CHEBYUNIT: ENERGY-EFFICIENT FPGA LOW COMPUTATION COMPLEXITY ARTIFICIAL INTELLIGENT ACCELERATOR

Anonymous authors
Paper under double-blind review

ABSTRACT

Multi-Layer Perceptrons (MLPs) achieve high accuracy but are hindered by a large number of parameters, leading to significant memory and power consumption. While Kolmogorov-Arnold Networks (KANs) address this by using learnable functions instead of weight matrices, their B-spline implementations are complicated in hardware designs. To overcome this limitation, we propose a novel hardware framework for Chebyshev-KANs, leveraging the recursive properties and numerical stability of Chebyshev polynomials. Our core component, the ChebyUnit, efficiently generates polynomial bases and reuses coefficients from on-chip storage to perform lightweight inner product operations in a streaming fashion. This approach significantly reduces external memory access (DDR traffic) and resource utilization while maintaining high throughput. Our Verilog implementation on a Xilinx ZCU102 FPGA demonstrates over 90% reductions in LUT, FF, and DSP utilization compared to a baseline high-level synthesis (HLS) design, all while preserving excellent approximation accuracy. These findings confirm the practical efficiency of Chebyshev-KANs, positioning them as a promising solution for interpretable and energy-efficient neural networks, particularly in resource-constrained edge AI applications.

1 INTRODUCTION

Despite the remarkable advances driven by deep learning models in fields such as image classification, speech recognition, and natural language processing, their increasing size and complexity, as exemplified by models with hundreds of billions of parameters[6], have led to a significant escalation in computational and memory demands. Traditional Multi-Layer Perceptrons (MLPs), a foundational neural network architecture, achieve high accuracy at the cost of massive parameter counts. This characteristic makes them particularly resource-intensive, requiring extensive memory access and consuming considerable energy, thereby limiting their viability for deployment in resource-constrained environments such as mobile and edge devices. In response, Kolmogorov-Arnold Networks (KANs)[7][9] were introduced to replace the static weight matrices of MLPs with learnable functional bases. This innovative approach not only drastically reduces the number of parameters while maintaining comparable accuracy but also enhances model interpretability due to the clear mathematical nature of its bases. Nevertheless, standard KAN implementations, often relying on B-spline functions, suffer from complex recursive operations that are challenging to efficiently map onto hardware. To overcome this, we propose an FPGA-based accelerator for a Chebyshev-KAN, which takes advantage of the numerical stability and recursive simplicity of Chebyshev polynomials[13]. Our design not only inherits the parameter efficiency and interpretability of KANs but also incorporates weight reuse strategies to minimize memory traffic and resource utilization, including Look-Up Tables (LUTs) and Digital Signal Processors (DSPs). Implemented in Verilog, our accelerator for Chebyshev-KAN demonstrates substantial performance and efficiency gains, making it a highly promising solution for energy-efficient edge AI applications.

2 RELATED WORKS

2.1 KOLMOGOROV ARNOLD NETWORKS

In this work, we adopt the Kolmogorov-Arnold representation[9] , which states that any continuous multivariate function can be written as

$$f(x) = f(x_1, \ldots, x_n) = \sum_{q=1}^{2n+1} \Phi_q \left(\sum_{p=1}^{n} \phi_{q,p}(x_p) \right) \tag{1}$$

From equation (1), $\phi_{q,p}$ and Φ_q are univariate continuous functions. The theorem provides a constructive scheme for expressing a complicated function as a combination of one-dimensional functions. In this representation, the high-dimensional functions can be controlled by analyzing their behavior through single-variable components. For its core, the result states that a function of many variables can be written as a finite sum of terms, each depending on the interpretation of control points . This decomposition is broadly useful in mathematics and applied science because it streamlines both theoretical analysis and practical computation for multivariate models. It shows that continuous multivariate functions are not more sophisticated than univariate functions since they can be composed of one-variable functions. Moreover, we have turned this mathematics theorem into hardware architecture in the following section.

For a layer l with input width n_l and output width n_{l+1}, KAN replaces edge weights with univariate edge functions. The j-th output coordinate is the sum of those edge activations. In the matrix form, the "product" denotes applying each univariate function to the corresponding input and summation along the rows. Then, for layer-L KAN with layer widths $(n_0, \ldots, n_L)$ and a scalar output $n_L = 1$, the network can be written as the following sum of the edge function (3).

$$x_{l+1,j} = \sum_{i=1}^{n_l} \tilde{x}_{l,j,i} = \sum_{i=1}^{n_l} \phi_{l,j,i}(x_{l,i}), \quad j = 1, \cdots, n_{l+1} \tag{3}$$

In matrix form, this represents as follows

$$\mathbf{x}_{l+1} = \underbrace{\begin{pmatrix} \phi_{l,1,1}(\cdot) & \phi_{l,1,2}(\cdot) & \cdots & \phi_{l,1,n_l}(\cdot) \\ \phi_{l,2,1}(\cdot) & \phi_{l,2,2}(\cdot) & \cdots & \phi_{l,2,n_l}(\cdot) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_{l,n_{l+1},1}(\cdot) & \phi_{l,n_{l+1},2}(\cdot) & \cdots & \phi_{l,n_{l+1},n_l}(\cdot) \end{pmatrix}}_{\Phi_l} \mathbf{x}_l \tag{4}$$

$$f(\mathbf{x}) = \sum_{i_{L-1}=1}^{n_{L-1}} \phi_{L-1,i_L,i_{L-1}} \left(\sum_{i_{L-2}=1}^{n_{L-2}} \cdots \left(\sum_{i_2=1}^{n_2} \phi_{2,i_3,i_2} \left(\sum_{i_1=1}^{n_1} \phi_{1,i_2,i_1} \left(\sum_{i_0=1}^{n_0} \phi_{0,i_1,i_0}(x_{i_0}) \right) \right) \right) \right) \tag{5}$$

KANs were proposed to address the limitations of MLPs in parameter efficiency and interpretability. Instead of using fixed node activations, KANs put learnable univariate functions which are typically B-spline parameterized on edges. Therefore, each weight becomes a small function that adapts to the data, increasing the flexibility of the model. This design is inspired by the Kolmogorov-Arnold representation, which turns edge parameters into learnable functions and improves adaptability compared to fixed activations in MLPs.

From the perspective of hardware design, previous studies on KANs have emphasized their compactness. KANs can achieve an accuracy comparable to that of MLPs while using fewer parameters, suggesting potential savings in storage and data movement. However, straightforward implementations of the B-spline evaluation are computationally intensive and consume lots of hardware resources[15]. Related hardware-focused research on KANs further highlights this challenge: recursive and nonlinear B-spline evaluations are implemented complicatedly from the hardware point of view and significantly retard the acceleration. It shows that direct LUT-mapping of B-splines incurs substantial LUT/MUX /decoder overhead on edge devices; hardware-aware quantization is required to be cost-efficiency.

Finally, early FPGA studies comparing KAN and MLP indicate that while KANs can be more parameter-efficient, straightforward KAN accelerators may consume significantly more LUT and DSP resources than MLPs if basis evaluation and memory layout are not optimized. This highlights the importance of discreet KAN-hardware design.

2.2 CHEBYSHEV POLYNOMIAL

The Chebyshev polynomial basis can be defined by the polynomial form[13] :

$$\begin{aligned} T_0(x) &= 1 \\ T_1(x) &= x \\ T_2(x) &= 2x^2 - 1 \\ T_3(x) &= 4x^3 - 3x \\ T_n(x) &= 2xT_{n-1}(x) - T_{n-2}(x) \quad \text{for } n \geq 2 \end{aligned} \tag{6}$$

The Chebyshev Kolmogorov–Arnold Network (Chebyshev-KAN) is a novel architecture designed to improve the efficiency and accuracy of nonlinear function approximation. By combining Kolmogorov-Arnold theorem with Chebyshev polynomials, Chebyshev-KAN offers advantages over traditional MLPs. While MLPs require many parameters and often suffer from inactive neurons, Chebyshev-KAN assigns learnable activation functions to edges, allowing the model to achieve lower parameter counts while maintaining or even improving accuracy.

Since edge functions are explicit and one-dimensional, they can be visualized and analyzed directly. This gives Chebyshev-KAN better interpretability than MLPs, especially in scientific and engineering applications to understand learned transformations. In terms of numerical stability and approximation accuracy, Chebyshev polynomials are renowned for their fast convergence and stable approximation. By integrating these polynomials into KANs, the network can achieve higher approximation accuracy, which, in turn, directly optimizes hardware deployment.

Besides, the network can be expressed as

$$y_{bo} = \sum_{i=1}^{\text{input\_dim}} \sum_{j=0}^{\text{degree}} T_{bij} \cdot C_{ioj} \tag{7}$$

We compute the layer output y by contracting the Chebyshev-basis tensor T with the learnable coefficient tensor C via Einstein summation (einsum), where i indicates the input dimensions and j indexes the degree of the Chebyshev polynomials. This operation combines the polynomial bases with the learned coefficient to produce the final output layer y. In short, it copies numerous Chebyshev- basis along the input tensor and then sums over the degree index to produce the output so that each input-multiply weight can be retrieved from the Chebyshev curve. Prior hardware efforts mainly target B-spline KAN and hardware-aware quantization, whereas our design builds a Chebyshev-basis KAN on FPGA with lightweight tensor generation and on-chip reuse, avoiding large LUT/MUX fabrics. Additionally, we have tried to convert the theory into hardware architecture and will elaborate the hardware system in the next section.

2.3 FPGA ACCELERATORS DESIGN

Early FPGA accelerators for DNN focus on CONV+FC (convolution and full-connected) layers, because they dominate both parameter counts and arithmetic. A canonical breakdown on VGG-11 shows that CONV+FC contribute 99 percentage of weights and operations; similarly, RNNs are largely FC-dominated, so most systems target these two kernels [3].

A typical accelerator system connects a host CPU and an FPGA: the host issues commands over PCIe/Ethernet/AXI, and each side keeps its own external DDR; SoC platforms (Xilinx Zynq) co-locate host and FPGA in one package. Designs usually place the NN accelerator on the FPGA and orchestrate it from the host.

The central bottleneck is memory. Although FPGAs have registers/BRAM on-chip, capacity is small relative to modern models (common networks use 100–1000 MB; large FPGAs provide 50

MB on-chip). Consequently, designs must stream from external DDR, and DDR bandwidth and energy dominate overall performance/efficiency.

On the compute side, FPGAs expose hundreds to thousands of DSPs, reaching up to 10 TFLOP/s on high-end parts; by contrast, low-end devices such as Xilinx XC7Z020 deliver only 20 GFLOP/s, which constrains real-time workloads.

For non-convolutional models such as KAN/Chebyshev-KAN, we adopt fixed-point quantization to maximize DSP utilization (f, P, η) , schedule on-chip buffers to minimize DDR traffic, and organize computation around recurrence-based basis generation plus einsum-style contraction to keep W low and the design on the compute, not memory.(throughput IPS = (f * P * η) / W, f for working frequency of the computation units, P for number of computation units in the hardware design, η for utilization of the computation and W for Workload for each inference)[4]

3 Hardware Implementation Approach

In this work, the hardware implementation was carried out using Verilog hardware description language (HDL) and Xilinx Vivado de- sign suite. Verilog provides precise control over datapaths, timing, and logic, making it well-suited for FPGA deployment. Vivado offers an integrated flow for simulation, synthesis, and implementation, along with timing analysis, resource utilization, and power reporting. Our designed modules are parameterizable, where the input and output dimensions and hidden layer sizes can be flexibly adjusted at the HDL level, so a single register-transfer level (RTL) design can support different network architectures with minimal modification. The accelerator was implemented from ChebyUnit inner-product operations up to full network interconnections and deployed on a Xilinx MPSoC UltraScale+ ZCU102 platform with a 100 MHz clock frequency and fixed-point data type which is configurable bit-width.

3.1 Activation Unit Design

The Activation Unit is the fundamental hardware block in a Chebyshev-KAN. It computes the output of a single neuron by generating Chebyshev basis values, multiplying them with stored coefficients (controlling points), and accumulating the corresponding results. Conceptually, the unit transforms one input value into its neuron-level contribution, which can later be combined with other neurons to form the output of the network layer. To make this design practical for FPGA, the Activation Unit is built as a parameterizable and reusable module. Key parameters such as input dimension, polynomial degree, data bit width, and fixed point precision can be adjusted at the HDL level. With these configurable options and parameters, the same hardware block can support different network sizes and accuracy requirements while balancing resource utilization and performance. Since this Activation Unit is specifically constructed with Chebyshev polynomials as its basis functions, it will hereafter be referred to as the ChebyUnit and this name will be used consistently throughout the remainder of this work. Figure 1 illustrates the internal datapath of the ChebyUnits. It is mainly composed of three main components: Chebyshev Tensor Function Generator, Control Point Line Buffer and Inner Product Computation.

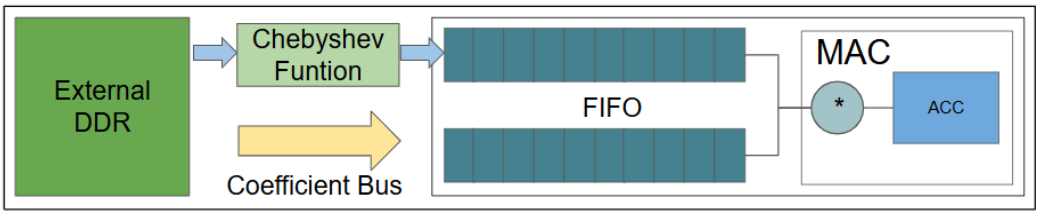

Figure 1: ChebyUnit, single PE structure

3.1.1 Chebyshev Tensor Function Evaluation

The first stage of the ChebyUnits is the Chebyshev Tensor Function Generator, which produces the polynomial basis values needed for neuron computation. Given an input data x, the module

computes all Chebyshev terms $T0(x)$, $T1(x)$,...,$Td(x)$ based on the recursive relation mentioned above. In hardware, this recursion is translated into a streaming datapath: The input data x is first registered and its scaled value ($2x$) is obtained by a one-bit left shift, which is stored for reuse in subsequent computations. This approach eliminates the need to use DSP slices for repeated multiplications by 2, thereby saving valuable multiplier resources for other operations. Multiplications in Chebyshev polynomials are mapped to FPGA DSP slices and the simple shift is efficiently handled by basic logic resources.

Once computed, the tensor values are passed one by one, starting from $T0(x)$ up to $Td(x)$. This streaming output allows the next modules, such as the control point buffer and inner-product unit—to begin processing immediately, improving overall throughput without waiting for all terms to finish.

All computations are performed in fixed-point arithmetic, with parameterizable bit-width and fractional precision. This flexibility allows the designer to adjust numerical precision according to resource constraints: a larger bit-width increases accuracy but consumes more LUTs and DSPs, while smaller sizes improve area and power efficiency. To ensure numerical stability, custom arithmetic functions handle saturation and overflow, preventing computational error under limited precision. By organizing the computation as a datapath—input scaling, iteratively updating, temporary storage, and streaming output—the Chebyshev tensor function evaluation module delivers a compact and efficient hardware block. As a result, the module is compact and resource-efficient, making it a suitable basis for the subsequent stages of the ChebyUnits.

3.1.2 CONTROL POINT STORAGE

The Control Point Line Buffer serves as an intermediate buffer that manages trained parameters during inference. After the model parameters are transferred from external DDR memory to the FPGA through the Direct Memory Access(DMA) interface, they are temporarily stored in this module before computation begins. The storage depth is defined as $degree + 1$, which corresponds to the number of coefficients required to evaluate the Chebyshev polynomial basis functions. During computation, these stored control points are sequentially read out and multiplied with the corresponding polynomial values streamed from the evaluation layer, forming the Multiply Accumulate(MAC) operations that drive the network.

From a hardware implementation perspective, the Control Point Line Buffer is realized using LUT-based distributed RAM. Since control points are repeatedly accessed during inference, pre-storing them locally avoids repeated DDR fetches, thereby reducing data transfer overhead between memory and FPGA fabric,enhancing the data reuse. This design ensures fast, low-latency access to coefficients and minimizes stalling in the computation pipeline.

3.2 CHEBYSHEV KAN NETWORK

Based on the previously introduced ChebyUnit, the Chebyshev KAN Network organizes multiple units in parallel according to the hidden number parameter. This parameter directly determines how many ChebyUnits are instantiated within the network. According to the Kolmogorov–Arnold representation theorem, the maximum required number of hidden units is bounded by twice the input dimension plus one. However, in practice, it is not always necessary to allocate the full number of units. Instead, the hidden number can be flexibly determined based on the characteristics of the dataset, aiming to achieve good performance with a minimal number of unit. As illustrated in Figure 2, our design enables parallel evaluation of inner product computations. Such structure emphasizes parallel computation, where multiple polynomial expansions are processed simultaneously, thereby improving the throughput and scalability of the overall system.

The control point parameters, trained offline and stored in DDR, are transferred to the FPGA through the DMA engine via the Advanced eXtensible Interface (AXI) Stream (AXIS) interface. These parameters are then distributed by a custom-designed bus structure, which delivers the incoming data to the LUTbased storage inside each ChebyUnit. Using a finite-state machine (FSM) controls this process, determining both the data flow path and whether a given ChebyUnit should accept the incoming data. This mechanism ensures that each ChebyUnit receives the correct set of control points while maintaining efficient data movement across the parallel structure.

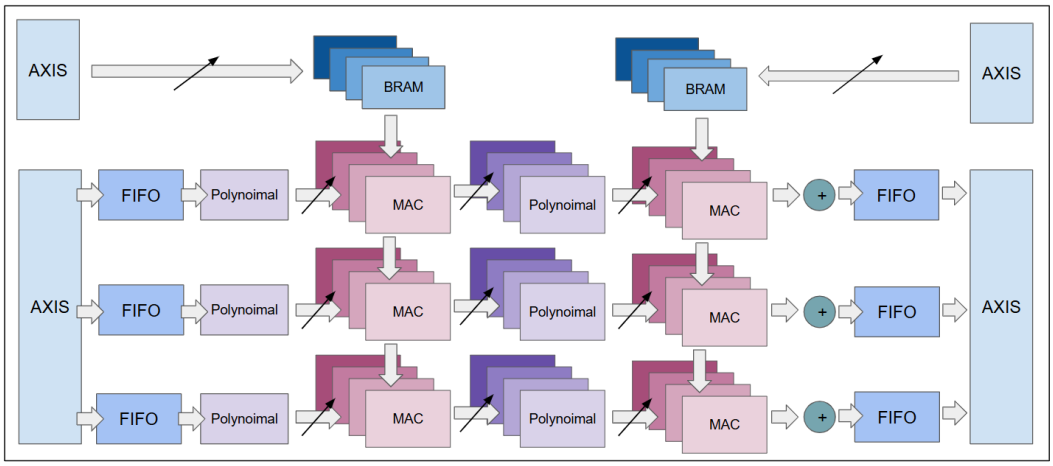

Figure 2: Chebyshev Network Architeture

Once the control points are fully loaded and stored in the ChebyUnit, the system proceeds with the inference phase by feeding input data into the network. To ensure that all variables are applied synchronously to the model, and given the limitation on the number of data bits transferred through the DMA interface, the serial input stream from DMA is first buffered in a linebuffer. This linebuffer is implemented using BRAM on the FPGA, temporarily holding the input stream before releasing it in parallel form. In this way, the design guarantees synchronized delivery of inputs to the network while efficiently overcoming the bitwidth constraint of the DMA interface. As illustrated in Figure 3, this process is part of the overall processing-system/programmable-logic (PS/PL) integration, where external DDR provides data, DMA engines manage streaming through AXI interfaces, and the custom hardware intellectual-property (IP) performs the Chebyshev-KAN computation. The results are then streamed back to DDR via a symmetric DMA path. This system-level setup provides the foundation for the input and output buffers described in the next section.

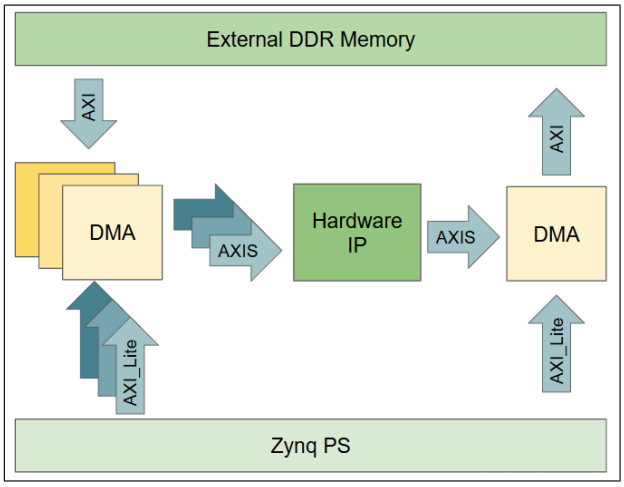

Figure 3: PS and PL configuration

4 EXPERIMENTS AND IMPLEMENTATION RESULTS

4.1 SETUPS

Using the open-source project provided by SynodicMonth on GitHub[14], we obtain trained parameters of the Chebyshev KAN model, along with its curve fitting results and a performance comparison against a conventional MLP neural network in terms of convergence speed. This setup enables us to compare the numerical results generated in PyTorch with those obtained from hardware simulation, ensuring that our architecture reproduces the intended computations. The overall workflow is described as follows: In the PyTorch part, the required trained coefficients (C_{ioj}, as mentioned in 2.2) are extracted from the model. These coefficients are then applied to the hardware part, where the computations are performed. Validation is also conducted using Vivado through simulation, synthesis, and implementation. Finally, the design is deployed on the Xilinx MPSoC UltraScale+ ZCU102 platform, where the actual power consumption and resource utilization are measured[1]. This process enables us to evaluate both the feasibility of hardware implementation and the consistency between software-based and hardware-based computations.

4.2 RESULTS

For the following experiment, the input data is defined within the range of [-1, 1]. The test result is visualized using Excel for clarity.

For the target function :

$$f(x, y) = e^{(x^2+y^2)} \cdot \sin\big(2\pi(x^2 + y^2)\big)$$

It is carried out with an input dimensionality of 2, a polynomial degree of 8, 4 hidden units, and coefficients represented in the Q16.16 format (32 bits). Note that due to the limited number of test data points, the plotted results may appear less smooth when visualized in Excel. Nevertheless, even with the current sampling density, numerical verification confirms that the computed values are correct and consistent with the expected results.

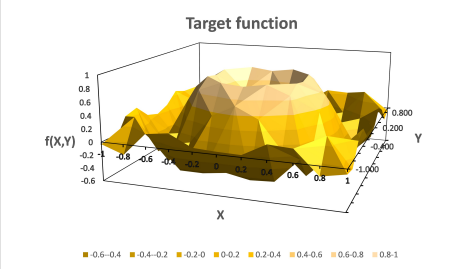

Figure 4: Target Function Implementation

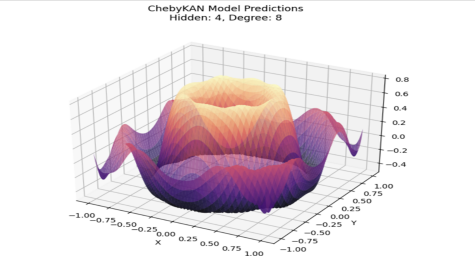

Figure 5: Training Result

From the experiment, it verifies the accuracy of our hardware implementation. The computations perform in hardware are consistent with the expected outputs, which demonstrates that the design is reliable and reproduces the intended numerical behavior. On the other hand, the dominant factor influencing precision is the fixed-point format to influence the precision of the hardware results is the fixed-point format, namely the number of bits used to represent the data. The bit-width directly determines the resolution and dynamic range of numerical representation, thereby imposing a fundamental trade-off between resource utilization and computational precision. A higher number of bits improves the approximation capability of target functions, but at the cost of increased hardware resources such as logic units, registers, and power consumption. In contrast, reducing the bit-width lowers resource requirements but limits the system's ability to precisely capture numerical variations. These observations highlight the importance of carefully selecting an appropriate fixed-point format when implementing learning models in hardware, as it significantly impacts both efficiency and performance.

4.3 RESOURCE AND POWER ANALYSIS

To better highlight the efficiency of our design, we report both the hardware resource utilization and the percentage reduction compared to the HLS baseline[15]. It is worth mentioning that the architecture of the HLS implementation differs from ours, as it is based on B-spline to construct each learnable activation functions of the KAN[9]. Table 1 presents the actual resource utilization for each case, while Table 2 reports the reduction percentage, calculated as : [1] : The utilization of resources in HLS implementation[15].

In Table 2, our method achieves significant reductions in LUTs, FFs, and DSPs, with savings exceeding 90% in datasets such as Dry Bean and Mushroom. The Chebyshev-based design (abbreviated as Chebyshev in Table 1), due to its simpler function expression compared to the B-spline based design (abbreviated as B-spline in Table 1), requires fewer hardware resources, especially DSPs, and thus holds a significant advantage in hardware computation. Furthermore, as shown in Table 1, our design consistently achieves a latency of 13 cycles and demonstrates better stability in power consumption. Even for larger model sizes, our approach exhibits lower power usage. To sum up, these results indicate that our method delivers better energy efficiency and data transfer performance.

Dataset	**Model size**	**Type**	**Freq (MHz)**	**Hardware Resources**			**Power (W)**	**Latency**	
				LUTs	**FFs**	**DSPs**		**Cycle**	**Time (ns)**
Moons	2,2,1	B-spline	100	17877	8622	120	0.717	128	1280
		Chebyshev		**9888**	12150	**40**	3.034	**13**	**130**
Wine	13,4,3	B-spline		146843	74741	960	1.349	688	6880
		Chebyshev		**30154**	**22104**	**324**	3.293	**13**	**130**
Dry Bean	16,2,7	B-spline		1677558	734544	9111	14.802	1896	18960
		Chebyshev		**27359**	**25198**	**256**	3.271	**13**	**130**
Mushroom	8,24,2	B-spline		3112275	1337291	16299	-	3434	34340
		Chebyshev		**80393**	**38985**	**1088**	3.809	**13**	**130**

Table 1: Actual Resource Utilization Under Different Datasets and Implementations types

- : Power consumption is not included, as the hardware requirements surpass the FPGA board capacity described in [15].

Dataset	**LUTs Reduction**	**FFs Reduction**	**DSPs Reduction**	**Latency Reduction**
Moons	44.69%	-	66.67%	89.84%
Wine	79.47%	70.43%	66.25%	98.11%
Dry Bean	98.37%	96.57%	97.19%	99.31%
Mushroom	97.42%	97.08%	93.32%	99.62%

Table 2: Utilization and Latency Reduction Result

- : In a small-scale model, the synthesizer often implements temporary buffers directly with flip-flops instead of other larger memory blocks.

For the case of the (2,2,1) model size, the pie chart (Figure 6) presents the distribution of power consumption between different components[10]. In the pie chart, values without the label "static" correspond to dynamic power consumption. It can be observed that the Processing System (PS Dynamic) dominates the overall power consumption, representing the primary source of energy dissipation. In contrast, the contributions from the programmable logic domain, including signals, clocks, logic elements, Block Random Access Memory (BRAM), and DSP blocks, are comparatively minor. This observation suggests that the power of internal signal activities within the programmable logic is comparatively small. Despite their small contribution to power, the Programmable Logic (PL) components efficiently handle complex computations, highlighting their effectiveness and energy efficiency.

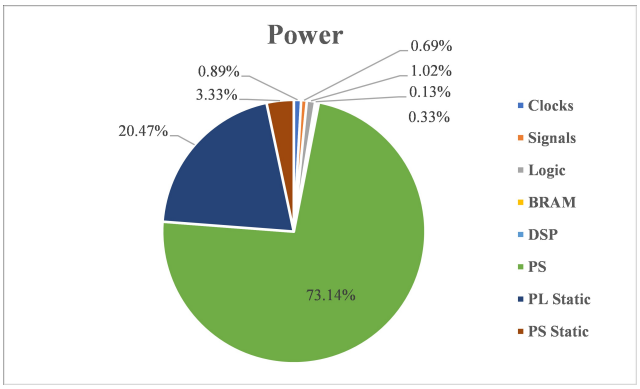

Figure 6: Power Consumption Distribution of the (2,2,1) Model Size

5 Discussion and conclusion

In this work, we built an FPGA-based accelerator for Chebyshev-Kolmogorov Arnold Networks (Chebyshev-KAN) to deal with the challenge of running models that are parameter-efficient but not always hardware-friendly, especially in energy-constrained settings. Using the recursive structure and numerical stability of Chebyshev polynomials, we proposed the modular ChebyUnit design, which dramatically reduces LUT, FF, and DSP usage while keeping good approximation accuracy[11; 8]. Our experiments on the Xilinx ZCU102 confirm that the design works as expected and delivers substantial energy savings—over 90% compared to a straightforward HLS baseline.

What really stands out is the trade-off between precision and efficiency. Wider fixed-point formats enhance accuracy but increase DSP and BRAM cost, while narrower formats save resources at the cost of fidelity. This suggests that a single precision scheme is suboptimal. Future work should explore adaptive or mixed-precision strategies, where network components use different precisions based on error sensitivity [5; 2], enabling better accuracy–efficiency trade-offs.

Scalability is another important takeaway. The modular ChebyUnit allows the design to scale with input sizes and network depths, but memory bandwidth and on-chip storage become bottlenecks. To address this, we incorporate weight reuse, enabling coefficients to be shared across computations [12]. This reduces redundant memory transfers, reduces resource demand, and maintains throughput without raising hardware cost. Combined with memory scheduling and compression, weight reuse offers a practical path for scaling Chebyshev-KAN accelerators.

Beyond architectural efficiency, the proposed accelerator is highly suitable for edge AI applications. Using functional bases reduces parameters, computation, and memory traffic, leading to lighter hardware. The ChebyUnit further minimizes DSP usage, and local coefficient storage cuts off-chip access. Together, these factors reduce power consumption, making the design attractive for IoT devices, biomedical tasks such as medical image segmentation [4], and real-time sensing systems. In short, with fewer parameters and lower energy use, Chebyshev-KAN is a promising candidate for efficient, interpretable edge AI.

Of course, there are still limitations. Our tests focused on function approximation and small datasets; real-world workloads in vision, speech, or time-series tasks will be more demanding. Precision settings were manually tuned, suggesting the need for systematic quantization strategies. Moreover, evaluation was limited to a single FPGA (ZCU102); broader testing across FPGAs or ASICs would provide a more complete performance picture.

Overall, this work shows that Chebyshev-KAN is not only theoretically elegant but also practically deployable. FPGAs, with their flexibility and energy efficiency, offer a promising pathway for interpretable and resource-efficient neural networks at the edge. We envision AI systems that are accurate, efficient, and transparent, while remaining adaptable to real-world constraints.

REFERENCES

[1] AMD. *ZCU102 Evaluation Board User Guide (UG1182)*. AMD, Santa Clara, CA, revision 1.7 edition, February 2023. URL https://docs.amd.com/v/u/en-US/ug1182-zcu102-eval-bd. Document ID: UG1182.

[2] Jia-Wei Chang, Po-Chun Chen, Wei Hsiao, Yu-Hsin Chen, and Chia-Lin Hsu. Ilmpq: An intra-layer multi-precision deep neural network quantization framework for fpga. In *Proceedings of the 2021 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8. IEEE, 2021. doi: 10.1109/ICCAD51958.2021.9643432.

[3] Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang, and Huazhong Yang. [dl] a survey of FPGA-based neural network inference accelerator. *ACM Transactions on Reconfigurable Technology and Systems*, 9(4):Article 11, December 2017. doi: 10.1145/0000001.0000001. URL https://arxiv.org/abs/1712.08934. arXiv:1712.08934 [cs.AR].

[4] Xuhui Guo, Tanmoy Dam, Rohan Dhamdhere, Gourav Modanwal, and Anant Madabhushi. UNETVL: Enhancing 3d medical image segmentation with Chebyshev KAN powered vision-LSTM. In *Proceedings of the International Conference on Medical Image Computing and Computer-Assisted Intervention (MICCAI)*. Springer, 2023. MICCAI 2023.

[5] Xiangyu Huang, Runnan Zhao, Yiran Chen, and Yu Wang. On-chip hardware-aware quantization for mixed precision neural networks. *arXiv preprint arXiv:2309.01945*, 2024. URL https://arxiv.org/abs/2309.01945.

[6] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling Laws for Neural Language Models, January 2020. URL https://arxiv.org/abs/2001.08361. arXiv:2001.08361.

[7] A. N. Kolmogorov. On the representation of continuous functions of several variables as superpositions of continuous functions of a smaller number of variables. *Doklady Akademii Nauk SSSR*, 108(2):179–182, 1956. in Russian.

[8] Ben Lee, Peter Y. K. Cheung, Wayne Luk, and John D. Villasenor. Hardware implementation trade-offs of polynomial approximations and interpolations. In *IEEE Transactions on Computers*, volume 57, pp. 475–487. IEEE, 2008. doi: 10.1109/TC.2007.70784.

[9] Ziming Liu, Yixuan Wang, Sachin Vaidya, Fabian Ruehle, James Halverson, Marin Soljačić, Thomas Y. Hou, and Max Tegmark. KAN: Kolmogorov–Arnold networks, April 2024. URL https://arxiv.org/abs/2404.19756.

[10] Fuad Mammadzada. Design of a kolmogorov-arnold network hardware accelerator. Master's thesis, Lund University, Department of Electrical and Information Technology, June 2025. Supervisors: Dr. Joachim Rodrigues, Masoud Nouripayam, Kristoffer Westring. Examiner: Dr. Pietro Andreani.

[11] Elie Nicolas, Rafic Ayoubi, and Samir Berjaoui. Chebyshev approximation technique: Analysis and applications. *International Journal of Electrical and Computer Engineering*, 2024. URL https://www.bohrium.com/paper-details/chebyshev-approximation-technique-analysis-and-applications/1030325950128062485-2679.

[12] Sai Krishna Oppu, Anil Kumar, and Venkatesh Rao. Fpga-orthopoly: A hardware implementation of orthogonal polynomials. In *2024 International Conference on Field-Programmable Logic and Applications (FPL)*, pp. 421–428. IEEE, 2024. URL https://www.bohrium.com/paper-details/fpga-orthopoly-a-hardware-implementation-of-orthogonal-polynomials/812788746691805185-3886.

[13] S. S. Sidharth, R. Gokul, K. P. Anas, and A. R. Keerthana. Chebyshev Polynomial-Based Kolmogorov–Arnold Networks: An efficient architecture for nonlinear function approximation, June 2024. URL https://arxiv.org/abs/2406.07200. arXiv:2406.07200.

[14] SynodicMonth. Chebykan: Kolmogorov-arnold networks using chebyshev polynomials. https://github.com/SynodicMonth/ChebyKAN. GitHub repository, accessed: 2025-09-17.

[15] Van Duy Tran, Tran Xuan Hieu Le, Thi Diem Tran, Hoai Luan Pham, Vu Trung Duong Le, Tuan Hai Vu, Van Tinh Nguyen, and Yasuhiko Nakashima. Exploring the Limitations of Kolmogorov–Arnold Networks in Classification: Insights to software training and hardware implementation, July 2024. URL https://arxiv.org/abs/2407.17790. arXiv:2407.17790.