fan Chang, and Shimeng Yu. Hardware acceleration of kolmogorov-arnold network (kan) for lightweight edge inference. In *Proceedings of the 30th Asia and South Pacific Design Automation Conference*, pp. 693–699, 2025.

Wei-Hsing Huang, Jianwei Jia, Yuyao Kong, Faaiq Waqar, Tai-Hao Wen, Meng-Fan Chang, and Shimeng Yu. Hardware acceleration of kolmogorov–arnold network (kan) in large-scale systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2026.

William B. Jones and W. J. Thron. *Continued Fractions: Analytic Theory and Applications*. Encyclopedia of Mathematics and its Applications. Addison-Wesley, 1980.

Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pp. 1–12, 2017.

H. T. Kung. Why systolic architectures? *IEEE Computer*, 15(1):37–46, 1982.

Ian Kuon and Jonathan Rose. Measuring the gap between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):203–215, 2007.

Yuhao Liu, Salim Ullah, and Akash Kumar. Bika: Kolmogorov-arnold-network-inspired ultra lightweight neural network hardware accelerator. In *2026 27th International Symposium on Quality Electronic Design (ISQED)*, pp. 1–9. IEEE, 2026.

Ziming Liu, Yixuan Wang, Sachin Vaidya, Fabian Ruehle, James Halverson, Marin Soljacic, Thomas Hou, and Max Tegmark. Kan: Kolmogorov–arnold networks. In *International conference on learning representations*, volume 2025, pp. 70367–70413, 2025.

Alejandro Molina, Patrick Schramowski, and Kristian Kersting. Padé activation units: End-to-end learning of flexible activation functions in deep networks. In *International Conference on Learning Representations (ICLR)*, 2020.

Matteo Perotti, Matheus Cavalcante, Nils Wistoff, Renzo Andri, Lukas Cavigelli, and Luca Benini. A "new ara" for vector computing: An open source highly efficient risc-v v 1.0 vector processor design. In *2022 IEEE 33rd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 43–51. IEEE, 2022.

Isha Puri, Amit Dhurandhar, Tejaswini Pedapati, Karthikeyan Shanmugam, Dennis Wei, and Kush R. Varshney. CoFrNets: Interpretable neural architectures from continued fractions. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

Cynthia Rudin. Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nature machine intelligence*, 1(5):206–215, 2019.

Ananda Samajdar, Jan Moritz Joseph, Yuhao Zhu, Paul Whatmough, Matthew Mattina, and Tushar Krishna. A systematic methodology for characterizing scalability of DNN accelerators using SCALE-sim. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 58–68, 2020.

Sidharth SS, Keerthana AR, Anas KP, et al. Chebyshev polynomial-based kolmogorov-arnold networks: An efficient architecture for nonlinear function approximation. *arXiv preprint arXiv:2405.07200*, 2024.

Aaron Stillmaker and Bevan Baas. Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm. *Integration*, 58:74–81, 2017.

Clifford Wolf, Johann Glaser, and Johannes Kepler. Yosys-a free verilog synthesis suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, volume 97, pp. 1–6, 2013.

Mingkun Yu, Heming Zhong, Jiazhi Jiang, Dan Huang, and Yutong Lu. Polykan: A high-performance and universal gpu operator library for polynomial kolmogorov-arnold networks. In *Proceedings of the 40th ACM International Conference on Supercomputing*, pp. 1102–1114, 2026.

A Appendix

A.1 Measurement methodology

Four rules govern every measurement. First, frequency and area come from placed and routed designs with registered out-of-context boundaries, and a single placement carries a spread of about $\pm 5\%$. Second, power comes from simulating the post-route netlist on real data and annotating the routed design with that switching activity. Each power figure is quoted with its net-match rate, as an activity file that matches no nets silently falls back to Vivado's vectorless estimate, which overstated power by 53% on this design. Third, each design decision is priced against the alternative built from the same RTL through the same flow. Fourth, node-independent quantities, cycles per ladder and throughput per LUT, are reported apart from node-dependent ones.

A.2 Pseudocode: folded forward scan and closed-form backward

The pseudocode below gives the per-ladder computation of Section 4.2, with coefficients $a_0, \ldots, a_{d-1}$, partial numerators $b_1, \ldots, b_{d-1}$ in the generalized case, and pole guard ε.

```
# Forward: folded scan (one 2x2 MAC reused per cycle)
K[d] = 1 ; K[d+1] = 0
for k = d-1 down to 0:                  # b[k+1]=1, unit case
    K[k] = a[k]*K[k+1] + b[k+1]*K[k+2]
rK0 = reciprocal( sign(K[0]) * max(|K[0]|, eps) )
f   = K[1] * rK0                        # (* b[0], gen. case)

# Backward: scalar recurrence -> full K array (kscan), rK0 reused
K[d] = 1 ; K[d+1] = 0
for k = d-1 down to 0:
    K[k] = a[k]*K[k+1] + b[k+1]*K[k+2]
    store K[k]                          # kept on-chip
for j = 0 to d-1:                       # d parallel lanes
    ratio     = K[j+1] * rK0
    sign_j    = -1 if (j % 2 == 0) else +1
    grad_a[j] = sign_j * ratio * ratio * grad_out
# grad_b[j]: analogous closed form, derived in the next subsection
```

A.3 CLOSED-FORM GRADIENT: DERIVATION

We derive the closed form used in Section 4.8 by a discrete Wronskian (Casoratian) argument applied to the three-term recurrence of Equation 1. Fix an index $j \in \{0, \dots, d-1\}$ and write K_k for the forward continuants. Define the gradient sequence $G_k := \partial K_k / \partial a_j$. Differentiate Equation 1 with respect to a_j. For $k \neq j$ this gives the same recurrence, Equation 2:

$$G_k \;=\; a_k G_{k+1} + b_{k+1} G_{k+2}, \tag{2}$$

since a_k and b_{k+1} do not depend on a_j. Direct differentiation fixes the boundary values at $k = j$, giving $G_j = K_{j+1}$ and $G_{j+1} = 0$.

Both K and G solve the same recurrence for $k < j$. Define their Casoratian, Equation 3:

$$W_k \;:=\; K_k G_{k+1} - K_{k+1} G_k. \tag{3}$$

Substituting Equations 1 and 2 into Equation 3 gives the reduction $W_k = -\,b_{k+1}\, W_{k+1}$, and hence Equation 4:

$$W_0 \;=\; (-1)^j \Big(\textstyle\prod_{i=0}^{j-1} b_{i+1} \Big)\, W_j, \qquad W_j \;=\; K_j G_{j+1} - K_{j+1} G_j \;=\; -K_{j+1}^2, \tag{4}$$

using the boundary values $G_j = K_{j+1}$ and $G_{j+1} = 0$. Given that $f = K_1/K_0$, combining Equation 4 with the quotient rule yields Equation 5:

$$\frac{\partial f}{\partial a_j} = \frac{G_1 K_0 - K_1 G_0}{K_0^2} = \frac{W_0}{K_0^2} = (-1)^{j+1} \Big(\textstyle\prod_{i=0}^{j-1} b_{i+1} \Big) \Big(\frac{K_{j+1}}{K_0} \Big)^2. \tag{5}$$

In the unit case, where $b \equiv 1$, Equation 5 reduces to $\pm(K_{j+1}/K_0)^2$ with sign $(-1)^{j+1}$. This is exactly the operation listed in Appendix A.2. Square the ratio $K_{j+1} \cdot rK_0$ and flip the sign on even j, using the single reciprocal $rK_0 = 1/K_0$ already computed for the forward pass. The generalized-numerator gradient follows the same argument with the boundary shifted by one index. The shift arises as the recurrence depends on b_j in the equation for K_{j-1} rather than K_j. This gives Equation 6:

$$\frac{\partial f}{\partial b_j} = (-1)^j \Big(\textstyle\prod_{i=0}^{j-2} b_{i+1} \Big) \frac{K_j K_{j+1}}{K_0^2}. \tag{6}$$

Equations 5 and 6 were verified numerically against automatic differentiation and finite differences to within 2.4×10^{-6} relative error.

A.4 THE BACKWARD PATH

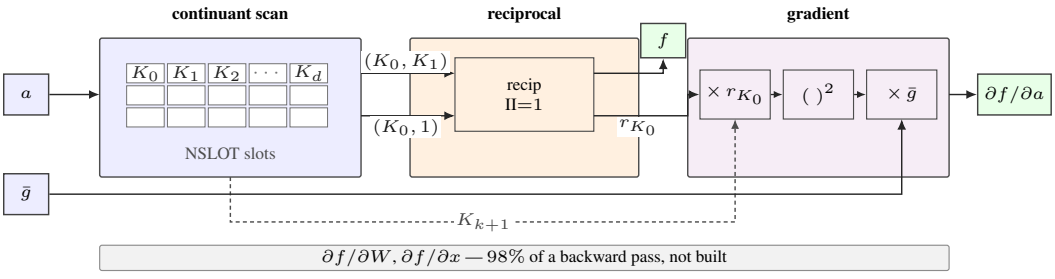


Figure 4: The backward path, one lane. The scan keeps the whole continuant sequence rather than the final pair, and the gradient reads K_{k+1} back (dashed) instead of recomputing it. One reciprocal serves both directions: (K_0, K_1) gives the forward output f and $(K_0, 1)$ gives r_{K_0}, two issues at II=1 in place of a second divider. The three gradient stages are a feed-forward chain, one result per cycle. Grayed: not built.

Figure 4 shows the backward datapath of Section 4.8. Only the $\partial f/\partial a$ path is pipelined, reaching 133.8 MHz. The generalized-numerator gradient remains combinational, and the weight and input gradients are not built.

A.5 BASELINE COMPARISON

The GPU runs fused Triton kernels on an otherwise idle card with TF32 disabled, and at bf16 it accumulates in fp32, which favors it. Its energy is read from runs longer than the power-sampling interval, as a single short sample understates it by 4.3×. The CPU baseline runs on one core of an AMD Ryzen 9 9950X with AVX-512 and cache-resident weights, and its energy is package RAPL energy less an equal idle window, 57 to 64 nJ per ladder over three runs. Table 5 lists every configuration, from which Table 2 takes each baseline's best.

Table 5: RONDO against both baselines at matched precision, one input through the layer. ASAP7 rows are synthesized, not measured. Latency with launch includes the GPU's CUDA-Graph launch; RONDO and the CPU have no launch. Energy is dynamic power for RONDO, board power for the GPU and idle-subtracted package power for the CPU.

			batch-1 latency (µs)		throughput	energy
platform	fmt	clock	kernel	with launch	(M ladders/s)	(nJ/ladder)
RONDO, 28 nm FPGA, 8 lanes	bf16	168.9 MHz	2.20	2.20	29.1	52.2
RONDO, ASAP7 7 nm, 8 lanes	bf16	1.19 GHz	0.311	0.311	206	0.72
RONDO, 28 nm FPGA, 8 lanes	fp32	146.7 MHz	2.53	2.53	25.3	149.1
RONDO, ASAP7 7 nm, 8 lanes	fp32	767 MHz	0.485	0.485	132	1.75
GPU, batch-1 kernel	bf16	2.4 GHz	1.02	6.27	—	—
GPU, throughput kernel	bf16	2.4 GHz	7.78	10.6	112,038	4.43
GPU, batch-1 kernel	fp32	2.4 GHz	0.99	6.24	—	—
GPU, throughput kernel	fp32	2.4 GHz	32.2	35.1	29,880	17.51
CPU, one core	fp32	—	0.36	0.36	177	61.3

Table 6 scales the eight-lane bf16 instance to the GPU's 750 mm^2 at 4 nm with published FPGA-to-ASIC and node factors (Kuon & Rose, 2007; Stillmaker & Baas, 2017), where memory bandwidth rather than area bounds throughput. These factors are about 1.5× more favorable to RONDO than our ASAP7 synthesis and are used nowhere else.

Table 6: RONDO scaled to 750 mm^2 at 4 nm, bf16, against the measured GPU.

memory system	G ladders/s	bound by	vs. GPU
GDDR7, 1.79 TB/s (the GPU's own)	551	bandwidth	4.9×
HBM3, 3.9 TB/s	1,200	bandwidth	10.7×
HBM3E, 8 TB/s	2,339	compute	20.9×

Table 7 and Figure 5 compare RONDO with the Ara vector processor (Perotti et al., 2022), whose RTL runs both forms of the ladder under Verilator, fp32, 256 ladders, d=7. Ara needs fewer cycles per ladder at two lanes and RONDO needs fewer at eight, while only RONDO keeps its scaling efficiency as lanes are added.

Table 7: PULP Ara against RONDO, cycles per ladder. The reformulation column is Ara evaluating the ladder both ways. Scaling is measured from each design's own two-lane point.

	Ara (RVV, fp32)			RONDO (fp32)	
lanes	continuant	direct	reformulation	cycles/ladder	scaling
2	14.46	21.91	1.52×	23.16	—
4	10.82	12.85	1.19×	11.59	2.00×
8	8.12	8.38	1.03×	5.81	3.99×

A.6 ASIC SYNTHESIS

One lane of each datapath is synthesized for ASAP7 at 0.7 V, 25 °C, with regular-threshold cells (Clark et al., 2016), using Yosys and ABC with Booth multipliers and Kogge-Stone adders (Wolf

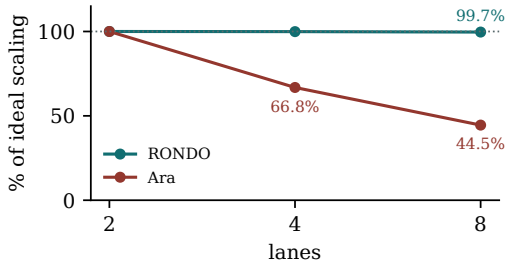


Figure 5: Scaling efficiency of RONDO and Ara on the same operator, each from its own two-lane point.

et al., 2013). OpenSTA reports timing and reports power from a gate-level simulation of the trained Waveform model, in which the netlists reproduce every reference decision (Cherry & Scott, 2025). Memories follow the OpenROAD ASAP7 memory models, charged per read, and eight lanes scale one lane. The figures exclude the clock tree and wire parasitics, and the fp32 pipeline cuts were placed for FPGA multipliers. Table 8 lists the results.

Table 8: One lane of each RONDO datapath synthesized for ASAP7 at the typical corner, and eight lanes derived from it. The KAN lanes use one basis unit at fp32 and five at bf16, as on the FPGA. Memory figures are model estimates.

	bf16	fp32
CoFrNet layer		
clock frequency (MHz)	1,194	767
logic area per lane (μm^2)	6,929	17,783
memory area per lane (μm^2)	1,505	6,966
area, eight lanes (mm^2)	0.067	0.198
single-input latency, eight lanes (µs)	0.311	0.485
throughput, eight lanes (M ladders/s)	206	132
energy per ladder (nJ)	0.72	1.75
of which memory (nJ)	0.063	0.125
Chebyshev KAN layer		
clock frequency (MHz)	1,193	895
logic area per lane (μm^2)	10,253	21,619
memory area per lane (μm^2)	1,720	3,440
area, eight lanes (mm^2)	0.096	0.201
single-input latency, eight lanes (µs)	0.351	0.683
energy per output (nJ)	1.03	2.92
of which memory (nJ)	0.070	0.103

A.7 Comparison with a KAN Accelerator

PolyKAN's code is unpublished, and we reimplement its forward algorithm, sweeping the table size and input tile. Table 9 compares its lookup table with the recurrence it replaces inside the same kernel. The recurrence wins at every batch, as the lookup's input-dependent gathers cannot be coalesced, and PolyKAN leads only at small fp32 batches. SparseKAN's released fp32 layer kernel runs on our layer with a Chebyshev basis in place of its Gram basis, through the same power flow on the FPGA and, with HardFloat units of equal latency in place of its Xilinx floating-point cores, through the same ASAP7 flow, where its outputs match the FPGA netlist bit for bit.

A.8 Extended Results

Design decisions. Table 10 lists each design choice of RONDO against the alternative it replaces.

Table 9: PolyKAN's lookup table against the three-term recurrence it replaces, measured inside one kernel on an RTX 5090, p=40 and 64 outputs. Both arms differ only in the basis evaluation.

	throughput (M outputs/s)			energy (nJ/output)		
batch	table	recurrence	gain	table	recurrence	gain
fp32						
4,096	2,399.8	10,998.0	4.6×	167.9	41.3	4.1×
65,536	2,150.5	13,422.1	6.2×	181.6	33.8	5.4×
524,288	1,904.2	12,201.5	6.4×	200.8	37.4	5.4×
bf16						
4,096	3,083.4	10,993.4	3.6×	147.2	41.4	3.6×
65,536	2,813.6	12,038.5	4.3×	161.9	37.8	4.3×
524,288	3,304.9	13,088.8	4.0×	139.9	34.8	4.0×

Table 10: Each design choice of RONDO against the alternative it replaces, built with the same RTL and tools, after placement and routing, d=7. Arrows read from the alternative to RONDO.

section	RONDO	alternative	alternative → RONDO
4.2 (P1)	one final division	one division per level	throughput per lane: 105× higher
4.2 (P2)	first-column scan	full 2×2 matrix scan	LUT: 29,728 → 2,175; DSP: 16 → 2
4.3 (P3)	five-stage multiply-add	combinational	clock: 47.8 → 168.9 MHz
4.3 (P4)	$N \geq L_{\mathrm{fma}}+1$ slots	$N = L_{\mathrm{fma}}$ slots	results: silently wrong → correct
4.4 (P5)	on-chip coefficients	shared input port	multiplexer area: 63% → none
4.5 (P6, P7)	parallel multiply-adds	chained accumulators	layer clock: 11.6 → 158.9 MHz
4.6 (P8)	data gating	no gating	dynamic power: 37.2% lower
A.11	bf16	fp32	energy per ladder: 2.85× lower

Accuracy and precision. Table 11 gives classification accuracy from the RTL's own outputs, Table 12 gives the bf16 error of the coefficient sum as the input dimension grows, and Table 13 gives the layer at CoFrGeNet dimensions. fp16 is not usable, as the continuants overflow its five-bit exponent.

Table 11: Classification accuracy computed from the RTL's own outputs on the full test sets.

dataset	p	inputs	cycles/ladder	acc. (RTL, %)	acc. (ref., %)
fp32, four lanes					
Waveform	40	1,500	11.59	85.47	85.47
Credit Card	23	9,000	7.88	79.58	79.58
MAGIC	10	5,706	5.03	78.09	78.09
bf16, eight lanes					
Waveform	40	1,500	5.81	85.20	85.47
Credit Card	23	9,000	3.95	79.58	79.58
MAGIC	10	5,706	2.53	78.41	78.09

Second basis, coefficient generator and lanes. Table 14 compares the scan on both bases, Figure 6 shows the coefficient generator with chained accumulators, and Figure 7 plots the lane sweep.

A.9 Experimental setup

A.10 Per-module synthesis

Table 16 reports Yosys results for the modules discussed in the paper.

Table 12: Relative error of a p-term inner product accumulated in bf16, against an fp64 reference.

p	median	p99
10 (MAGIC)	5.6×10^{-3}	2.1×10^{-1}
23 (Credit Card)	6.4×10^{-3}	3.0×10^{-1}
40 (Waveform)	1.0×10^{-2}	3.0×10^{-1}
256	2.7×10^{-2}	1.2×10^{0}
768 (CoFrGeNet)	4.7×10^{-2}	2.1×10^{0}

Table 13: The layer at CoFrGeNet dimensions, 64 samples, bf16 operands throughout.

coefficient accumulator	max. rel. error	agreement (%)
bf16 (as built)	7.58×10^{6}	45.31
fp32	1.38×10^{6}	85.94

A.11 VERIFICATION

Table 17 compares RONDO with a 64-bit floating-point reference. *Agreement* is the fraction of inputs on which the RTL and the reference make the same decision, reported beside the maximum relative error, as the two fail in different ways. At fp32, RONDO agrees on all 16,206 test inputs, including the 19 to 26% that contain a ladder near the pole. At bf16, agreement is 96.9 to 98.3%, and changed decisions concentrate on near-pole inputs. All results are reproduced under Verilator and Icarus.

Table 14: The scan evaluating each basis, d=8, eight concurrent evaluations, post-route on the Genesys 2. Power is SAIF-annotated at 125 MHz with 77% of nets matched; energy per evaluation is dynamic power per cycle times cycles per evaluation.

	continued fraction ($b = +1$)	Chebyshev ($b = -1$)
max. rel. error vs. reference	—	1.4×10^{-6}
$F_{\max}$	163.8 MHz	163.8 MHz
area	2,521 LUT, 2 DSP	2,521 LUT, 2 DSP
latency, batch of eight	384.6 ns	384.6 ns
cycles per evaluation	7.875	7.875
throughput	20.8 M/s	20.8 M/s
dynamic power	0.078 W	0.079 W
energy per evaluation	4.91 nJ	4.98 nJ

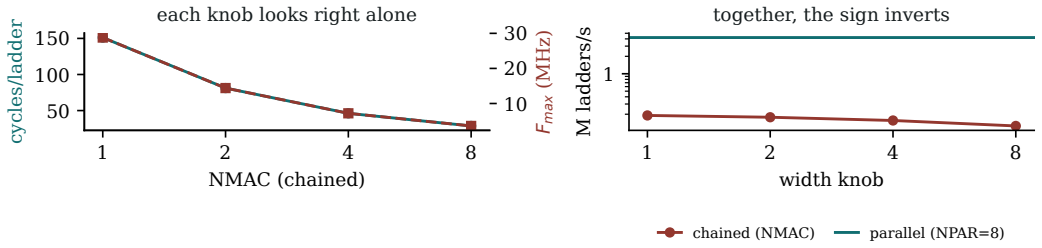


Figure 6: Widening the coefficient generator with NMAC chained accumulators: cycles and clock frequency (left), and their product, throughput (right).

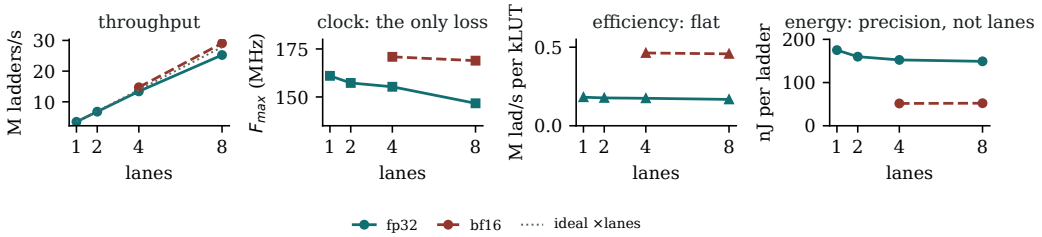


Figure 7: Lane scaling, post-route on the Genesys 2: throughput, clock, throughput per LUT and energy per ladder. The first three panels use one 5.5 ns target; energy comes from the activity-based sweep and does not depend on the clock.

Table 15: Tools, target device, and defaults.

Item	Value
RTL simulation	Icarus 13.0 and Verilator 5.050, cocotb 2.0.1
Synthesis / implementation	Yosys 0.66 (`synth_xilinx -family xc7`); Vivado 2025.2, out of context
Target device	Genesys 2, Kintex-7 XC7K325T (`xc7k325tffg900-2`): 203,800 LUT, 840 DSP48E1, 445 BRAM, 28 nm
FP arithmetic	Berkeley HardFloat, pipelined into five stages in this work
GPU baseline	RTX 5090, PyTorch 2.12 / CUDA 13, fused Triton kernel, measured at fp32 (IEEE, TF32 disabled) and bf16 on an idle card
Systolic baselines	SCALE-Sim v2 (Samajdar et al., 2020); Gemmini v3.1 RTL (Genc et al., 2021), bf16
Models	CoFrNets trained on Waveform (p=40), Credit Card (p=23), MAGIC (p=10); L=64, d=7
Numeric format	IEEE fp32 at any p; bf16 as evaluated at $p \leq 40$, needing an fp32 accumulator by p=768. **fp16 is not usable** at any p: the continuants overflow its five-bit exponent at d=7
Pole guard	clamp $\lvert K_0 \rvert \geq \varepsilon$, $\varepsilon = 10^{-2}$

Table 16: Yosys synthesis, fp32, d=7. Percentages are of the Genesys 2's 203,800 LUTs.

Module	LUT	DSP48E1	FF	% LUT
forward, pipelined				
`fp_fma_pipe`	1,993	2	417	1.0%
`ladder_engine_inter` (fp32)	9,695	8	4,622	4.8%
`ladder_engine_inter` (bf16)	3,786	4	2,326	1.9%
`fused_coproc_pipe` (bf16, 8 lanes)[†]	62,335	0	—	30.6%
forward, combinational (superseded)				
`fp_fma`	5,730	2	0	2.8%
`ladder_scan_iter`	29,728	16	1,071	14.6%
backward, combinational				
`kscan`	2,337	2	478	1.1%
`grad_a_unit`	41,832	42	0	20.5%
`grad_gen_unit`	111,496	112	0	**54.7%**
`train_coproc` (L=4)	54,207	46	2,160	26.6%

[†]post-route (Vivado), not Yosys; the others are Yosys estimates and run consistently high against place-and-route.

Table 17: Decision agreement with the fp64 reference. Trained models are evaluated on their full test sets, and the last column gives the share of test inputs with at least one ladder near the pole, $\lvert K_0 \rvert < \varepsilon$. The CoFrGeNet-size row (L=p=768) uses 64 samples of synthetic coefficients. [a]Median, not maximum. [b]bf16 operands with the coefficient sum carried in fp32.

			fp32		bf16		near
model	p	inputs	max. rel. err	agreement (%)	max. rel. err	agreement (%)	pole (%)
MAGIC	10	5,706	5.1×10^{-2}	100.00	2.4×10^{3}	98.25	23.6
Credit Card	23	9,000	7.7×10^{-2}	100.00	8.1×10^{3}	97.62	19.2
Waveform	40	1,500	7.5×10^{-3}	100.00	5.0×10^{2}	96.93	26.5
CoFrGeNet size[b]	768	64	4.1×10^{-7} [a]	—	1.38×10^{6}	85.94	—