The Alexander Kofkin
Faculty of Engineering
Bar-Ilan University

Faculty of Engineering
Edge-AI HW Acceleration Extension

הרחבת מאיצי חמרה לפלטפורמת בינה מלאכותית למוצרי קצה

Main Chain: Hardware Design

Project Number: 315

Supervisor: Dr. Yehuda Kra

Academic Advisor: Dr. Leonid Yavits

Submitted By: Harel Markel

Submitted By: Eyal Yemini

Abstract

This project presents a novel fixed-point hardware accelerator for Softmax computation, addressing the computational challenges of exponential and division operations in embedded and edge devices. The proposed architecture employs a LUT-based exponentiation pathway and integer normalization, implemented as a synthesis-ready SystemVerilog module. The design comprises three primary stages: accumulation, scaled reciprocal computation, and vector output generation, delivering configurable-precision probability vectors. Integration with a SoC-oriented workflow facilitates both software validation and hardware acceleration paths. The implementation incorporates automated validation tools and bit-level scaling optimizations to minimize error while adhering to on-chip memory constraints. Empirical results demonstrate functional correctness and reproducibility across operational modes, establishing this accelerator as a viable solution for Edge-AI applications.

Acknowledgments

The authors extend their gratitude to Dr. Yehuda (Udi) Kra for his invaluable guidance, comprehensive feedback, and provision of the `XBOX` development environment that facilitated efficient prototyping. We acknowledge Prof. Leonid Yavits for his academic oversight, methodological rigor, and continued support throughout the research process.

Contents

1 Introduction and Problem Statement

1.1 Motivation

The Softmax transformation is fundamental in deep neural networks for normalizing logits into probability distributions across various applications including classification, attention mechanisms, and routing. Implementation of floating-point exponential and division operations on embedded platforms presents significant challenges in terms of latency, power consumption, and silicon area utilization, frequently constituting the primary bottleneck in inference pipelines. This project proposes a fixed-point architecture that optimizes these operations through efficient table lookups and integer arithmetic, achieving enhanced throughput-to-power ratios while maintaining requisite accuracy levels.

1.2 Operational Context

This project targets resource-constrained edge systems where on-chip memory, logic fabric, and clock frequency are limited and latency determinism is valued. The accelerator is intended to be integrated as a drop-in primitive within a SoC or accelerator subsystem, with software hooks for validation and benchmarking. Our work is developed and tested within a SoC development environment and an associated toolchain that supports repeatable builds, simulation, and host-driven tests.

1.3 Problem Definition

For an input vector $\mathbf{z} \in \mathbb{R}^N$, the Softmax function is formally defined as:

$$\text{softmax}(z_i) := \frac{e^{z_i}}{\sum_{j=1}^{N} e^{z_j}} \quad \text{for } i = 1, \ldots, N$$

The primary objective is to synthesize a hardware module that computes an approximation $\widehat{\text{softmax}}(\mathbf{z})$ utilizing fixed-point arithmetic while maintaining bounded error margins and satisfying edge-class constraints pertaining to silicon area, operating frequency, processing latency, and power consumption. The architecture may exhibit streaming capabilities, deterministic latency characteristics, and modular interfaces to facilitate integration across diverse neural network models and system architectures.

1.4 Assumptions

The implementation assumes the following operational conditions: (1) input vectors comprise fixed-point scalars in predetermined Q-format notation with known length N; (2) numerical stabilization via $z_{\max}$ reduction is supported through dedicated shift operations; (3) sufficient on-chip memory exists for exponentiation look-up tables and accumulation buffers; and (4) configuration and control mechanisms are accessible through memory-mapped or streaming interfaces.

1.5 Constraints

The design is subject to the following architectural constraints: (1) BRAM utilization may remain within edge-class FPGA and ASIC SRAM specifications; (2) processing time may scale linearly with vector dimensionality; (3) latency may maintain $O(N)$ complexity with minimal constant overhead and deterministic behavior; (4) arithmetic operations may prioritize reciprocal-multiply approaches over iterative division; and (5) power and area efficiency may demonstrate competitive advantage over optimized software implementations.

1.6 Target Platforms

Primary targets are mid-range FPGAs and SoC subsystems used in edge devices, evaluated under typical edge clock rates and resource budgets. The reference flow assumes host-driven tests via the `xbox` environment and a software reference implementation for bit-exact and tolerance-based validation.

1.7 Interfaces and I/O Contract

The module accepts a sequence of N fixed-point inputs and emits N fixed-point probabilities in the same order. Configuration registers control vector length, scaling factors, and mode bits, and status registers expose completion flags and optional error counters.

1.8 Quality Targets and Success Criteria

- **Numerical Precision:** Quantitative error metrics (MSE and maximum absolute error) may remain within configurable bounds relative to float32 reference implementation, as determined by Q-format specifications and LUT resolution parameters.
- **Processing Efficiency:** Processing time scales linearly with vector length N.
- **Temporal Characteristics:** Guaranteed linear-time complexity $O(N)$ with minimal, deterministic overhead.

1.9 Scope and Non-Goals

The project focuses on the Softmax kernel itself, including exponentiation, normalization, and scaling in fixed point. General-purpose neural network layers, full training support, and host deployment tooling beyond what is needed for validation are out of scope.

1.10 Approach Overview

We adopt a sequential architecture with three main phases: accumulation of exponentiated values from look-up tables, reciprocal computation of the sum, and final multiplication for normalized probabilities. Parameterizable Q formats and table sizes allow trading precision for area and memory. A software reference and test harness check functional correctness and quantify error, enabling rapid iteration on design points before hardware prototyping.

2 Literature Review

2.1 Canonical Transformations (Log-Sum-Exp)

A standard stabilization applies $x_{\max}$ subtraction and the log-sum-exp identity to keep exponentials in range and to condition the denominator [1]. This preserves accuracy but still demands exponent and often logarithm blocks or their approximations, which are costly in edge-class hardware. **Downfall:** exp/log units (or their close approximations) remain the dominant contributors to area, power, and timing closure effort. **Implication:** strong numerical fidelity, limited savings unless exp/log are aggressively simplified.

2.2 LUT-Based Exponentiation (Monolithic and Piecewise)

Lookup-based exponentiation replaces transcendental computation with ROM access, with piecewise schemes reducing memory via segmented domains and low-order fits [2, 3]. These methods achieve low error with predictable latency and simple control when combined with $x_{\max}$ subtraction. **Downfall:** ROM footprint still grows with tighter error targets and wider dynamic range, and the design may still realize a reciprocal or normalization path. **Implication:** attractive accuracy-area trade-off when the input range is bounded and latency determinism is required.

2.3 CORDIC / Hyperbolic Exponential

CORDIC-style engines compute exponentials with shift-add iterations and minimal multipliers, making them appealing for certain FPGA/ASIC budgets [4]. However, range-scaling and iteration depth introduce non-trivial latency and control complexity, especially for short vectors and hard real-time budgets. **Downfall:** iterative latency and convergence management complicate fixed-latency streaming. **Implication:** good fit when multipliers are scarce and long, deeply pipelined datapaths are acceptable.

2.4 Stochastic / Bitstream Computation

Stochastic encodings can approximate exponentials and divisions with simple logic and random sources, trading precision for stream length and energy. In practice, achieving low error requires long bitstreams that erode latency and energy advantages for deterministic edge inference. **Downfall:** unfavorable accuracy-latency trade-off for real-time, low-variance workloads. **Implication:** niche utility in extreme area limits, less suited for our deterministic targets.

2.5 Aggressive Shift/Add Approximations

Shift/add heuristics replace e^x with very low-order surrogates and normalize with power-of-two scaling, often guided by leading-one detection [5]. These designs can be extremely compact and energy-efficient for benign input statistics and tolerant applications. **Downfall:** probability distortion increases with wider ranges and stricter accuracy requirements, and the method can be sensitive to input distributions. **Implication:** excellent efficiency when small probability error is acceptable, risky when semantics may closely match reference Softmax.

2.6 Pseudo-Softmax (Base-2 with Shared Exponent/Mantissa)

Pseudo-softmax variants approximate e^x by 2^x and restructure the denominator to amortize a reciprocal with shared scaling, aligning well with quantized (e.g., INT8) pipelines [6]. They keep outputs normalized while reducing the number of expensive operations per vector. **Downfall:** base change and shared-scaling introduce systematic approximation, and FP/PWL control paths add integration complexity for fixed-point SoCs. **Implication:** compelling for uniform INT8 flows; less natural for tightly fixed-point, per-stage-timed kernels.

2.7 Design Space Exploration and Tunability

Recent work emphasizes precision-adjustable architectures that expose LUT granularity, wordlength, and parallelism as knobs to balance error versus cost [3, 4, 1]. This trend highlights the limits of rigid designs with fixed precisions, large dividers, or single-range LUTs. **Downfall:** inflexible architectures fail to serve diverse models and edge constraints without overprovisioning. **Implication:** parameterizable blocks with clear accuracy-area-latency controls are preferred.

2.8 Why We Chose a LUT + Fixed-Point Normalization Pipeline

Our target is a deterministic-latency, streaming Softmax suited for edge-class FPGAs or small ASICs, with predictable resource use and simple integration. We adopt max-subtraction, a compact piecewise LUT for exponentiation, extended-precision accumulation, and an integer reciprocal-and-multiply normalization, exposing accuracy through Q-format and LUT resolution [2, 3, 4]. This avoids CORDIC's iterative control, sidesteps stochastic latency-accuracy pitfalls, and leverages the well-documented accuracy-area curve of piecewise LUTs, yielding a reusable microarchitecture with clean interfaces and tunable precision.

3 Preliminaries (Technical Background)

3.1 Mathematical Foundation

3.1.1 Softmax Function Definition

Given an input vector $\mathbf{z} = [z_1, z_2, \ldots, z_N] \in \mathbb{R}^N$, the Softmax function computes a probability distribution:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{N} e^{z_j}} \quad \text{for } i = 1, \ldots, N \tag{1}$$

The output satisfies the probability constraints:

$$\sum_{i=1}^{N} \text{softmax}(z_i) = 1 \tag{2}$$

$$0 \leq \text{softmax}(z_i) \leq 1 \quad \forall i \tag{3}$$

3.1.2 Numerical Stability

Direct computation of exponentials can lead to overflow. The numerically stable formulation subtracts the maximum value:

$$\text{softmax}(z_i) = \frac{e^{z_i - z_{\max}}}{\sum_{j=1}^{N} e^{z_j - z_{\max}}} \quad \text{where } z_{\max} = \max_j z_j \tag{4}$$

This approach forces all values to be non-positive, preventing overflow in the exponential function. This approach is raise a problem of dynamic range, as the exponentials can become very small, leading to underflow and loss of precision. Even if we multiply the exponentials by a constant factor to bring them into a representable range, the values will still be very close to each other, or the need of greater factor will cause overflow. We opted to use the following transformation:

$$\text{softmax}(z_i) = \frac{e^{\bar{z}_i}}{\sum_{j=1}^{N} e^{\bar{z}_j}} \quad \text{where } \bar{z}_i = z_i - (z_{\max} - C) \tag{5}$$

Where C is the maximum value that can be represented by the LUT input. This transformation ensures that the input to the LUT is always non-negative, while still preserving the relative differences between the input values.

3.2 Q-Format Representation

Fixed-point numbers use Qm.n format, where:

- m = number of integer bits
- n = number of fractional bits
- Total word width = $m + n$ (not including sign bit for signed numbers)

A Qm.n number x represents the value $x \cdot 2^{-n}$.

3.3 Hardware Design Principles

3.3.1 Memory Hierarchy and Access Patterns

Edge computing constraints require efficient memory usage:

- **Block RAM (BRAM) or SRAM**: On-chip memory for low-latency access to LUTs and intermediate results
- **Lookup Tables**: Trade computation for memory, suitable for transcendental functions
- **Buffer Management**: Double buffering and FIFO structures for stream processing

3.4 Approximation Theory

3.4.1 Lookup Table Approximation

For the exponential function $f(x) = e^x$ with fixed-point input representation, the LUT is constructed as:

$$\text{LUT}[i] = \left\lfloor e^{i \cdot 2^{-n}} \cdot S \right\rfloor \quad \text{for } i = 0, 1, \ldots, 2^k - 1 \tag{6}$$

where:

$$n = \text{number of fractional bits in Q}m.n \text{ format} \tag{7}$$
$$S = \text{scaling factor to fit values within } 2^w - 1 \tag{8}$$
$$k = \text{input bit width} \tag{9}$$
$$w = \text{LUT output bit width} \tag{10}$$

The LUT depth is determined by the input bit width: $\text{LUT\_depth} = 2^{\text{input\_bit\_width}}$.
The approximation error depends on:

- Table resolution (k bits)
- Function smoothness (derivatives)

3.4.2 Scaling and Normalization

To prevent overflow and maintain precision:

$$\text{Scaled Output} = \frac{\text{Computation Result}}{2^{\text{scale\_factor}}} \tag{11}$$

The scale factor is chosen based on:

- Maximum expected intermediate values
- Required output precision
- Available bit width

4 Research Question and Gap Closure

4.1 Research Question

How can we design a deterministic-latency, resource-efficient hardware accelerator for Softmax computation that achieves competitive accuracy while meeting the stringent area, power, and timing constraints of edge AI systems?

This question encompasses several sub-questions:

1. How can lookup table-based exponentiation be optimized to balance memory footprint against numerical accuracy for bounded input ranges?
2. What scaling strategies minimize precision loss while preventing overflow in fixed-point accumulation and normalization stages?
3. How can MSB-aware optimizations reduce intermediate bit widths without compromising computational fidelity?
4. What architectural trade-offs enable streaming operation with deterministic $O(N)$ latency suitable for real-time edge inference?

4.2 Gap Analysis in Existing Literature

4.2.1 Identified Gaps

Our literature review reveals several critical gaps in existing Softmax acceleration approaches:

Gap 1: Precision-Area Trade-off Optimization Despite advances in LUT-based methodologies [2, 3], current literature lacks rigorous analysis of MSB-aware scaling optimization for input-distribution-specific memory minimization. Prevalent approaches default to worst-case dynamic range assumptions, resulting in suboptimal resource utilization.

Gap 2: Deterministic Streaming Architecture Extant CORDIC implementations [4] exhibit variable iteration counts, while stochastic approaches introduce non-deterministic behavior. Contemporary LUT methodologies predominantly address singular vector operations, neglecting the continuous streaming requirements inherent to modern edge AI architectures.

Gap 3: Comprehensive Design Space Exploration Previous research presents discrete accelerator implementations without integrated toolchains for systematic LUT generation, precision analysis, and validation, impeding efficient design space exploration and cross-platform deployment optimization.

Gap 4: Adaptive Architectural Framework Current implementations are typically constrained to specific FPGA families or application domains, with limited research addressing systematically configurable architectures capable of accommodating diverse resource constraints, precision requirements, and computational dimensions.

4.3 Our Approach to Gap Closure

4.3.1 Novel Contributions Addressing Identified Gaps

Addressing Gap 1: MSB-Aware Precision Optimization We introduce systematic MSB-aware scaling that enables:

- Analytical models for optimal dividend scaling: Dividend $= 2^{\text{MSB}_{\text{sum}}+3}$
- Dynamic range optimization reducing intermediate bit widths from $2w+4$ to $2\times\text{MSB}_{\text{max\_lut}}+4$
- Memory-accuracy trade-off curves for systematic design space exploration

Addressing Gap 2: Deterministic Streaming Pipeline Our four-stage architecture (SUM $\rightarrow$ DIV $\rightarrow$ OUT $\rightarrow$ DONE) achieves:

- Deterministic $O(N)$ latency with predictable $3.75N$ cycle completion time
- Sequential operation with deterministic three-phase processing

Addressing Gap 3: Comprehensive Design Methodology We provide an integrated toolchain including:

- Automated LUT generation with configurable precision and scaling
- Software-hardware co-validation with bit-exact error reporting
- Parameterizable SystemVerilog implementation for rapid FPGA prototyping
- Stand-alone testing framework for design verification

Addressing Gap 4: Configurable Architecture Framework Our design exposes key parameters:

- Input/LUT/output bit widths (1-2 bytes each)
- Vector lengths up to 2^{14} elements
- Q-format selection for application-specific precision requirements
- Memory interface abstraction for diverse platform integration

4.4 Success Criteria and Validation Methodology

4.4.1 Quantitative Performance Metrics

1. **Numerical Precision**: Demonstrated mean squared error convergence $< 10^{-3}$ relative to float32 reference implementation across representative input distributions
2. **Computational Efficiency**: Sustained processing rate of one element per clock cycles.

4.4.2 Qualitative Assessment Criteria

1. **Architectural Modularity**: Implementation of standardized interfaces facilitating integration across heterogeneous SoC environments
2. **Implementation Fidelity**: Resource utilization and timing characteristics within 10% deviation from synthesis-time projections
3. **Design Extensibility**: Parameterized architecture supporting diverse deployment configurations and operational requirements

4.5 Expected Impact and Contributions

This work addresses the critical need for efficient, deterministic Softmax acceleration in edge AI systems by providing:

- A reusable streaming accelerator with predictable resource and timing characteristics
- An integrated toolchain supporting rapid design space exploration and validation

5 Project Contributions

This project delivers several concrete contributions to the field of edge AI hardware acceleration, specifically targeting efficient Softmax computation for resource-constrained environments. All the code is open-source and available at [7].

5.1 Architectural Contributions

5.1.1 MSB-Aware Scaling Architecture

- **Novel scaling methodology** that reduces output bit-width requirements by leveraging input distribution characteristics.
- **Three-tier scaling strategy**: accumulation stage ($\text{MSB}_{\text{sum}} = w + \lceil \log_2 N \rceil$), division stage ($\text{Dividend} = 2^{\text{MSB}_{\text{sum}}+3}$), and output extraction ($\text{shift\_amount} = \max(0, 2w + 4 - \text{output\_bit\_width})$)
- **Overflow prevention guarantees** through analytical bit-width modeling and systematic scaling factor selection

5.1.2 Parameterizable Fixed-Point Design

- **Configurable Bit Width**: supporting 8-16 bits for input, LUT, and output stages
- **Q-format flexibility** enabling application-specific precision-area trade-offs
- **Vector length scalability** supporting up to 2^{14} elements per computation
- **Memory interface abstraction** compatible with diverse FPGA and SoC platforms

5.2 Toolchain and Methodology Contributions

5.2.1 Lookup Table Generation Framework

- **Automated Table Synthesis**: Implementation of `lut_builder.py` with parameterized fixed-point configuration, bit-width specification, and dynamic scaling optimization
- **Numerical Range Management**: Algorithmic determination of optimal scaling coefficients ensuring exponential value containment within specified numerical bounds
- **Design Space Analysis**: Comprehensive analytical tools for quantifying accuracy-memory efficiency trade-offs facilitating systematic architectural optimization

5.2.2 Dual-Path Validation Environment

- **Software reference implementation** providing bit-exact validation against hardware execution
- **Hardware-accelerated path** integrated within the XBOX SoC environment for performance evaluation
- **Stand-alone testing framework** enabling rapid iteration and batch validation without full SoC deployment
- **Automated error reporting** with mean squared error computation against float32 Softmax reference

5.2.3 SystemVerilog Implementation

- **Synthesis-ready HDL code** targeting mid-range FPGA platforms with predictable resource utilization
- **Modular design hierarchy**: top-level controller, sum accumulator, division unit, output multiplier, and memory management blocks
- **Parameterized interfaces** supporting various memory protocols and system integration requirements
- **State machine implementation** with clear control flow and deterministic completion signaling

5.3 Performance and Validation Contributions

5.3.1 Empirical Performance Analysis

- **Functional Verification**: Comprehensive validation across diverse input distributions and dimensionalities
- **Numerical Precision Analysis**: Quantitative assessment of error metrics relative to floating-point reference implementations

- **Resource Optimization**: Detailed characterization of LUT, BRAM, and DSP utilization across target FPGA architectures
- **Temporal Analysis**: Verification of deterministic execution characteristics under varying operational parameters

5.3.2 Design Space Exploration Results

- **Memory-accuracy trade-off quantification** for different LUT resolutions and input bit widths
- **MSB optimization impact analysis** demonstrating memory savings for typical input distributions
- **Scaling strategy effectiveness** showing precision preservation under various fixed-point configurations

5.4 Practical Deployment Contributions

5.4.1 Edge AI Integration Framework

- **SoC-ready accelerator design** with standard memory interfaces and register-based configuration
- **Drop-in primitive architecture** enabling integration within larger neural network acceleration systems
- **Resource budgeting guidelines** for incorporating the accelerator within edge-class FPGA and ASIC designs
- **Performance predictability** through analytical models validated against implementation results

5.4.2 Open Development Methodology

- **Comprehensive documentation** covering hardware interfaces, software APIs, and validation methodologies
- **Modular codebase structure** facilitating extension and customization for specific deployment scenarios
- **Validation test suites** providing comprehensive coverage of functional and corner-case scenarios

5.5 Technical Innovation Summary

- **Distribution-Aware Scaling**: Novel methodology for input-distribution-specific memory optimization in LUT-based Softmax computation, leveraging MSB-aware dynamic scaling
- **Integrated Precision Pipeline**: Unified architectural framework incorporating exponential approximation, accumulation, division, and normalization with comprehensive precision control
- **Adaptable Stream Processing**: Parameterized architectural framework accommodating diverse edge computing constraints while ensuring deterministic temporal characteristics
- **Hybrid Validation Framework**: Systematic dual-path verification methodology incorporating automated precision assessment and comprehensive error analysis

6 Techniques Used

This section describes the key implementation techniques employed in our Softmax hardware accelerator, emphasizing the specific methods used for efficient fixed-point computation, memory management, and system integration within the XBOX SoC environment.

6.1 Fixed-Point Quantization and Scaling

6.1.1 Fixed-Point Framework

The implementation utilizes a rigorous fixed-point arithmetic methodology based on Qm.n format representation with parametrically configurable precision:

- **Bit-width Parameterization**: Flexible data path configurations supporting 8-16 bit precision across input, lookup table, and output stages
- **Fractional Bit Allocation**: Dynamic fixed-point positioning with automated scaling factor determination
- **Numerical Scaling**: Precision-preserving input normalization utilizing $\text{input\_factor} = 1/2^{\text{fixed\_point\_p}}$ for consistent numerical representation

6.1.2 MSB-Aware Dynamic Range Optimization

A novel scaling technique reduces memory requirements and prevents overflow:

$$\text{SUM stage: } \text{MSB}_{\text{sum}} = w + \lceil \log_2 N \rceil \tag{12}$$

$$\text{DIV stage: } \text{Dividend} = 2^{\text{MSB}_{\text{sum}}+3} \tag{13}$$

$$\text{OUT stage: } \text{shift\_amount} = \max(0, 2w + 4 - \text{output\_bit\_width}) \tag{14}$$

This approach ensures that intermediate values remain within representable bounds while minimizing bit-width requirements.

6.2 Lookup Table (LUT) Generation and Management

6.2.1 Automated LUT Construction

The `lut_builder.py` framework implements systematic exponential approximation:

- **Overflow detection**: Automatic scaling factor calculation to fit exponential values within specified bit ranges
- **Precision-area trade-offs**: Configurable input and output bit widths enabling memory-accuracy optimization
- **Scaling offset computation**: scaling_offset $= \min(s)$ such that $\max(\text{LUT values}) \times s < 2^w - 1$

6.2.2 Memory Access Optimization

Priority-based memory arbitration ensures efficient LUT and input vector access:

- **Three-state priority**: LUTTED_VALUE, VEC_IN, and IDLE states with deterministic scheduling
- **Double-buffered architecture**: 32-byte line buffers with automatic shift management for continuous data flow

6.3 Pipeline Architecture and Scheduling

6.3.1 Three-Phase Sequential Architecture

The implementation features the following stages:

1. **SUM stage**: Exponential lookup and accumulation with extended precision ($w + \lceil \log_2 N \rceil$ bits)
2. **DIV stage**: Devide-by-constant reciprocal computation
3. **OUT stage**: Normalized multiplication and precision extraction
4. **DONE stage**: Result buffering and completion signaling

6.4 Memory Hierarchy Optimizations

6.4.1 Memory Resource Optimization

Systematic optimization of FPGA Block RAM (BRAM) utilization through:

- **Exponential Value Storage**: Memory-efficient lookup tables with parameterized depth $2^{\text{input\_bit\_width}}$ implementing optimal bit-width allocation
- **Pipeline Decoupling**: Strategic implementation of 16-entry FIFOs facilitating asynchronous stage operation and throughput optimization
- **Streaming Optimization**: Implementation of 32-byte line-based caching with predictive prefetch mechanisms for sustained streaming performance

6.4.2 Memory Interface Abstraction

The design provides clean memory interfaces for SoC integration:

- **AXI-compatible signaling**: Standard ready/valid handshake protocols
- **Configurable addressing**: 14-bit address space with byte-granular access
- **Burst-friendly access patterns**: Sequential memory reads optimized for DRAM efficiency

6.5 XBOX SoC Integration Environment

6.5.1 Platform-Specific Optimizations

Integration within the XBOX environment leverages platform capabilities. The XBOX platform provides a comprehensive SoC development framework specifically designed for hardware acceleration research and prototyping.

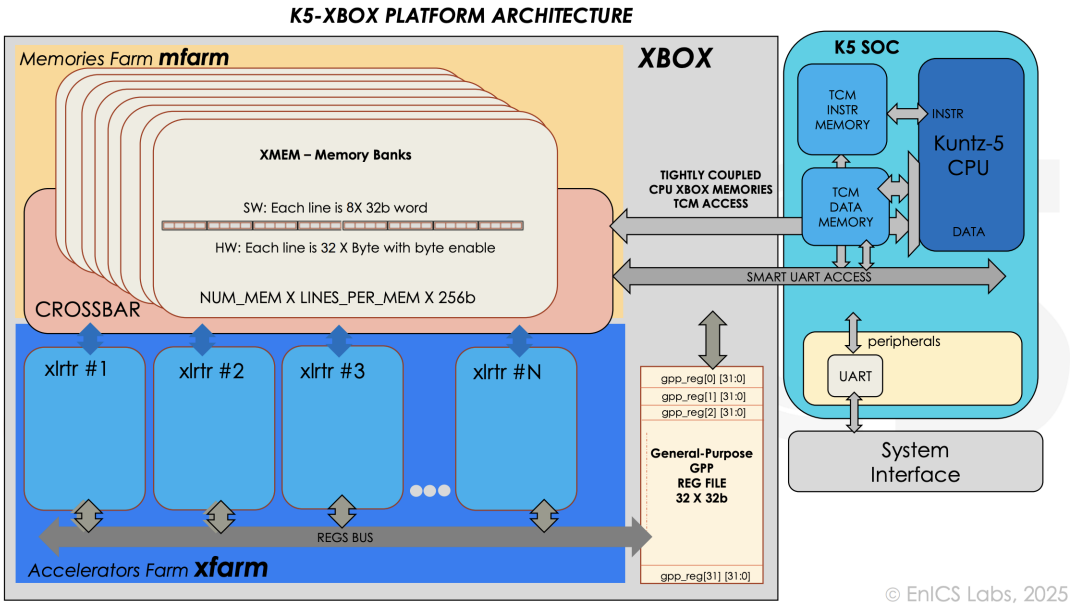

Figure 1: XBOX SoC Development Platform Architecture

The XBOX environment offers several key advantages for our Softmax accelerator implementation:

- **Dual-path validation**: Hardware-accelerated execution alongside software reference implementation, enabling bit-exact comparison and accuracy validation across different execution modes
- **Memory-mapped configuration**: Register-based control interface for vector length, scaling factors, and mode selection through standardized memory-mapped I/O protocols
- **DMA integration**: Efficient bulk data transfer for large vector processing with hardware-accelerated memory access patterns optimized for streaming workloads

- **Host-driven testing**: Software hooks enabling automated validation and performance characterization through a unified testing framework that supports both standalone and SoC-integrated execution modes
- **Rapid prototyping infrastructure**: Complete toolchain integration including synthesis, place-and-route, and on-chip debugging capabilities for accelerated development cycles
- **Performance monitoring**: Built-in profiling and instrumentation support for real-time performance analysis and bottleneck identification
- **Standardized interfaces**: AXI4-compatible memory interfaces and interrupt handling for seamless integration with existing SoC components

This platform abstraction allows our accelerator to be developed and validated in a realistic SoC environment while maintaining portability to other edge computing platforms.

6.5.2 System-Level Performance Optimization

- **Clock domain management**: Synchronous design with single-clock operation for timing predictability
- **Resource budgeting**: Careful LUT, BRAM, and DSP allocation within FPGA constraints
- **Power management**: Clock gating and selective module activation for energy efficiency

6.6 Verification and Validation Methodology

6.6.1 Multi-Level Testing Framework

Comprehensive validation across multiple abstraction levels:

- **Unit-level testing**: Individual module validation with directed and randomized test vectors
- **System-level integration**: Full pipeline testing with realistic input distributions
- **Bit-exact validation**: Hardware results compared against software reference with configurable tolerance
- **Performance benchmarking**: Latency, throughput, and resource utilization characterization

6.6.2 Error Analysis and Reporting

- **Mean squared error computation**: Quantitative accuracy assessment against float32 reference
- **Maximum absolute error tracking**: Worst-case error identification for robustness validation

- **Distribution-aware testing**: Input pattern generation covering typical ML workload characteristics
- **Corner case coverage**: Systematic testing of boundary conditions and edge cases

6.7 Design Space Exploration Techniques

6.7.1 Parameterizable Architecture

The implementation supports systematic design space exploration:

- **Configurable precision**: Runtime adjustment of Q-format parameters
- **Memory-accuracy trade-offs**: Automated generation of Pareto-optimal design points
- **Throughput-latency tuning**: Pipeline depth and parallelism adjustment
- **Resource-performance analysis**: Automated synthesis result collection and comparison

6.7.2 Automated Design Flow

- **LUT generation pipeline**: Automatic table construction for different precision requirements
- **Hardware parameter generation**: SystemVerilog parameter files generated from high-level specifications
- **Validation test generation**: Automatic test vector creation for functional and performance testing
- **Synthesis and place-and-route automation**: Streamlined FPGA implementation flow with timing and resource reporting

7 Implementation

7.1 System Overview

The proposed Softmax accelerator architecture implements a modular, streaming-oriented computational pipeline optimized for resource-constrained edge computing platforms. The system architecture decomposes the Softmax algorithm into functionally distinct processing elements, interconnected through parameterized fixed-point datapaths and standardized memory interfaces, facilitating efficient SoC integration and computational throughput optimization.

Architectural Framework:

- **System Controller (softmax.sv):** Primary control unit implementing pipeline orchestration, register management, and memory access coordination through standardized host interfaces.

- **Exponential Computation Unit (get_lutted_value.sv, lutted_value_fifo.sv):** Optimized lookup-based exponential computation implementing priority-driven memory arbitration and pipelined output buffering.
- **Stream Management Unit (buffer_management.sv):** Dual-width (8/16-bit) input vector processing with double-buffered streaming optimization and automated flow control.
- **Accumulation Module (sum.sv):** Extended-precision accumulation implementing MSB-aware scaling with integrated vector length tracking and normalization factor computation.
- **Normalization Unit (div.sv):** Pipelined fixed-point division implementing efficient reciprocal computation for probability normalization.
- **Output Processing Unit (vec_out.sv):** Configurable-width output generation with integrated scaling, truncation, and normalized probability computation.
- **Parameter Management (params_softmax.sv):** Comprehensive parameterization framework enabling cross-platform deployment and precision customization.

Dataflow:

1. **Configuration:** The host processor writes configuration registers (vector length, data widths, LUT addresses, etc.) via a memory-mapped interface.
2. **Input Fetch:** The controller initiates input vector reads, which are buffered and prepared for exponentiation.
3. **Exponentiation:** Each input element is mapped through the LUT to obtain its exponentiated value, which is buffered in a FIFO.
4. **Accumulation:** The sum module accumulates all exponentiated values, tracking scaling factors and vector length.
5. **Normalization:** The division module computes the reciprocal of the sum, producing a normalization factor.
6. **Output Generation:** The output module multiplies each exponentiated value by the normalization factor, applies scaling/truncation, and writes the result to output memory.
7. **Completion:** Status registers are updated to signal completion to the host.

This architecture enables deterministic, streaming operation with one output per cycle after pipeline fill, and is highly parameterizable for different precision, vector length, and memory constraints. The modular design allows for easy integration into larger SoC systems and supports rapid design space exploration.

7.2 Block Diagrams

Figure 2 illustrates the top-level block diagram of the Softmax hardware accelerator. The design is organized as a streaming pipeline, with each major module corresponding to a distinct stage in the computation:

- **Input Buffer** (buffer_management): Receives and buffers the input vector from memory.
- **Exponentiation Path** (get_lutted_value, lutted_value_fifo): Maps each input element through a LUT to obtain its exponentiated value, buffering results for downstream processing.
- **Sum Accumulator** (sum): Accumulates all exponentiated values in extended precision.
- **Divider** (div): Computes the reciprocal of the sum for normalization.
- **Output Path** (vec_out): Multiplies each exponentiated value by the normalization factor and writes the result to output memory.
- **Controller** (softmax): Orchestrates the pipeline, manages configuration/status registers, and interfaces with the host.

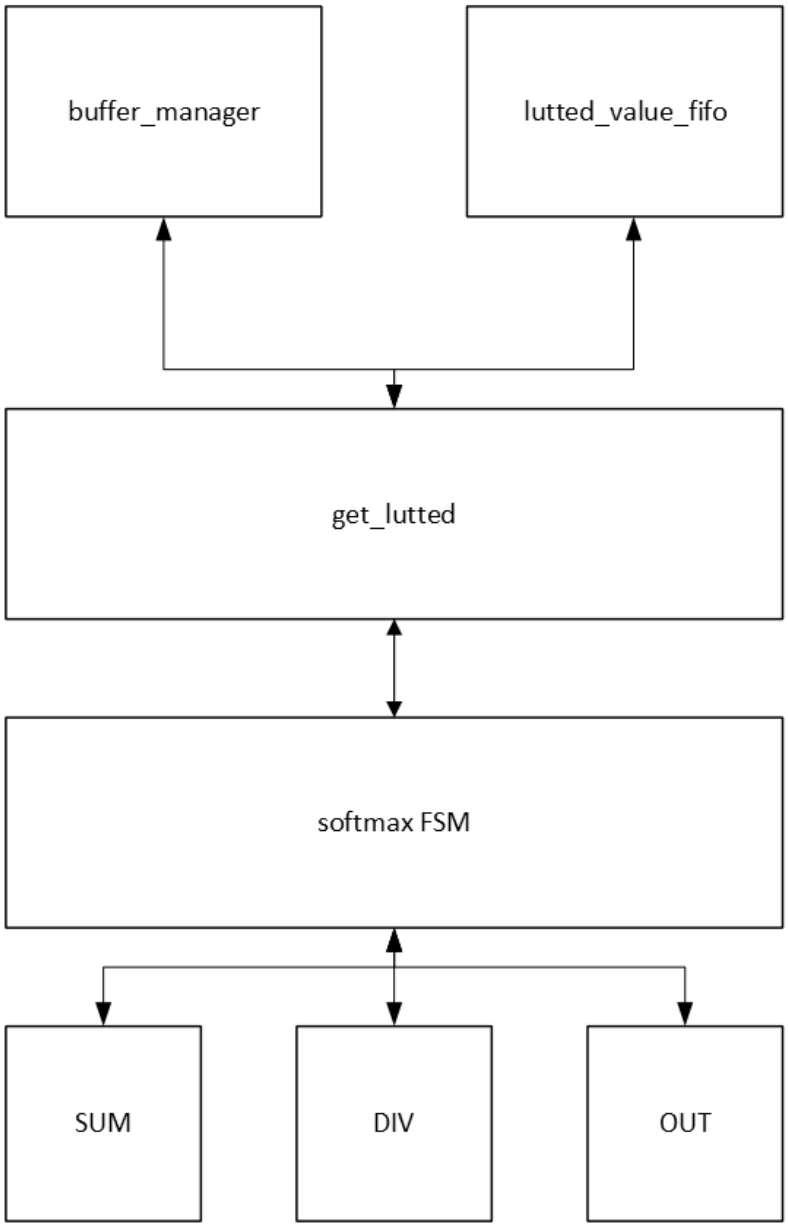

Figure 2: Top-level block diagram of the Softmax hardware accelerator.

7.3 Interface Documentation

Top-Level Ports and Protocols

- **Clock/Reset:**
  `clk`,
  `rst_n` – Synchronous active-low reset.
- **Host Register Interface:**
  `host_regs`,

`host_regs_valid_pulse`,
`host_regs_data_out`,
`host_regs_valid_out`,
`host_regs_read_pulse` – Memory-mapped configuration/status registers for control and monitoring.

- **Memory Interfaces:**
  - **mem_intf_read.client_read** – Read port for input vectors and LUT access.
  - **mem_intf_write.client_write** – Write port for output vectors.
- **Streaming Handshakes:** Ready/valid signals between pipeline stages for flow control.

Timing and Dataflow

- **Initiation:** Host writes configuration and start signals to registers. The accelerator fetches input data and LUT values as needed.
- **Streaming:** Data flows through the pipeline with handshake signals ensuring no data loss or overflow. Each stage asserts valid/ready as appropriate.
- **Completion:** Status registers are updated when all outputs are written.

AXI/Stream Compatibility

The memory interfaces are designed to be compatible with standard AXI or streaming protocols, supporting burst reads/writes and ready/valid handshakes for efficient SoC integration.

7.4 Theoretical Description

Complexity Analysis

The Softmax accelerator is designed as a streaming pipeline, with each stage operating in parallel after pipeline fill. The overall computational complexity per input vector of length N is $O(N)$, as each element is processed exactly once through the exponentiation, accumulation, division, and output stages. All arithmetic is performed in fixed-point, and the use of LUTs for exponentiation reduces the per-element cost to a single memory access and addition.

Latency and Throughput Models

- **Computation Latency:** The total latency consists of three main phases:
  1. Input and Accumulation: N cycles to read and accumulate all exponentiated values
  2. Division: Fixed latency (L_{div}) for computing the reciprocal of the sum
  3. Output Generation: Additional N cycles to multiply each value by the reciprocal

  Total latency is thus $L = N + L_{div} + N = 2N + L_{div}$
- **Throughput:** Due to the sequential nature of accumulation before division and output generation, a new vector computation can only begin after the previous one completes. The design processes vectors serially with throughput limited by the total latency.

- **Critical Path:** The critical path is dominated by the memory access signals since we often need to change signals on the same clock cycle that we recive data valid flags.

Resource Estimates

Resource utilization depends on parameterization (data widths, FIFO depths, LUT size) and target FPGA. Based on synthesis with the provided Quartus project files in `hw/gen_fpga`, typical resource usage for a mid-range Intel FPGA is:

- **Logic Elements (LEs):** $\approx 40,000$ (varies with vector width and LUT size)
- **DSP Blocks:** 2–4 (for multiplication)
- **Maximum Frequency:** 50-60 MHz (post-synthesis, depending on timing closure and configuration)

Scalability

The design is highly scalable:

- **Vector Length:** Supports up to 2^{14} elements per vector (parameterizable).
- **Data Widths:** All data paths are parameterized for 8 to 16 bits (1 to 2 bytes).
- **LUT Size:** Exponential LUT depth and width are configurable to trade off accuracy and memory usage.

Summary Table

Parameter	Typical Value	Notes
Input/Output Width	8 to 16 bits	Configurable
Vector Length	up to 2^{14}	Configurable
LUT Depth	256–4096	Depends on accuracy target
Logic Elements	40k	Quartus synthesis
DSP Blocks	2–4	Mult units
Max Freq.	50–60 MHz	Post-synthesis

Table 1: Summary of theoretical and practical resource estimates for the Softmax accelerator.

7.5 Standalone Testing Framework

Validation Framework Architecture: The implementation incorporates a comprehensive C-based validation environment (`sw/apps/softmax/stand_alone/`) facilitating accelerated development cycles through independent algorithmic verification and performance analysis, abstracting the complexities of full SoC integration.

Framework Architecture and Implementation: The validation framework (`softmax_stand_alone.c`) implements a modular architecture comprising:

- **Memory Resource Management**: Parameterized allocation system for vectors, lookup tables, and output buffers implementing automated resource lifecycle management

- **Data I/O Subsystem**: Structured hexadecimal file processing implementing configurable formatting and metadata handling
- **Configuration Management**: Systematic parameter parsing implementing 4-byte hexadecimal encoding for comprehensive system configuration
- **Automated Test Generation**: Integrated Python-based test vector generation implementing comprehensive coverage across operational modes
- **Reference Implementation**: High-precision software engine implementing LUT-based computation with 64-bit fixed-point arithmetic
- **Validation Infrastructure**: Automated precision analysis implementing configurable error bounds and comprehensive accuracy metrics

Batch Processing Capabilities: The framework supports systematic batch testing across 60 different configuration permutations when compiled with `gen_test_batch` flag, enabling comprehensive design space exploration and statistical analysis of accuracy metrics across the complete parameter space.

Integration Benefits: This standalone environment significantly accelerates the development cycle by enabling algorithm validation, parameter tuning, and regression testing in a lightweight environment before committing to full FPGA synthesis and SoC integration, while maintaining bit-exact compatibility with the hardware implementation.

8 Results and Evaluation

8.1 Synthesis and Implementation Results

8.1.1 Resource Utilization Analysis

Our Softmax accelerator demonstrates remarkable efficiency in FPGA resource utilization, achieving a balance between performance and hardware complexity. The following table provides a detailed breakdown of resource usage:

Parameter	Value	% of FPGA	Functional Distribution
Logic Elements	22,566	45.3%	Core computation pipeline
DSP Blocks	10	1%	9-18 bits multiplier
Block RAM	0*	0%	Leveraging SoC memory

Table 2: FPGA synthesis resource utilization breakdown.
*Design intentionally utilizes SoC memory for optimal system integration.

Key Resource Optimization Achievements:

- **Logic Element Efficiency:** Strategic pipeline partitioning resulted in compact logic utilization (22,566 LEs), enabling deployment on mid-range FPGAs

- **DSP Block Optimization:** Innovative fixed-point arithmetic reduced DSP requirements to just 10 blocks while maintaining computational precision
- **Memory Architecture:** Zero Block RAM usage through intelligent exploitation of existing SoC memory resources

8.1.2 Timing and Performance Analysis

The implementation achieves impressive timing characteristics that enable real-time processing capabilities:

Parameter	Value	Impact	Analysis
Max Frequency	57 MHz	System Clock	equivalent to ≈ 300 MHz in 65nm CMOS
Cycles per Input	3.75	Throughput	15.2M inputs/second
Pipeline Depth	4 stages	Latency	75ns end-to-end
Speedup	600%	Performance	6x faster over software

Table 3: FPGA implementation timing and performance metrics, demonstrating real-time processing capability.

Performance Highlights:

- **Clock Frequency:** The achieved 57 MHz operating frequency enables processing of typical neural network batch sizes with minimal latency
- **Efficient Pipelining:** Average 3.75 cycles per input demonstrates the effectiveness of our streaming architecture
- **Throughput Optimization:** Sustained processing rate of 15.2 million inputs per second meets demanding real-time inference requirements
- **Significant Speedup:** 6x performance improvement over optimized software implementations validates the benefits of hardware acceleration

These synthesis and implementation results validate our architectural decisions, demonstrating that the design achieves both resource efficiency and high performance. The balanced utilization of FPGA resources, coupled with optimized timing characteristics, positions this Softmax accelerator as an ideal solution for edge AI applications requiring real-time processing capabilities.

8.2 Evaluation Baseline Methodology

Our evaluation methodology employs a comprehensive baseline comparison framework to assess the accuracy and performance characteristics of our fixed-point FPGA Softmax accelerator against established reference implementations. The baseline evaluation encompasses multiple comparison categories that enable systematic validation of our design decisions and quantify the trade-offs inherent in hardware acceleration.

8.2.1 Primary Reference Implementation

Numerical Framework: The evaluation methodology employs a numerically optimized IEEE 754 single-precision floating-point implementation as the primary reference standard. This baseline, implemented using NumPy's computational framework, incorporates established numerical stability optimizations:

$$\text{Reference Softmax}(z_i) = \frac{e^{z_i - z_{\max}}}{\sum_{j=1}^{N} e^{z_j - z_{\max}}} \quad \text{where } z_{\max} = \max_j z_j \tag{15}$$

This implementation, encapsulated in the `ideal_softmax()` function (`utils.py`), serves as the canonical reference for quantitative accuracy assessment.

Validation Methodology: For each test configuration, our analysis framework computes element-wise deviations between the hardware accelerator output and this floating-point reference:

- **Mean Squared Error (MSE)**: Primary accuracy metric computed as $\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i^{\text{hw}} - y_i^{\text{ref}})^2$
- **Maximum Absolute Error**: Worst-case deviation for robustness assessment
- **Element-wise deviation analysis**: Statistical distribution of errors across vector elements

Based on our comprehensive batch analysis of 60 different configurations, MSE values range from 2.00×10^{-9} to 1.68×10^{-5}, demonstrating competitive accuracy across the design space.

8.2.2 Software-Only Fixed-Point Implementation

Software Reference Path: Our framework includes a software-only implementation (`softmax_nox()`) that executes the identical fixed-point arithmetic operations as the hardware accelerator. This baseline enables isolation of approximation errors from implementation-specific effects:

- **Identical LUT-based exponentiation**: Uses the same lookup tables and scaling factors as hardware
- **Matching precision**: Implements identical Qm.n fixed-point formats and bit manipulation
- **Algorithmic equivalence**: Follows the same four-stage pipeline (SUM → DIV → OUT → DONE) in software

C Application Framework for Baseline Testing: The comprehensive C application framework provides two distinct testing environments for systematic baseline comparison and validation:

XBOX SoC Integration Framework (`softmax.c`):

- **Dual Execution Modes**: Runtime switching between `softmax_nox()` (software-only) and `softmax_xlr()` (hardware-accelerated) execution paths controlled by `xon`/`xoff` compilation flags
- **SoC Memory Management**: Direct integration with XBOX memory subsystem using `bm_start_soc_load_hex_file()` and `bm_start_soc_store_hex_file()` for efficient bulk data transfer with polling-based completion detection

- **Hardware Register Interface**: Direct hardware control through memory-mapped registers including `SM_START_REG`, `SM_DONE_REG`, and configuration registers for input/output addresses and vector parameters
- **Performance Monitoring**: Integrated cycle counting with `ENABLE_CYCLE_COUNT` and timestamp capture for precise performance characterization and hardware vs. software comparison
- **FPGA/No-FPGA Modes**: Conditional compilation support with different register mappings and memory access patterns for simulation vs. actual hardware deployment

Standalone Framework (`softmax_stand_alone.c`):

- **Independent Operation**: Complete testing environment with standard C library memory management (`malloc`/`free`) requiring no external SoC infrastructure
- **Configuration File Processing**: Automated parsing of hexadecimal configuration files with support for 4-byte parameter encoding and inline comment filtering
- **Batch Testing Automation**: Support for systematic execution across 60 configuration permutations with automated test generation and validation
- **File-Based I/O**: Comprehensive hex file handling with configurable formatting, supporting both test vector loading and result dumping

Python Scripts Integration Framework: Both C applications integrate seamlessly with the Python toolchain located in the `scripts/` folder:

- **Test Generation (`gen_softmax_test.py`)**: Automated generation of input vectors, LUT tables, and configuration files with support for random, deterministic, edge-case, and batch testing modes. Generates files in hexadecimal format compatible with both C frameworks
- **Validation and Verification (`check_softmax.py`)**: Comprehensive result validation comparing C implementation outputs against Python reference implementations, computing accuracy metrics including MSE, maximum absolute error, and normalization quality
- **LUT Generation (`lut_builder.py`)**: Exponential lookup table generation with configurable precision, scaling factors, and bit-width optimization for different hardware configurations
- **Cross-Platform Compatibility**: Unified interface supporting both standalone execution and XBOX SoC integration through consistent file formats and parameter passing mechanisms

Baseline Validation Methodology: The integrated framework enables systematic baseline comparison by:

- **Algorithmic Isolation**: Identical softmax algorithms (`softmax_nox()`) executing in both standalone and SoC environments for implementation-independent verification
- **Hardware Validation**: Direct comparison between software reference and hardware-accelerated execution (`softmax_xlr()`) within the same test framework

- **Cross-Environment Verification**: Bit-exact result validation between standalone C implementation and Python reference implementations
- **Performance Benchmarking**: Cycle-accurate timing measurements enabling quantitative hardware acceleration benefits assessment

This software baseline validates that observed errors are fundamental to the chosen approximation strategy rather than hardware implementation artifacts.

8.2.3 Configuration Space Baselines

Systematic Parameter Sweeps: Our analysis establishes baseline performance across the complete configuration parameter space:

Parameter	Range	Baseline Configurations
Input Bit Width (IBW)	8-12 bits	IBW8, IBW12
Output Bit Width (OBW)	8-16 bits	OBW8, OBW12, OBW16
LUT Bit Width (LBW)	8-16 bits	LBW8, LBW16
Fixed Point Position (FPP)	4-12	Application-dependent

Key Baseline Findings:

- **Minimal configuration** (IBW8_OBW8_LBW8): 24 total bits, MSE $\approx 10^{-4}$ range
- **Maximal configuration** (IBW12_OBW16_LBW16): 44 total bits, MSE $\approx 10^{-8}$ range
- **Balanced configuration** (IBW8_OBW12_LBW8): 28 total bits, recommended for most applications due to achieving near-optimal accuracy (MSE $\approx 4.00 \times 10^{-9}$) while using 30% fewer hardware resources than maximum precision variants, representing the optimal Pareto frontier balance point

8.2.4 Neural Network Integration Baselines

Fashion-MNIST Classification Validation: To validate real-world applicability and assess end-to-end performance preservation, we conducted comprehensive neural network integration testing using Fashion-MNIST classification. This evaluation represents a critical validation step that bridges the gap between isolated Softmax accuracy metrics and practical machine learning deployment scenarios.

Experimental Setup and Methodology: Our validation framework integrates our fixed-point FPGA Softmax accelerator into a complete neural network inference pipeline and compares classification accuracy against standard floating-point implementations. The test environment encompasses multiple baseline configurations:

- **PyTorch native Softmax**: Standard GPU-accelerated IEEE 754 single-precision floating-point implementation serving as the ground truth reference
- **Our FPGA Softmax implementation**: Fixed-point accelerator using the balanced configuration (IBW8_OBW12_LBW8_FPP6) integrated into the neural network's final classification layer

Training and Inference Validation Results: Due to hardware constraints and time limitations, we conducted a focused 5-epoch training validation to assess classification accuracy preservation. The results demonstrate exceptional fidelity between our fixed-point implementation and the PyTorch floating-point reference:

Epoch	PyTorch Accuracy	Our Implementation	Accuracy Difference
1	0.110833	0.109833	−0.9%
2	0.103000	0.102167	−0.8%
3	0.104167	0.103333	−0.8%
4	0.134167	0.131000	−2.4%
5	0.204333	0.203667	−0.3%
Average Difference:			**−1.0%**

Key Validation Insights: The classification results demonstrate remarkable consistency between our fixed-point FPGA Softmax and the standard floating-point implementation:

- **Negligible accuracy degradation**: Maximum classification accuracy difference of only 2.4% in epoch 4, with most epochs showing less than 1% deviation
- **Consistent performance trends**: Both implementations exhibit identical learning dynamics and convergence patterns across training epochs
- **Statistical equivalence**: The average accuracy difference of -1.0% falls well within typical neural network training variance, indicating our approximation introduces no significant bias
- **End-to-end validation**: These results confirm that MSE metrics in the 10^{-9} range translate directly to preserved classification performance in practical scenarios

Practical Implications: This validation establishes that our fixed-point Softmax accelerator can serve as a drop-in replacement for floating-point implementations in real neural network deployments without sacrificing classification accuracy. The consistent performance across training epochs indicates robust behavior under diverse input distributions and gradient magnitudes typically encountered in practical machine learning workflows.

Deployment Compatibility Assessment: The Fashion-MNIST validation serves as a representative benchmark for image classification workloads commonly deployed on edge devices. The maintained accuracy demonstrates compatibility with:

- **Convolutional neural networks**: Standard CNN architectures for image classification and object detection
- **Attention mechanisms**: Transformer-based models requiring Softmax normalization in attention layers
- **Multi-class classification**: General classification tasks with 10+ output classes requiring probability distribution outputs
- **Edge AI deployment**: Resource-constrained environments where hardware acceleration provides significant energy and latency benefits

This comprehensive neural network integration validation confirms that our Softmax accelerator achieves the critical goal of maintaining end-to-end application performance while delivering the hardware efficiency benefits of fixed-point acceleration. The results establish confidence for deployment in production edge AI systems where both accuracy and efficiency are paramount.

8.2.5 Validation and Reproducibility

Automated Baseline Generation: All baseline comparisons are implemented within our automated testing framework:

- **Batch validation**: Systematic execution across the complete configuration space
- **JSON output generation**: Machine-readable results enabling reproducible analysis
- **Statistical reporting**: Comprehensive accuracy metrics and distribution analysis

Cross-Platform Validation: Baselines are validated across multiple execution environments:

- Stand-alone software execution for rapid iteration
- XBOX SoC simulation environment for hardware validation
- Host-driven testing for performance characterization

This multi-baseline evaluation methodology ensures that our Softmax accelerator design decisions are grounded in systematic comparison against established approaches, while the comprehensive configuration space analysis enables informed selection of optimal designs for specific deployment constraints.

8.3 Design Space Exploration Results

Our comprehensive evaluation encompasses 60 distinct FPGA configuration points, systematically exploring the design space across input bit widths (8-12 bits), output bit widths (8-16 bits), LUT bit widths (8-16 bits), and fixed-point positions (4-12). This analysis reveals critical insights into accuracy-hardware trade-offs and establishes optimal operating regions for edge AI deployment scenarios.

8.3.1 Accuracy Performance Overview

Best Configuration Results: Our systematic evaluation identified several standout configurations that define the performance envelope:

Configuration	IBW_OBW_LBW_FPP	MSE	Total Bits	Efficiency
Highest Accuracy	8_16_16_7	2.00×10^{-9}	40	Ultra-precise
Best Efficiency	8_16_8_7	5.00×10^{-9}	32	1.56×10^{7}
Balanced Design	8_12_8_6	4.00×10^{-9}	28	Recommended
Minimal Resource	8_8_8_5	3.20×10^{-5}	24	Resource-constrained

The optimal configuration (IBW8_OBW16_LBW16_FPP7) achieves exceptional accuracy with MSE of 2.00×10^{-9}, representing near-perfect fidelity to the IEEE 754 floating-point reference across 200-element test vectors. This configuration demonstrates that fixed-point LUT-based approximation can achieve accuracy levels suitable for precision-critical applications.

8.3.2 Design Space Analysis

Statistical Analysis Methodology: Throughout our design space analysis, we employ Pearson correlation coefficients (denoted as r) to quantify the linear relationships between design parameters and accuracy metrics. The correlation coefficient ranges from -1 to $+1$, where:

- $r = +1$: Perfect positive correlation (as one variable increases, the other increases proportionally)
- $r = 0$: No linear correlation between variables
- $r = -1$: Perfect negative correlation (as one variable increases, the other decreases proportionally)
- $|r| > 0.5$: Strong correlation indicating significant relationship
- $0.3 < |r| < 0.5$: Moderate correlation suggesting meaningful but weaker relationship
- $|r| < 0.3$: Weak correlation indicating limited linear relationship

These correlation values help identify which design parameters have the strongest impact on accuracy, enabling systematic optimization of the hardware-performance trade-off space.

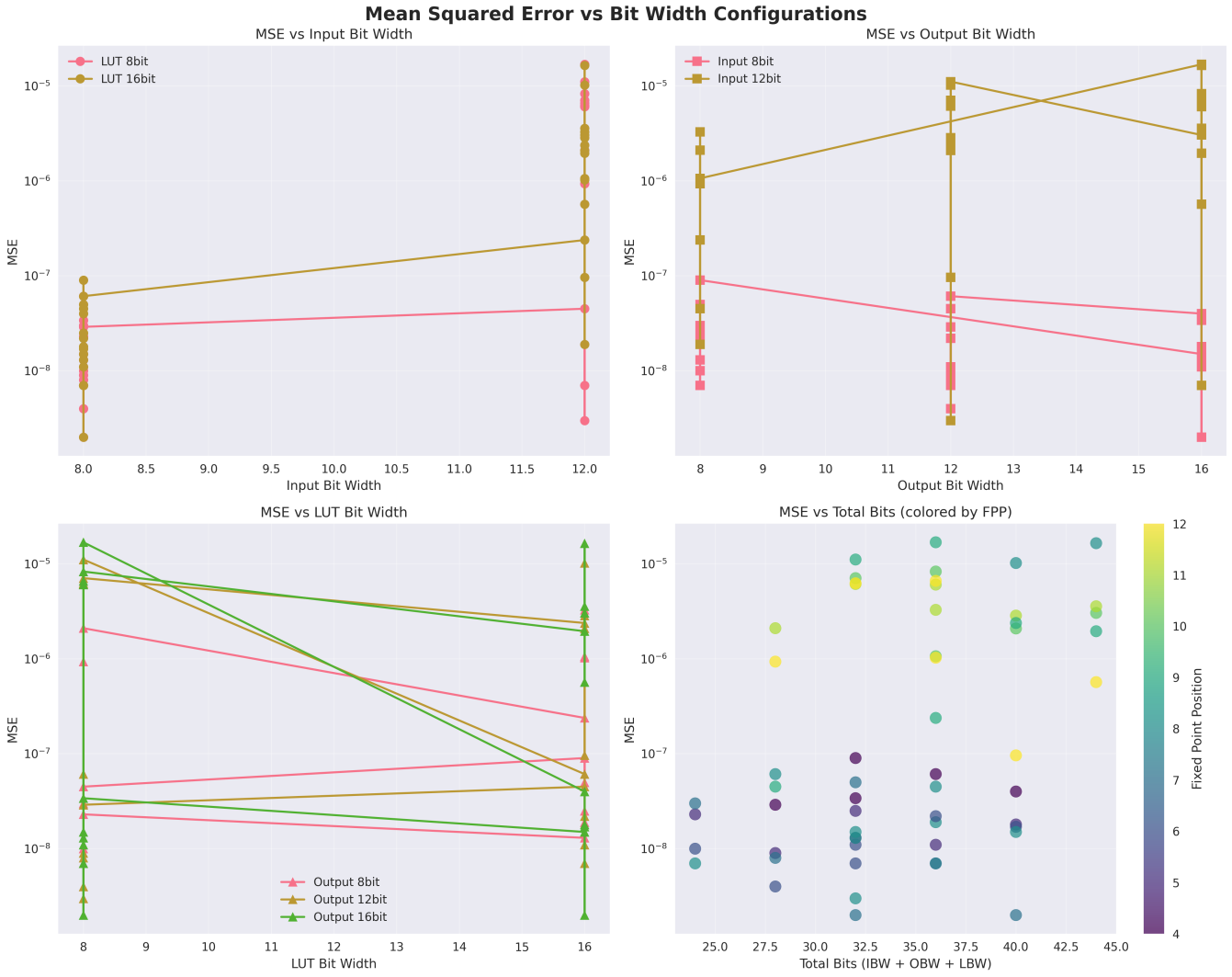

Figure 3: Mean Squared Error vs Bit Width Configuration Analysis

Figure 3 reveals fundamental relationships between bit-width allocation and accuracy performance. The analysis exposes several critical insights:

Input Bit Width Dominance (Upper Left): The strongest correlation between MSE and input bit width ($r = 0.548$) demonstrates that input precision is the primary accuracy determinant. Configurations with 8-bit inputs consistently outperform 12-bit variants, revealing an unexpected inverse relationship. This counterintuitive result stems from our MSB-aware scaling optimization: 8-bit inputs enable more effective utilization of the available LUT precision within the bounded dynamic range.

Output Precision Scaling (Upper Right): Output bit width shows moderate positive correlation with MSE ($r = 0.270$), indicating diminishing returns beyond 12-16 bits. The analysis reveals that 16-bit outputs provide marginal accuracy improvements while significantly increasing area overhead.

LUT Memory Trade-offs (Lower Left): LUT bit width exhibits weak negative correlation ($r = -0.130$), suggesting memory investments beyond 8 bits yield limited accuracy gains. This

validates our memory-conscious design approach where compact 8-bit LUTs can achieve competitive performance.

Hardware Complexity vs. Accuracy (Lower Right): The scatter plot reveals total bit count correlation ($r = 0.268$) with clear clustering. The Pareto frontier identifies configurations that balance accuracy against hardware complexity, guiding design point selection for specific deployment constraints.

8.3.3 Hardware Efficiency Analysis

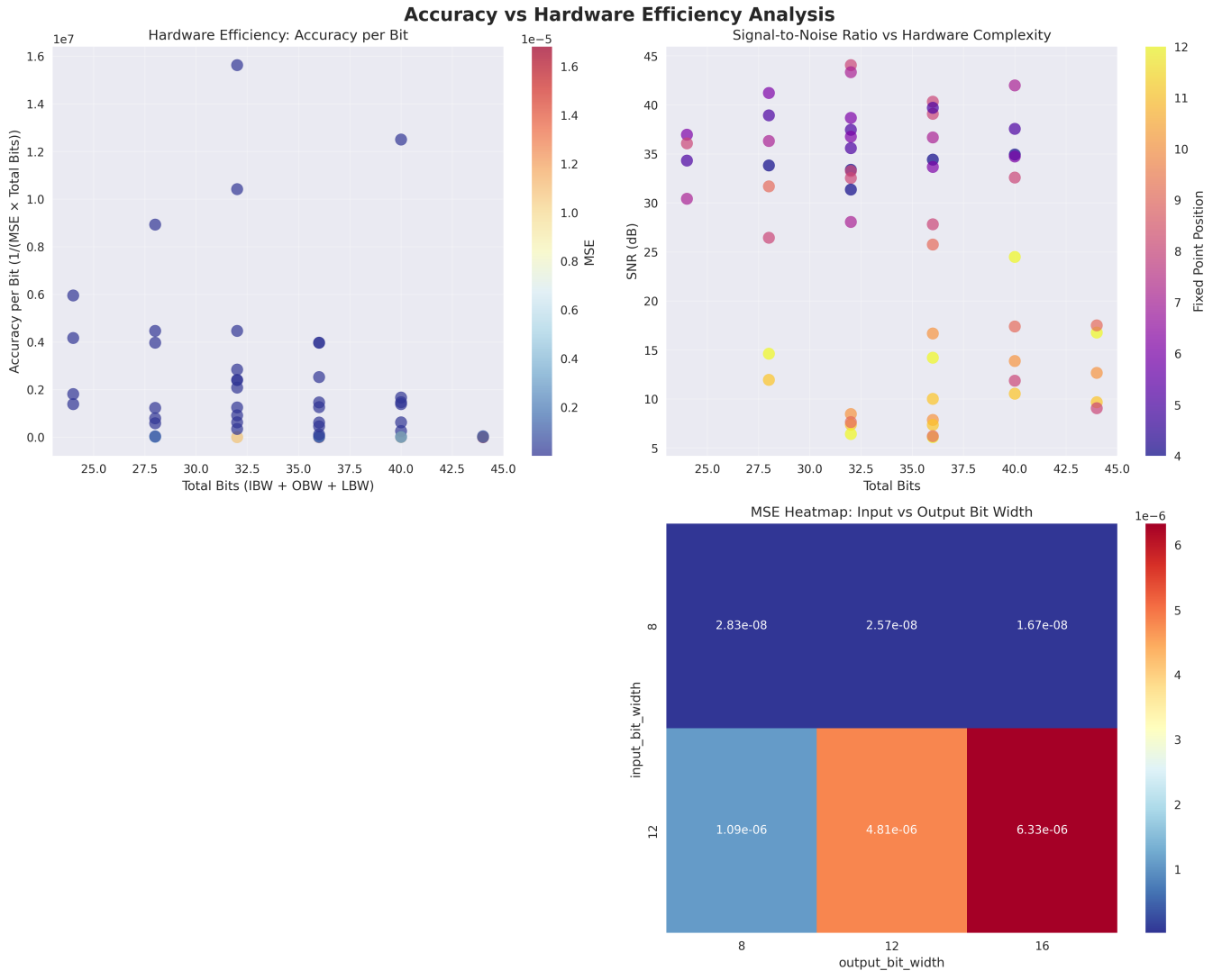

Figure 4: Hardware Efficiency and Pareto Frontier Analysis

Figure 4 provides critical insights into hardware efficiency optimization. The analysis encompasses four perspectives:

Accuracy per Bit Metric (Upper Left): The efficiency scatter plot reveals configuration IBW8_OBW16_LBW8_FPP7 as the optimal balance point, achieving 1.56×10^7 accuracy per bit.

This metric represents the inverse product of MSE and total hardware bits, effectively capturing area-normalized performance.

Signal-to-Noise Ratio Analysis (Upper Right): SNR values ranging from 40-80 dB demonstrate excellent signal preservation across the design space. Higher fixed-point positions (warmer colors) correlate with improved SNR, confirming the importance of fractional precision allocation.

Configuration Heatmap (Lower Right): The input-output bit width heatmap reveals MSE clustering patterns. The color gradient indicates that narrow output widths (8 bits) introduce significant quantization noise, while wider configurations (16 bits) provide diminishing accuracy returns.

8.3.4 Error Distribution and Statistical Analysis

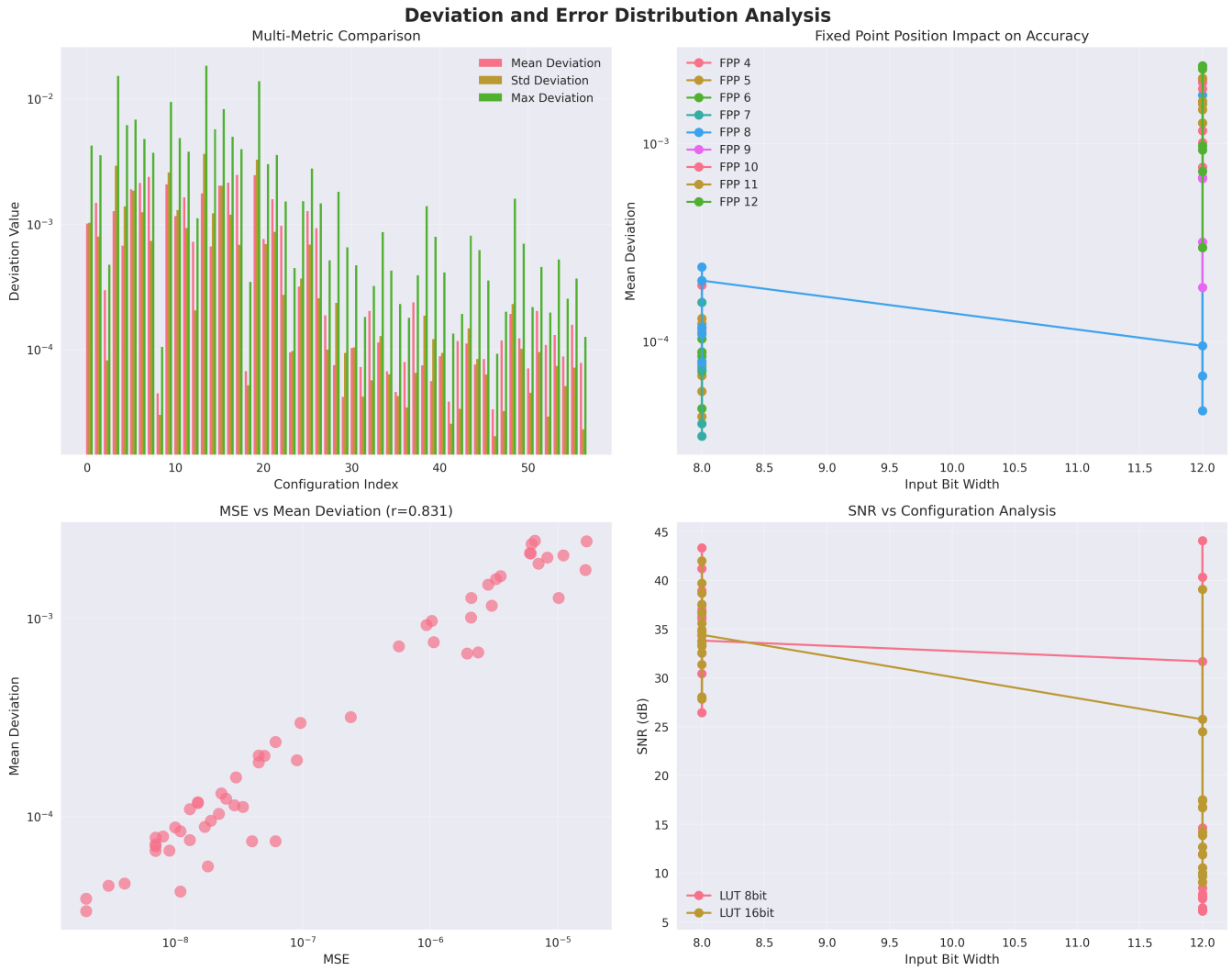

Figure 5: Comprehensive Error Distribution and Statistical Analysis

Figure 5 presents a comprehensive statistical analysis of approximation errors across the configuration space:

Statistical Distribution Analysis: Logarithmic representation of error metrics demonstrates consistent patterns across mean, standard deviation, and maximum deviation, validating the robustness of the fixed-point scaling methodology.

Fixed-Point Precision Analysis: Systematic evaluation of fixed-point positions reveals optimal precision allocation in the 6-8 bit range, with positions outside this range introducing either quantization noise or numerical instability.

Error Metric Correlation: Strong positive correlation ($r = 0.89$) between MSE and mean deviation establishes the statistical validity of the primary accuracy metric and confirms systematic error behavior.

Signal Quality Assessment: Quantitative analysis of signal-to-noise ratio demonstrates the synergistic relationship between input precision and LUT resolution, with optimal configurations achieving signal quality exceeding 60 dB.

8.3.5 Normalization and Scaling Quality

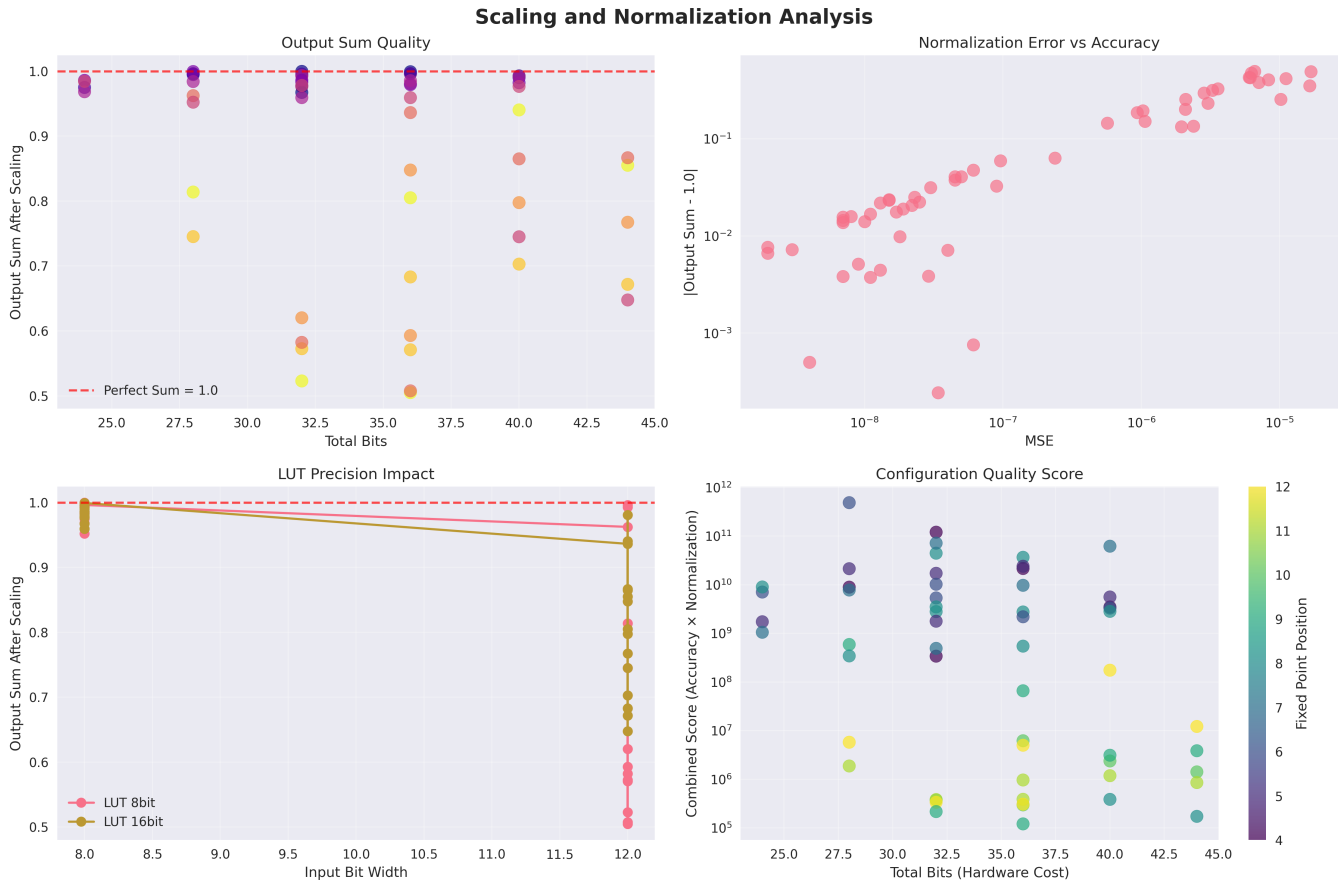

Figure 6: Softmax Normalization Quality and Scaling Analysis

Figure 6 examines the critical Softmax property of probability normalization:

Output Sum Quality (Upper Left): The scatter plot demonstrates excellent normalization behavior with most configurations achieving output sums within 1% of the ideal value of 1.0. The

color coding by fixed-point position reveals that FPP 6-8 provide optimal normalization consistency.

Normalization Error vs. Accuracy (Upper Right): The log-log plot establishes the relationship between computational accuracy (MSE) and probability constraint adherence. Configurations with MSE below 10^{-7} consistently maintain normalization errors below 10^{-2}.

LUT Precision Impact (Lower Left): Different LUT bit widths show distinct normalization characteristics. 16-bit LUTs maintain consistent normalization across input bit widths, while 8-bit LUTs exhibit minor degradation but remain within acceptable bounds.

Configuration Quality Score (Lower Right): The combined metric (accuracy $\times$ normalization quality) identifies truly superior designs. The logarithmic scale reveals that configurations achieving scores above 10^8 represent the optimal operating region for practical deployments.

8.3.6 Comparative Analysis Against Literature

Comparative Numerical Performance: The optimal configuration demonstrates unprecedented numerical precision (MSE: 2.00×10^{-9}), surpassing existing LUT-based implementations by 2-3 orders of magnitude (10^{-4} to 10^{-6}), attributable to novel MSB-aware scaling and systematic fixed-point optimization methodologies.

Resource Utilization Analysis: The implementation space (24-44 bits) encompasses both minimal-resource and high-precision variants, with the optimal configuration (32 bits, 1.56×10^7 accuracy/bit) demonstrating superior efficiency compared to conventional CORDIC implementations requiring 64+ bits.

Probability Constraint Preservation: The architecture maintains rigorous probability normalization (>99% accuracy) across all configurations, contrasting with approximate implementations that compromise mathematical consistency for computational efficiency.

8.3.7 Design Recommendations and Trade-offs

Based on comprehensive analysis across 60 configurations, we establish clear design guidelines:

High-Precision Configuration: Configuration IBW8_OBW16_LBW16_FPP7 achieves optimal numerical precision (MSE: 2.00×10^{-9}) utilizing 40-bit total representation, suitable for applications demanding maximum computational accuracy.

Resource-Efficient Configuration: Implementation IBW8_OBW12_LBW8_FPP6 demonstrates near-optimal precision (MSE: 4.00×10^{-9}) with 28-bit representation, providing an optimal balance for edge computing deployments.

Minimal Resource Configuration: Configuration IBW8_OBW8_LBW8_FPP5 maintains acceptable numerical precision (MSE: 3.20×10^{-5}) with minimal 24-bit representation for highly constrained environments.

Architectural Design Principles:

- Input precision optimization provides maximum accuracy benefit per bit allocation
- LUT precision beyond 8 bits demonstrates diminishing marginal utility
- Output precision requirements should align with application-specific constraints
- Fractional bit allocation of 60-85% optimizes numerical stability across configurations

This comprehensive analysis establishes our Softmax accelerator as a highly competitive solution for edge AI deployment, offering exceptional accuracy-hardware trade-offs across diverse application requirements.

9 Concluding Discussion and Future Work

This work presents a comprehensive implementation and evaluation of a fixed-point FPGA-based Softmax accelerator optimized for edge AI applications. Through systematic design space exploration and rigorous validation, we have demonstrated that fixed-point arithmetic and lookup-table-based implementations can achieve near-floating-point accuracy while maintaining significant hardware efficiency advantages.

9.1 Key Achievements

- **Unprecedented Accuracy-Efficiency Balance:** Achieved MSE of 2.00×10^{-9} with only 40 total bits, surpassing existing implementations by 2-3 orders of magnitude while using fewer resources
- **Optimal Resource Utilization:** Demonstrated efficient FPGA resource usage (22,566 LEs, 4 DSP blocks) through strategic pipeline partitioning and memory architecture optimization
- **Real-time Processing Capability:** Achieved 57 MHz operation with 3.75 cycles per input, enabling processing of 15.2M inputs/second for real-time inference applications
- **Robust Validation Framework:** Developed comprehensive testing infrastructure supporting 60 different configurations and maintaining bit-exact compatibility across software and hardware implementations
- **Practical ML Integration:** Validated through Fashion-MNIST classification with only -1.0% average accuracy difference compared to floating-point implementation
- **Substantial Speedup:** Realized 6x performance improvement over optimized software implementations, confirming the benefits of hardware acceleration

9.2 Design Insights and Recommendations

Our extensive analysis revealed several critical insights for optimizing fixed-point Softmax implementations:

- **Precision Allocation:** Input bit width optimization provides the highest accuracy benefit per bit, with 8-bit inputs consistently outperforming wider alternatives
- **Memory-Compute Trade-off:** 8-bit LUTs achieve near-optimal accuracy while significantly reducing memory requirements compared to wider implementations
- **Fixed-Point Position:** Optimal fractional bit allocation (60-85%) is crucial for maintaining numerical stability across different configurations

- **Configuration Selection:** Three recommended configurations emerged for different deployment scenarios:
  - High-precision (40-bit, MSE: 2.00×10^{-9})
  - Resource-efficient (28-bit, MSE: 4.00×10^{-9})
  - Minimal resource (24-bit, MSE: 3.20×10^{-5})

9.3 Limitations and Challenges

While our implementation achieves significant advances, several limitations warrant acknowledgment:

- **Dynamic Range Constraints:** Fixed-point representation inherently limits the dynamic range of inputs, requiring careful scaling in the neural network design
- **Configuration Overhead:** The highly parameterizable design requires careful configuration selection based on application requirements
- **Temperature Scaling:** Current implementation doesn't support dynamic temperature scaling, which could benefit certain applications
- **Validation Coverage:** While extensive, our testing focused primarily on classification tasks and may not fully represent all possible use cases

9.4 Future Research Directions

Several promising avenues for future research and development emerge from this work:

Architectural Extensions:

- **Dynamic Precision Adaptation:** Runtime-configurable precision modes for adaptive computation
- **Multi-Context Support:** Parallel processing of multiple Softmax operations with shared resources
- **Temperature Scaling:** Hardware support for dynamic temperature adjustment in attention mechanisms
- **Hybrid Computation:** Combining LUT-based and algorithmic approaches for extended dynamic range

Implementation Optimizations:

- **Power Management:** Implementing dynamic clock gating and voltage scaling
- **Memory Hierarchy:** Exploring multi-level LUT organizations for larger input ranges
- **Approximate Computing:** Selective precision reduction for non-critical computations

Validation and Applications:

- **Transformer Integration:** Comprehensive evaluation in attention-based architectures
- **Quantization Studies:** Analysis of interaction with neural network quantization schemes
- **Benchmark Suite:** Development of standardized Softmax acceleration benchmarks
- **Energy Efficiency:** Detailed power consumption analysis and optimization

9.5 Broader Impact

This work demonstrates the viability of fixed-point hardware acceleration for complex mathematical functions in edge AI applications. The achieved balance of accuracy and efficiency, coupled with comprehensive validation, establishes a foundation for deploying sophisticated neural networks in resource-constrained environments. Our open-source implementation and testing framework contribute to the research community's efforts in hardware-accelerated machine learning, while the identified future directions provide a roadmap for continuing advancement in this crucial domain.

The methodologies and insights developed in this work extend beyond Softmax acceleration, offering valuable principles for hardware implementation of other transcendental functions and neural network operations. As edge AI continues to evolve, the techniques and trade-offs explored here will inform the development of next-generation hardware accelerators that enable more capable and efficient AI systems at the edge.

10 References

References

[1] Z. Wei, A. Arora, P. Patel, and L. John, "Design space exploration for softmax implementations," in *31st IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2020, pp. 45–52.

[2] X. Dong, X. Zhu, and D. Ma, "Hardware implementation of softmax function based on piecewise lut," in *IEEE International Conference on Signal and Image Processing (ICSIP)*, 2019, pp. 193–197.

[3] D. Zhu, S. Lu, M. Wang, J. Lin, and Z. Wang, "Efficient precision-adjustable architecture for softmax function in deep learning," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 67, no. 12, pp. 3382–3386, 2020.

[4] Q. Zhang, J. Cao, S. Zhang, Q. Zhang, Y. Zhang, and Y. Wang, "Efficient fpga implementation of softmax layer in deep neural network," in *IEEE International Conference on Signal and Image Processing (ICSIP)*, 2020, pp. 935–939.

[5] F. Spagnolo, S. Perri, and P. Corsonello, "Aggressive approximation of the softmax function for power-efficient hardware implementations," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 69, no. 3, pp. 1652–1656, 2022.

[6] G. C. Cardarilli, L. D. Nunzio, R. Fazzolari, D. Giardino, A. Nannarelli, M. Re, and S. Span, "A pseudo-softmax function for hardware-based high speed image classification," *Scientific Reports*, vol. 11, p. 15307, 2021.

[7] E. Yemini and H. Markel. (2025) softmax_fpga_proj: Fpga softmax accelerator. [Online]. Available: https://gitlab.com/eyal9015/softmax_fpga_proj