OpenLLM: Modular and Scalable zkSNARKs for Verifiable LLM Inference

Yunbo Yang*, Yupeng Ren*, Changtong Xu, Rui Zhang, Xuanming Liu, Jin Tan, Tao Wei, Bingsheng Zhang, and Kui Ren✉

***Abstract*—Large language model (LLM) is increasingly deployed as a remote service, where users rely on third-party servers to perform computation. However, such settings introduce critical integrity concerns, as an untrusted server may deviate from the prescribed computation, skip expensive operations, or return incorrect results, while users lack practical approaches to verify execution correctness. Ensuring the correctness of LLM inference under untrusted execution remains a fundamental challenge. Zero-knowledge proofs (ZKPs) provide a principled approach verifying computation correctness, but applying them to LLM inference remains challenging. Modern LLMs involve a large number of non-linear operations and require modeling real-valued computation in finite fields, introducing substantial computational and memory overhead and potential loss of numerical precision. Moreover, the large scale of LLMs makes end-to-end verification difficult to scale, limiting the practicality of existing approaches.**

This paper presents OpenLLM, an efficient and modular system for verifiable LLM inference. Our key idea is to decompose large-scale LLM inference into a set of reusable atomic operators, each equipped with efficient ZKP protocols, enabling scalable verification at the operator level. Based on this abstraction, we design succinct non-interactive zero-knowledge proof constructions for representative non-linear functions, which can be composed into end-to-end inference pipelines independent of model architectures.

We further evaluate OpenLLM across operator-level performance, end-to-end inference, layer-wise scaling, larger models, and approximation accuracy. The results show that OpenLLM achieves smaller proof sizes, lower verification cost, and improved numerical fidelity while scaling from individual operators to full-model inference. Compared with state-of-the-art interactive protocols, OpenLLM eliminates communication overhead through a fully non-interactive design while maintaining competitive efficiency. Building on this operator-level efficiency, it further enables a scalable and modular framework for end-to-end verifiable LLM inference, outperforming prior end-to-end approaches.

Yunbo Yang is with the State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China, and also with the Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security, Hangzhou, China. E-mail: yyb9882@gmail.com.

Yupeng Ren is with the State Key Laboratory of Cyberspace Security Defense, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China, and also with the School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China. E-mail: renyupeng@iie.ac.cn.

Changtong Xu, Jin Tan, and Tao Wei are with Ant Group, Hangzhou, China. E-mails: xuchangtong.xct@antgroup.com, tanjin.tj@antgroup.com, and lenx.wei@antgroup.com.

Rui Zhang is with the State Key Laboratory of Cyberspace Security Defense, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China. E-mail: zhangrui@iie.ac.cn.

Xuanming Liu, Bingsheng Zhang, and Kui Ren are with Zhejiang University, Hangzhou, China. E-mails: hinsliu@zju.edu.cn, bingsheng@zju.edu.cn, and kuiren@zju.edu.cn.

This work was supported by the Postdoctoral Fellowship Program of CPSF (Grant No. GZB20250407), the Zhejiang Provincial Natural Science Foundation of China (Grant No. LMS26F020011), and the China Postdoctoral Science Foundation (Grant No. 2025M771571, 2026T190432). (Corresponding author: Kui Ren✉. Yupeng Ren and Yunbo Yang contributed equally.)

I. Introduction

Large Language Models (LLMs) have achieved remarkable success in natural language generation, powering applications ranging from virtual assistants and code generation to enterprise-level AI services. Due to the high cost of training and deploying these models, most LLMs are offered via APIs, where model inference is performed on remote servers. However, such black-box access introduces critical trust concerns: inference is executed by an untrusted server that may deviate from the prescribed computation, skip expensive operations, or return incorrect results, while users have no practical means to audit or verify the execution process. Moreover, modern LLMs are highly complex, consisting of deep architectures with a large number of parameters and layers, which makes it challenging for users to validate the correctness of inference in practice. As LLM services are increasingly deployed in real-world applications, the lack of verifiability poses a fundamental barrier to building trustworthy AI systems.

Zero-Knowledge Proofs (ZKPs) have emerged as an essential cryptographic technique for verifying the integrity of LLM inference without revealing private internal model parameters. In LLM services, ZKPs allow a service provider to generate a cryptographic proof π that convinces a verifier that the claimed output y is indeed the result of executing the correct model function f on a public user input x and secret model weights w. In addition, the proof π leaks no additional information about the model parameters. In practical deployments, such proofs can be requested on demand to enforce service-level guarantees, detect misbehavior, and enable verifiable AI services under untrusted execution environments.

Yet, designing practical ZKPs for LLM inference remains challenging since it requires efficiently generating proofs for the diverse computational layers involved in LLMs, including matrix multiplication, attention mechanisms, activation functions, and normalization. This challenge arises mainly for two reasons: (1) LLMs perform computations over real values, whereas ZKPs operate over finite fields. Mapping real-valued operations to field arithmetic inherently introduces approximation complexity and additional system overhead, and (2) non-linear functions such as ReLU and Softmax are particularly difficult to handle within ZKP frameworks, as they involve square root, exponentiation, and division. Representing

or approximating them with addition and multiplication gates significantly increases circuit depth, computational overhead, and memory consumption. These challenges will make deploying a verifiable LLM inference system impractical.

Representative systems have explored different ways to address the two fundamental challenges. First, to bridge the gap between computation over real values and finite fields, Hao et al. [3] adopt fixed-point representations together with truncation proofs to ensure correctness during scaling and rounding. However, this approach reduces model precision and introduces additional prover cost, which accumulates across layers in large models. Meanwhile, ZIP [1] targets high numerical precision by encoding all possible computations into large tables and using cryptographic operations over elliptic curves to generate proofs, which incurs significant memory overhead and limits scalability in realistic deployment environments. Second, to address the inherent complexity of non-linear functions, recent works adopt different lookup-table-based techniques, either decomposing large tables into multiple smaller ones or directly using a single comprehensive table. For example, Hao et al. [3] use a small fixed lookup table and decompose large computations into multiple chunks, while zkGPT [2] precomputes the behavior of non-linear functions and verifies their correctness through extremely large lookup tables, resulting in substantial memory consumption and prover cost.

Although these approaches address some of the above challenges, they often tightly couple numerical representations, non-linear operator semantics, and proof mechanisms into a single construction. As a result, even small changes to the inference pipeline, such as modifying a non-linear operator or adjusting precision, typically require redesigning an entirely new proof protocol. From a system perspective, this tight coupling limits reuse and composition across different models and deployment settings. In contrast, if these operators can be cleanly decomposed and designed in a modular manner, different inference pipelines can be supported by simply recombining each operator. This modular design significantly simplifies system construction, improves scalability, and enables practical deployment.

In this context, Hao et al. [3] present a state-of-the-art composable ZKP framework for non-linear functions, representing an important step toward modular verification. However, their design still exposes a fundamental gap between composability and practical deployment: (1) it relies on extensive interaction between the prover and the verifier, introducing significant communication overhead and making it unsuitable for latency-sensitive or large-scale deployments. In typical LLM service settings, inference follows a query-response paradigm, where users expect immediate outputs, making multi-round interaction impractical and highlighting the necessity of non-interactive proofs, (2) fixed-point representations lead to reduced numerical precision and introduce additional proof overhead to ensure correctness during scaling and truncation, and (3) iterative algorithms for non-linear operations significantly increase prover cost and system complexity, making efficient proof generation difficult in practice, especially when proofs need to be generated on demand for each inference request.

These limitations indicate that existing approaches do not provide a practical and modular zero-knowledge proof framework that can be instantiated as a succinct, non-interactive proof with fast verification. Without these properties, deploying verifiable LLM inference in real-world systems remains challenging, which naturally raises the question.

Can we design a modular and scalable approach to ensure the integrity of LLM inference under untrusted servers, by decomposing complex computations into reusable units that achieve (1) fully non-interactive verification with minimal communication overhead, (2) high numerical accuracy without additional approximation costs, and (3) practical proof generation and verification overhead, enabling real-world deployment for verifiable LLM services?

A. Our Contribution

In this paper, we propose OpenLLM, a zero-knowledge succinct non-interactive argument of knowledge (zkSNARK) framework for non-linear functions in LLM inference, to answer the question. Our contributions are listed as follows.

A system-level decomposition framework for verifiable LLM inference. We propose a system-level decomposition framework that models LLM inference as a set of reusable operators, which serve as the basic units for verifiable computation across different architectures. This abstraction eliminates the need to manually translate each operator into low-level circuit constraints, significantly reducing proving complexity. More importantly, it provides a practical system abstraction for constructing verifiable LLM inference pipelines, enabling modular composition and reuse across different models and deployment settings. The resulting design enables modularity and reusability, supports parallel and distributed proof generation, and is suitable for scalable and real-world deployment of verifiable LLM systems.

Non-interactive, efficient, and high-precision proof components for non-linear operators. Building upon this abstraction, we design efficient zkSNARK constructions for representative non-linear operators in LLM inference, including GeLU, Softmax, and normalization. Our approach follows a table-lookup decomposition strategy, which reduces large lookup tables into fixed, small tables while preserving correctness and soundness. This design enables fully non-interactive verification while reducing proof generation and verification overhead and maintaining high numerical accuracy without additional approximation costs. It makes non-linear verification practical for large-scale and resource-constrained LLM inference settings.

Implementation and Evaluation. To comprehensively evaluate OpenLLM, we adopt a multi-level evaluation methodology that reflects our system design. Specifically, we evaluate (1) individual operators to validate efficiency and numerical accuracy, (2) composed layers to assess composability and overhead, and (3) end-to-end inference pipelines to capture system-level performance in realistic settings.

We implement OpenLLM and conduct extensive experiments across these settings. The results show that OpenLLM achieves smaller proof sizes, lower verification cost,

and improved numerical fidelity, while maintaining scalability from operator-level computations to end-to-end model inference. Compared with state-of-the-art interactive protocols for operator-level verification, OpenLLM eliminates communication overhead through a fully non-interactive design while maintaining competitive efficiency. Building on this operator-level foundation, our system further enables a scalable and modular framework for end-to-end verifiable LLM inference.

B. Related Work

Verifiable machine learning aims to ensure the correctness of outsourced training and inference, allowing clients to delegate computations to untrusted servers while still verifying their correctness. Existing approaches can be broadly divided into two categories: (1) verifiable computation (VC) and zero-knowledge proofs (ZKPs) over finite fields [5], [7], [8]; and (2) trusted execution environments (TEEs) [9], which execute native floating-point operations inside secure enclaves. TEE-based methods achieve faster performance but depend on hardware trust and remain susceptible to side-channel attacks.

Research on verifiable ML covers both classical and deep models, including decision trees, linear/SVMs, and deep neural networks (DNNs) [13] [10] [11] [12]. One line of work focuses on improving proof efficiency, such as reducing circuit size and proving or verification time, particularly for non-linear computations. These approaches approximate or precompute activation functions using fixed-point arithmetic or large lookup tables [12, 30, 41, 57]. Approximation-based methods replace non-linear activations with low-degree polynomials or piecewise-linear functions [14] [15], effectively reducing proving time and circuit complexity, but at the cost of numerical precision or additional table-storage overhead.

The second direction aims to enhance proof precision for non-linear functions, typically under fixed-point arithmetic. These systems design linear building blocks (gadgets) for each activation [13], apply bit decomposition [16] [12], or employ interactive protocols that delegate complex activation computations to clients [17], thus avoiding large circuits. While these techniques preserve higher-precision semantics for non-linear computations, they generally lead to larger circuits and incur high proving and communication costs, often supporting only a limited set of activation types.

Recent advances in verifiable LLM inference leverage lookup tables to efficiently prove non-linear functions, with ZKGPT [2], ZIP [1], and [3] representing the most notable approaches. ZKGPT employs large-scale lookup tables combined with the Lasso protocol [25], resulting in high memory consumption and unsound range proofs for certain operations. ZIP improves numerical precision by adopting IEEE-754 floating-point representations but incurs massive storage requirements and relies on KZG commitments, which limit scalability and distributed proving. Hao et al. [3] reduce memory usage via digital decomposition but require recursive computations and per-input VOLE interactions, leading to significant computational and communication overhead as well as reduced precision due to fixed-point truncation.

II. Preliminaries

Notations. We use $\langle i \rangle$ to denote the binary encoding of i, and $\lfloor u \rceil$ to denote the approximate value of the real value u. For a polynomial $A(\mathbf{X})$, we define $[A]$ as a public commitment to $A(\mathbf{X})$. In addition, the definition of interface is shown in TABLE VI of Appendix A.

Real-value Representation. For a real number x, a scale $s = \frac{a}{b}$, and a large prime p, the real-value quantization is as follows.

- $\mathsf{R2F}(x, s = \frac{a}{b}, p) \to q_x$: Output $q_x = \lfloor x \cdot \frac{b}{a} \rceil \bmod p$.
- $\mathsf{F2R}(q_x, s = \frac{a}{b}, p) \to x$: If $\frac{p+1}{2} \le q_x \le p-1$, output $x = (q_x - p) \cdot s$, and if $0 \le x \le \frac{p-1}{2}$, output $x = q_x \cdot s$.

We denote the scale of a real value x as $s = \frac{a}{b}$, where a and b are public parameters, and its integer part as q_x, which is kept secret. For a private real number x, it is sufficient to hide only one of s or q_x, instead of both, to reduce the overall cost.

Matrix Quantization. In zero-knowledge proofs for machine learning [21], all elements in the same matrix share the same scale, and the scale is kept public at all times. For a matrix A and each element $A_{i,j}$, it performs the following steps:

1) Compute the maximal x_{max} and minimal x_{min}.
2) Define width k. In the LLM setting, it is equal to the size of the predefined large prime p.
3) Select a and b, and compute scale $\frac{a}{b} \approx s = \frac{x_{\mathsf{max}} - x_{\mathsf{min}}}{2^k - 1}$, in which k is the length of integer.
4) Compute quantization $q_{A_{i,j}} = \lfloor \frac{A_{i,j}}{s} \rceil$.

A. Large Language Model

Many popular LLMs like GPT adopt the Transformer with multi-head self-attention, which consists of multiple layers and recursively executes the Transformer layer by layer. Each layer applies a fixed sequence of modules, including multi-head self-attention, feed-forward networks (FFN), residual connections, and layer normalization.

From a proof-construction perspective, we adopt a modular abstraction of Transformer execution, treating each type of operator as an independent component. The multi-head attention block computes query, key, and value projections, applies scaled dot-product attention with softmax function, and then performs output projection. The feed-forward block applies two linear transformations with a non-linear activation function (e.g., GELU). The layer normalization block performs token-wise normalization followed by an affine transformation, while residual connections and embedding layers involve only basic addition and scaling.

All these LLM operators can be reduced to combinations of linear or non-linear operations. The linear operation includes matrix multiplication, while the non-linear function includes softmax, GELU and layer normalization.

B. Universal zkSNARK

Universal zkSNARKs, such as Marlin [8] and Plonk [7], provide a flexible and powerful proving system. The universal property means that the setup procedure is executed only once and can be reused to generate proofs for any computation within a certain class. A universal zkSNARK is defined by three core algorithms:

- $\mathsf{zkSNARK.Setup}(1^\lambda) \rightarrow \mathsf{srs}$: Given a security parameter λ, this algorithm outputs a universal structured reference string srs.
- $\mathsf{zkSNARK.Prove}(\mathsf{srs}, C, \vec{x}; \vec{w}) \rightarrow \pi$: Using srs, a circuit C, a public input vector $\vec{x}$, and a private witness $\vec{w}$, it generates a proof π.
- $\mathsf{zkSNARK.Verify}(\mathsf{srs}, \pi, \vec{x}) \rightarrow \mathsf{accept/reject}$: Given srs, a proof π, and the public input $\vec{x}$, this algorithm outputs either accept or reject.

When evaluating a witness $\vec{w}$ and a public input $\vec{x}$ on a circuit C, one obtains an extended witness $\vec{\tilde{w}}$ that includes all intermediate wire values. Universal zkSNARKs must also satisfy the following properties:

- **Completeness.** An honest prover always convinces an honest verifier with probability 1.
- **Knowledge Soundness.** For any malicious prover, there exists an efficient extractor that can recover the private witness, even if the prover manages to convince the verifier.
- **Zero-knowledge.** The proof reveals no information about the private witness beyond the validity of the statement.

Design Paradigm of Universal zkSNARK. Bünz et al. [6] observe that a modern zero-knowledge succinct non-interactive argument of knowledge (zkSNARK) can be divided into three layers: a Polynomial Interactive Oracle Proof (PIOP), a Polynomial Commitment Scheme (PCS), and the Fiat–Shamir (FS) transformation. In the PIOP layer, the prover constructs a set of polynomials to show that an instance–witness pair satisfies the given constraints, while the verifier only has oracle access to these polynomials. Then, the PCS layer replaces this oracle access with commitments and opening proofs: the prover commits to the polynomials, responds with evaluations and opening proofs based on verifier challenges, and the verifier checks their validity. Finally, the FS transformation turns this interactive argument into a non-interactive proof.

C. Multivariate Polynomials and Sumcheck

We briefly review the multivariate sumcheck protocol, which serves as a core building block in our construction. Sumcheck enables a prover to convince a verifier of the correctness of a sum of a multivariate polynomial over the Boolean hypercube through a sequence of univariate polynomial checks. Basic PIOPs used in this work, including Logup lookup arguments [18], zero checks, product checks in HyperPlonk [5], and matrix operations, can be reduced to instances of the multivariate sumcheck protocol. Due to space constraints, we defer the formal definitions of multilinear extensions, vector and matrix polynomial representations, as well as detailed descriptions of these reductions to the multivariate sumcheck protocol in Appendix A.

D. Threat and Security Model

Threat Model. In our setting, the server is assumed to be fully untrusted and may behave arbitrarily, including deviating from the prescribed model computation, skipping expensive operations, or returning incorrect inference results. The client interacts with the server through a query-response interface and has no visibility into the internal execution process.

We assume the client is computationally bounded and follows the protocol, but may attempt to learn additional information about the model parameters from the proof. The server holds the model parameters and is responsible for generating both the inference result and a proof of correctness.

Security Goal. Our goal is to enable verifiable LLM inference that ensures both correctness of computation and privacy of the model. Specifically, we require the following properties:

- **Completeness.** An honest server, following the prescribed computation, can always produce a valid proof that is accepted by the client.
- **Soundness.** A malicious server cannot convince the client to accept an incorrect inference result, except with negligible probability.
- **Zero-Knowledge.** The proof reveals no additional information about the model parameters beyond what is inherently leaked by the output.

In addition, we note that our framework is constructed in a modular manner, where each operator is instantiated as an independent zkSNARK component satisfying completeness, soundness, and zero-knowledge. Since the overall LLM inference can be expressed as a composition of these operators, the security of the entire system follows from the standard composability of zkSNARKs under sequential composition.

III. Technical Overview

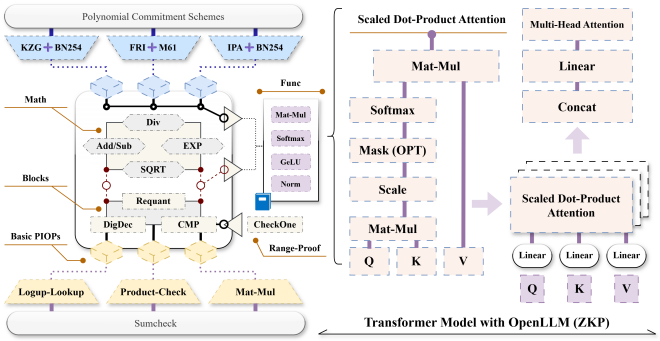

Fig. 1: Workflow of OpenLLM

As shown in Fig. 1, we present the workflow of OpenLLM through a top-down decomposition of LLM inference. Starting from the model architecture, high-level execution is decomposed into reusable operators (e.g., Softmax, GELU, and normalization), which are further reduced to mathematical functions and finally expressed in terms of proof-level primitives. This hierarchical decomposition forms the foundation of our modular verification framework and enables efficient and scalable verification of LLM inference.

Typical LLM architectures consist of linear and non-linear components. The linear component is primarily instantiated as matrix multiplication, while the non-linear component includes operators such as Softmax, GELU, and normalization. Building on this structure, we design an efficient and modular non-interactive zero-knowledge proof system that supports

composable operator-level verification and scalable end-to-end inference in real-world settings.

High-level Overview of OpenLLM. Our framework can be summarized into five layers. This layered structure follows the above decomposition, enabling a clean separation between arithmetic representation, proof construction, and application-level operators, which is critical for modularity and scalability in real-world LLM inference pipelines.

- **(1) Real-Value Quantization.** As the heart of OpenLLM, the real-value quantization defines a mapping for mutual conversion between reals and their field representation. It bridges a gap between machine learning over real numbers and ZKPs over a finite field.
- **(2) Basic PIOPs.** The sumcheck protocol is the most essential component of this layer, as productCheck, the LogUp lookup protocol, and matrix multiplication are all reduced to the sumcheck protocol. Meanwhile, all operations in layers (3), (4), (5) are reduced to this layer.
- **(3) Fundamental building blocks.** As the basis of OpenLLM, this level consists of three operations: digital decomposition, comparison, and requantization.
- **(4) Mathematical functions.** Building upon the building blocks, we design efficient zkSNARK protocols for complex mathematical functions, including exponential, division, square root, and addition.
- **(5) LLM applications.** With above ZK proofs for mathematical functions and building blocks, we naturally construct zkSNARK protocols for LLM non-linear functions, including GELU, Softmax, and Normalization.

Real-Value Quantization. Most algorithms represent numbers in fixed-point form [2], [3], [13], [19], [20], which encodes a real value with a binary scale 2^s and decodes with 2^{-s}. Both the recovery and truncation introduce additional prover complexity and accuracy loss. Therefore, we adopt a real-value quantization scheme tailored for OpenLLM. In brief, for a real number x and a large prime p, we choose a public scale $s = \frac{a}{b}$, and choose a private element q_x such that $q_x = \lfloor x \cdot \frac{b}{a} \rceil \bmod p$.

zkSNARK Design Paradigm for LLM Operators. Motivated by the row-wise execution of LLM operations, we treat the input to each invocation as a row of a matrix. In contrast to Hao et al. [3], which relies on VOLE, we build zkSNARKs based on a PIOP and a polynomial commitment scheme (PCS) to evaluate multiple elements in batch as follows. (1) Represent inputs $x_0, x_1, \cdot, x_{n-1}$ as a multivariate polynomial $f(\mathbf{X})$, (2) design tailored PIOP for each operator and reduce it to basic PIOPs (i.e., sumcheck, prodcheck, lookup, and matrix multiplication), and (3) according to the security and efficiency requirements, apply PCS such as KZG [22], IPA [23], or FRI-based PCS [24] on each polynomial. Furthermore, all optimizations of basic PIOPs and PCS can be applied to the proposed zkSNARK to further reduce the proof generation time, proof size, and verification time.

(1) Design of Building Blocks. As the most important basis of our framework, the building blocks (Section 4) consist of digital decomposition, comparison, and requantization.

Digital Decomposition. To avoid prohibitively large lookup tables, we decompose each input x into k small digits $x^{(0)}, \ldots, x^{(k-1)}$ with bit length d. Given a vector $\vec{a} = (x_0, \ldots, x_{n-1})$ encoded by $A(\mathbf{X})$ with $A(\langle i \rangle) = x_i$, this operation outputs an encoding $B(\mathbf{X}, \mathbf{Y})$ such that $B(\langle i \rangle, \langle j \rangle) = x_i^{(j)}$, $x_i = \sum_{j=0}^{k-1} x_i^{(j)} 2^{jd}$, and $x_i^{(j)} \in \{0,1\}^d$ for all $i \in [0, n-1]$, $j \in [0, k-1]$. After that, to prove the correctness of digital decomposition, the prover (1) applies the lookup protocol to show that each $x_i^{(j)}$ is bounded by $[0, 2^d - 1]$, and then (2) applies the sumcheck protocol to prove each $x_i^{(j)}$ is the correct digital decomposition of x_i such that $x_i = \sum_{j=0}^{k-1} x_i^{(j)} 2^{jd}$.

Comparison. We prove the predicate $x < c$ over $\mathbb{F}_p$, where x is the secret value and c is public. Followed by the work of [3], instead of comparing the whole words, we decompose x and c into short d-bit digits and verify a sequence of small comparisons on those digits, such that $\mathbf{1}\{x < c\} = \mathbf{1}\{x^{(1)} < c^{(1)}\} + \mathbf{1}\{x^{(1)} = c^{(1)}\} \cdot \mathbf{1}\{x^{(0)} < c^{(0)}\}$, for a two-digit split $x = x^{(1)} \| x^{(0)}$ and $c = c^{(1)} \| c^{(0)}$. More generally, the relation can be recursively invoked for $x = x^{(k-1)} \| \cdots \| x^{(0)}$ and $c = c^{(k-1)} \| \cdots \| c^{(0)}$.

To reduce the size of the lookup table, we build upon the optimization in [3]. Furthermore, the main difference is that the prover first encodes several inputs $x_0, x_1, \ldots, x_{n-1}$ as a polynomial $A(\mathbf{X})$, and then uses the proposed zkSNARK protocol to check if $x_i < c$ in a batch. In addition, one can invoke Checkone to check if every x_i in a vector $\vec{x}$ is less than c.

Requantization. At a high level, for each input x_i encoded by $A(\mathbf{X})$ at scale $s_1 = \frac{a_1}{b_1}$, the protocol outputs a value y_i encoded by $B(\mathbf{X})$ at scale $s_2 = \frac{a_2}{b_2}$ and a small remainder r_i that satisfy $a_1 \cdot b_2 \cdot x_i = a_2 \cdot b_1 \cdot y_i + r_i, \quad 0 \le r_i < a_2 \cdot b_1$. As $a_2 \cdot b_1$ is public, then the prover invokes the comparison protocol to ensure that $r_i, \quad 0 \le r_i < a_2 \cdot b_1$. In addition, if $a_2 \cdot b_1$ is small enough, then the prover can instead invoke a single lookup protocol on table $L = [0, a_2 \cdot b_1 - 1]$.

(2) Design of Mathematical Functions. Building on the developed building blocks, we design efficient ZK proofs for complex mathematical functions (Section 5), including square root, exponential, division, and addition.

Square Root. ZKGPT applies a range check to verify the correctness of the square root operation by enforcing the relation: $(2q_y - 1)^2 \le 4q_x \le (2q_y + 1)^2 - 1$, where $q_y = \mathsf{round}(\sqrt{q_x})$. However, this approach is potentially unsound due to possible overflows during the squaring operation. Specifically, when squaring a number near the field modulus, the result may wrap around and fall into an incorrect range (e.g., exceeding $(p+1)/2$ and being interpreted as a negative number). To mitigate this issue, we enhance the protocol with additional comparison constraints to ensure that (1) the squared value does not exceed the upper bound of the field $\frac{p-1}{2}$, and (2) the input q_x is non-negative before taking the square root. These additional checks guarantee soundness and prevent incorrect witness from passing verification.

Exponentiation. Instead of storing all possible input-output pairs as in ZKGPT, which relies on a large Lasso-style lookup table and incurs significant overhead, we adopt a more efficient approach based on digit decomposition. Specifically, we decompose a large input x into a sequence of smaller digits, construct a compact lookup table for all possible com-

binations of small-digit pairs, and apply the lookup protocol individually to each digit. The final result is then reconstructed using a product check across all digit outputs. Furthermore, our scheme explicitly handles the case where x is negative, ensuring correctness and consistency across signed inputs.

Division. To handle $z = \frac{x}{y}$, we constrain $z \cdot y + r = x$ with $r < y$, and additionally incorporate sign handling for both x and y, as well as correct behavior when $x < y$. Compared to the complex range-check-based division in ZKGPT and the recursive algorithm in Hao et al., our method achieves better efficiency while avoiding the unsoundness caused by overflows in range-check-based schemes.

Addition. To add two values represented under different scales, the protocol first invokes the requantization protocol to align both inputs to a common target scale and then performs addition under that unified scale. This ensures consistency in fixed-point arithmetic when handling values with heterogeneous precisions.

(3) Design of LLM Functions. Relying on the above ZK proofs for mathematical functions as well as our building blocks, we construct ZK protocols for LLM non-linear functions (Section 6), i.e., ReLU, normalization, and softmax.

GELU. We approximate the GELU function using $\mathsf{GELU}(x) = 0.5x(1+\tanh(\sqrt{2/\pi}(x+0.0447x^3)))$. The $\tanh$ operator is decomposed into exponentiations and divisions, while the cubic term is evaluated via repeated multiplication. Throughout the computation, we explicitly enforce scale consistency and use requantization to handle intermediate scales.

Softmax. To compute $\mathsf{softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$, we first apply comparison and product check to constrain the maximum value of each vector, then apply exponentials and avoid overflow for each value. Finally, the normalization step is performed with the division protocol.

Layer Normalization. For the mean μ, we use the sumcheck protocol to check the sum of each vector, while for the variance σ, we use the comparison protocol to check that $(x_i - \mu)^2$ is bounded, then use the sumcheck protocol. Finally, we compute the final result of layer normalization with affine parameters. Throughout the computation, we explicitly enforce scale consistency and use requantization to handle intermediate scales.

IV. Building Blocks

The building block components serve as the foundational elements for non-linear mathematical functions. We assume that all operations are performed over a vector with length n. The security proof of building blocks is shown in Figure B.

Digital Decomposition. To avoid an oversized lookup table, a natural idea is to decompose an input x into several small digits $x^{(0)}, x^{(1)}, \ldots, x^{(k-1)}$, and each digit $x^{(i)}$ is with bit length d. The protocol for digital decomposition Π_{DigitDec} is shown in Figure 2. This algorithm takes a polynomial $A(\mathbf{X})$ that encodes a vector $\vec{a} = (x_0, x_1, \ldots, x_{n-1})$ with length n such that $A(\langle i \rangle) = x_i$ for $\langle i \rangle \in \{0,1\}^{\log n}$ as inputs, and outputs a polynomial $B(\mathbf{X}, \mathbf{Y})$ such that $B(\langle i \rangle, \langle j \rangle) = x_i^{(j)}$ for $\langle i \rangle \in \{0,1\}^{\log n}$ and $\langle j \rangle \in \{0,1\}^{\log k}$. We denote the algorithm as $(\mathsf{DigitDec}, [A]; A(\mathbf{X})) \to ([B]; B(\mathbf{X}, \mathbf{Y}))$.

First, we describe the method for handling positive numbers, where the range of positive values is $[0, \frac{p-1}{2}]$. Specifically, (1) check that each digit lies in a given range $[0, 2^d - 1]$. This part can be invoked with a lookup protocol on $t_i(\mathbf{Y}) = B(\langle i \rangle, \mathbf{Y})$ for each $i \in [0, n-1]$. (2) for each $\langle i \rangle \in \{0,1\}^{\log n}$, $B(\langle i \rangle, \mathbf{Y})$ is the correct digital decomposition of $A(\langle i \rangle)$. This part can be reduced to a matrix multiplication relation and can be proved with a sumcheck protocol such that $\sum_{\mathbf{y} \in \{0,1\}^{\log n}} B(\mathbf{r}, \mathbf{y}) \cdot \hat{e}(\mathbf{y}) = A(\mathbf{r})$, in which $\mathbf{r} \leftarrow \mathbb{F}^{\log n}$ is a random challenge, and $\hat{e}(\mathbf{Y})$ is a public polynomial such that $\hat{e}(\langle j \rangle) = 2^{j \cdot d}$ for $\langle j \rangle \in \{0,1\}^{\log k}$.

Protocol Π_{DigitDec}

Parameter. $x_i = x_i^{(k-1)} || x_i^{(k-2)} || \ldots || x_i^{(0)}$, each $x_i^{(j)}$ with bit length d.
Input. A polynomial $A(\mathbf{X})$ that encodes a vector $\vec{a} = (x_0, x_1, \ldots, x_{n-1})$ that $A(\langle i \rangle) = x_i$ for $\langle i \rangle \in \{0,1\}^{\log n}$.
Public Input. A lookup table $L_1 = \{0, 1, 2, \ldots, 2^d - 1\}$.
Output. A multivariate polynomial $B(\mathbf{X}, \mathbf{Y})$ for each $\langle i \rangle \in \{0,1\}^{\log \mathsf{n}}$, $\langle j \rangle \in \{0,1\}^{\log \mathsf{k}}$ such that $B(\langle i \rangle, \langle j \rangle) = x_i^{(j)}$.
Protocol Execution.

1) For each $i \in [n]$, parse $x_i = x_i^{(k-1)} || x_i^{(k-2)} || \ldots || x_i^{(0)}$, and define a multivariate polynomial $B(\mathbf{X}, \mathbf{Y})$ for each $\langle i \rangle \in \{0,1\}^{\log \mathsf{n}}, \langle j \rangle \in \{0,1\}^{\log \mathsf{k}}$ such that $B(\langle i \rangle, \langle j \rangle) = x_i^{(j)}$.
2) **Step 1. Check $x_i^{(k)}$ lies in $[0, 2^d - 1]$.**
   a) For each $\langle i \rangle \in \{0,1\}^{\log n}$, set $t_i(\mathbf{Y}) = B(\langle i \rangle, \mathbf{Y})$.
   b) Randomly choose $\mathbf{r}_1 \leftarrow \mathbb{F}^{\log k}$, compute $t_i(\mathbf{r}_1)$ and $B(\langle i \rangle, \mathbf{r}_1)$, and check $t_i(\mathbf{r}_1)$=$B(\langle i \rangle, \mathbf{r}_1)$.
   c) For each $\langle i \rangle \in \{0,1\}^{\log n}$, run $(\mathsf{Lookup}, L_1, [t_i]; t_i(\mathbf{Y}))$.
3) **Step 2. Check $B(\mathbf{X}, \mathbf{Y})$ is correct digital decomposition of $A(\mathbf{X})$.**
   a) Randomly choose $\mathbf{r}_2 \leftarrow \mathbb{F}^{\log n}$.
   b) Run Sumcheck to check that $\sum_{\mathbf{Y} \in \{0,1\}^{\log k}} B(\mathbf{r}_2, \mathbf{Y}) \cdot \hat{e}(\mathbf{Y}) = A(\mathbf{r}_2)$, in which $\hat{e}(\langle j \rangle) = 2^{j \cdot d}$ for $\langle j \rangle \in \{0,1\}^{\log k}$.
4) Return $([B]; B(\mathbf{X}, \mathbf{Y}))$.

Fig. 2: Protocol for Digital Decomposition Π_{DigitDec}

Comparison. We draw on the approach of comparison protocol in [3] to reduce the number and size of lookup tables. The protocol for comparison Π_{Cmp} is shown in Figure 3. If the prover shows a value x is positive, then it is identical to show that $x < \frac{p+1}{2}$, while $x < p$ constrains that x is bounded by a large prime p. Π_{Cmp} takes a polynomial $A(\mathbf{X})$ that encodes a vector $\vec{a} = (x_0, x_1, \ldots, x_{n-1})$ with length n such that $A(\langle i \rangle) = x_i$ for $\langle i \rangle \in \{0,1\}^{\log n}$ and a public value c as inputs, and outputs a polynomial $z_A(\mathbf{X})$ such that $z_A(\langle i \rangle) = 1$ if $x_i < c$, otherwise $z_A(\langle i \rangle) = 0$ for $\langle i \rangle \in \{0,1\}^{\log n}$.

Our method leverages the key observation shown in Hao et al. [3] , which expresses the less-than comparison between two multi-bit values as $1\{x < c\} := 1\{x_1 < c_1\} + 1\{x_1 = c_1\} \cdot 1\{x_0 < c_0\}$, where $x := x_1 \| x_0$ and $c := c_1 \| c_0$. Specifically, for each pair of corresponding bit chunks (x_i, c_i), we perform table lookups to obtain $z_i^{lt} = 1\{x_i < c_i\}$ and $z_i^{eq} = 1\{x_i = c_i\}$. These values are then recursively combined from the most significant chunk to the least significant chunk, according to

the above formula, to determine $1\{x < c\}$ for the entire value. Furthermore, Hao et al [3] also present an optimized version to reduce the number of lookup tables and the table size.

Therefore, during the preprocessing phase, both parties prepare two lookup tables: $L_1^{(j)}$, which is used to verify whether $x_i^{(j)} < c^{(j)}$ for each digit, and L_2, which is used to verify whether $x_i < c$ for the entire value. The prover then proceeds as follows: (1) performs digital decomposition of the vector $\vec{a} = (x_0, x_1, \ldots, x_{n-1})$; (2) for each digit, computes z_i^j to determine whether $x_i^j < c^{(j)}$, and invokes the lookup protocol on $L_1^{(j)}$ to show the correctness of z_i^j; and (3) aggregates the values z_i^j to compute z_i and outputs a vector $\vec{y}$ indicating whether $x_i < c$. The prover further invokes the lookup protocol on L_2 to show the correctness of z_i and the final output vector $\vec{y}$.

In addition, to address the frequent need to check whether all elements of a vector are smaller than c, we introduce the subprotocol Π_{CheckOne}, as shown in Figure 4.

Requantization The protocol Π_{Requan} for transforming a value q_1 under public scale $s_1 = \frac{a_1}{b_1}$ to a value q_2 under public scale $s_2 = \frac{a_2}{b_2}$ such that $s_1 \cdot q_1 \approx s_2 \cdot q_2$ is shown in Figure 5. This building block serves a fundamental role in performing addition and subtraction under different scales. We denote the algorithm as $(\mathsf{Requan}, a_1, b_1, a_2, b_2, [A]; A(\mathbf{X})) \to ([B]; B(\mathbf{X}))$. Specifically, this protocol consists of two steps: (1) define a new vector $\vec{b} = (y_0, y_1, \ldots, y_{n-1})$ along with a remainder $\vec{r} = (r_0, r_1, \ldots, r_{n-1})$, (2) constrain $r_i < b_1 \cdot a_2$ to ensure the correctness of $\vec{b}$.

V. Mathematical Functions

This section designs efficient ZK protocols for complex mathematical functions based on the above building blocks. We avoid using the iteration algorithm to improve the proof generation. More importantly, for mathematical functions, operations on the fractional scale and integer part are generally separated, but for certain functions such as division x/y, we adjust the scale to ensure the numerator x integer part q_x is greater than the denominator y integer part q_y to prevent the final result being 0. The security proof is in Appendix.C.

Square Root. Given a real number r_1 with a public scaling factor $s_1 = \frac{a_1}{b_1}$ and a private value q_1, such that $r_1 \approx s_1 \cdot q_1$. We compute the square root $r_2 \approx \sqrt{r_1} \approx s_2 \cdot q_2$ as follows, (1) for scale s_2, define a new scaling factor $s_2 = \frac{a_2}{b_2}$, where $a_2 = \lfloor\sqrt{a_1}\rceil, \quad b_2 = \lfloor\sqrt{b_1}\rceil$, and (2) for q_2, we invoke the protocol Π_{Sqrt} as shown in Figure 6 to prove the relation $q_2 = \sqrt{q_1}$. Hao et al. [3] used an iterative algorithm to compute the square root. However, this method incurs significant prover time. Meanwhile, Qu et al. [2] adopted a Lasso-based range proof to compute square roots, which significantly reduces the prover time compared with the iterative algorithm. Meanwhile, this method in ZKGPT [2] is not sound, since (1) each q_1 is required to be strictly positive (i.e., $q_1 \in [0, \frac{p-1}{2}]$), and (2) when computing the square of a in the range proof, it is necessary to ensure that the result a^2 does not exceed $\frac{p-1}{2}$. If these two relations are not constrained, then the soundness will be broken.

Protocol Π_{Cmp}

Parameter. A public value $c = c^{(k-1)}||\ldots||c^{(0)}$, each $c^{(i)}$ is with length d.

Input. A polynomial $A(\mathbf{X})$ that encodes a vector $\vec{a} = (x_0, x_1, \ldots, x_{n-1})$ with length n such that $A(\langle i\rangle) = x_i$ for $\langle i\rangle \in \{0,1\}^{\log n}$.

Public Input. For each $j \in [0, k-1]$. $L_1^{(j)} = \left\{\left(x^{(j)},\ z^{(j)} = 2^{k+j} \cdot \mathbf{1}\{x^{(j)} < c^{(j)}\} + 2^j \cdot \mathbf{1}\{x^{(j)} = c^{(j)}\}\right)\right\}_{x^{(j)} \in \{0,1\}^d}$
$L_2 = \left\{\left(z_{k-1} = \sum_{j=0}^{k-1}\left(2^{k+j} \cdot \mathbf{1}\{x^{(j)} < c^{(j)}\} + 2^j \cdot \mathbf{1}\{x^{(j)} = c^{(j)}\}\right),\ y_{k-1}\right)\right\}_{x^{(j)} \in \{0,1\}^d}$

Output. A polynomial $t_y(\mathbf{X})$ to the vector $\vec{y} = (y_0, y_1, \ldots, y_{n-1})$. if $x_i < c$, then $y_i = 1$, otherwise $y_i = 0$.

Protocol Execution.

1) **Step 1: Digital Decomposition of $A(\mathbf{X})$.**
   a) Run $(\mathsf{DigitDec}, [A]; A(\mathbf{X})) \to ([B]; B(\mathbf{X}, \mathbf{Y}))$.
2) **Step 2: Compute $z_i^{(j)}$ to decide if $x_i^{(j)} < c^{(j)}$ with $L_1^{(j)}$.**
   a) For $\langle i\rangle \in \{0,1\}^{\log n}$ and $\langle j\rangle \in \{0,1\}^{\log k}$, define $Z(\mathbf{X}, \mathbf{Y})$ such that $Z(\langle i\rangle, \langle j\rangle) = 2^{k+j} \cdot \mathbf{1}\{x^{(j)} < c^{(j)}\} + 2^j \cdot \mathbf{1}\{x^{(j)} = c^{(j)}\}$.
   b) For $\langle j\rangle \in \{0,1\}^{\log k}$, define $t_{B_j}(\mathbf{X}) = B(\mathbf{X}, \langle j\rangle)$ and $t_{Z_j}(\mathbf{X}) = Z(\mathbf{X}, \langle j\rangle)$.
   c) Choose a random challenge $\mathbf{r}_j \leftarrow \mathbb{F}^{\log n}$, and check that $t_{B_j}(\mathbf{r}_j) = B(\mathbf{r}_j, \langle j\rangle)$ and $t_{Z_j}(\mathbf{r}_j) = Z(\mathbf{r}_j, \langle j\rangle)$ for $\langle j\rangle \in \{0,1\}^{\log k}$.
   d) For $\langle j\rangle \in \{0,1\}^{\log k}$, run $(\mathsf{Lookup}, L_1^{(j)}, [t_{B_j}], [t_{Z_j}]; t_{B_j}(\mathbf{X}), t_{Z_j}(\mathbf{Y}))$.
3) **Step 3. $\sum_{\langle j\rangle \in \{0,1\}^{\log k}} z_i^{(j)} = z_i$ and decide if $x_i < c$ with L_2.**
   a) Define a polynomial $\hat{Z}(\mathbf{X})$ such that $\hat{Z}(\langle i\rangle) = \sum_{j=0}^{k-1} Z(\langle i\rangle, \langle j\rangle)$ for $\langle i\rangle \in \{0,1\}^{\log n}$.
   b) Randomly choose $\mathbf{r} \leftarrow \mathbb{F}^{\log n}$, and run sumcheck on $\sum_{\mathbf{Y} \in \{0,1\}^{\log n}} Z(\mathbf{r}, \mathbf{Y}) \cdot \mathbf{1}(\mathbf{Y}) = \hat{Z}(\mathbf{r})$.
   c) Define a polynomial $t_y(\mathbf{X})$ such that $t_y(\langle i\rangle) = 1$ if $x_i < c$, otherwise $t_y(\langle i\rangle) = 0$ for $\langle i\rangle \in \{0,1\}^{\log n}$.
   d) Run $(\mathsf{Lookup}, L_2, [\hat{Z}], [t_y]; \hat{Z}(\mathbf{X}), t_y(\mathbf{X}))$.
4) Return $([t_y]; t_y(\mathbf{X}))$.

Fig. 3: Protocol for Comparison Π_{Cmp}

Protocol Π_{CheckOne}

Input. A polynomial $y(\vec{X})$ for each $\langle i\rangle \in \{0,1\}^{\log \mathsf{n}}$.

Protocol Execution.

1) Randomly chosen a challenge α and define a public polynomial $\hat{\alpha}(\mathbf{X})$ that $\hat{\alpha}(\langle i\rangle) = \alpha^i$ for $\langle i\rangle \in \{0,1\}^{\log n}$.
2) Run sumcheck on $\sum_{\mathbf{X} \in \{0,1\}^{\log \mathsf{n}}} y(\vec{\mathbf{X}}) \cdot \hat{\alpha}(\vec{\mathbf{X}}) = 1 + \alpha + \alpha^2 + \cdots + \alpha^{n-1}$.
3) If the sumcheck passes, then output accept, otherwise output reject.

Fig. 4: Protocol for CheckOne Π_{CheckOne}

Followed by the work of ZKGPT, given $y = \lfloor\sqrt{x}\rceil$, it is equivalent to prove that $(2y-1)^2 \le 4x < (2y+1)^2$, and

Protocol Π_{Requan}

Inputs. A polynomial $A(\mathbf{X})$ that encodes vector $\vec{a} = (x_0, x_1, \ldots, x_{n-1})$ under a source scale $s_1 = \frac{a_1}{b_1}$, such that $A(\langle i \rangle) = x_i$ for all $\langle i \rangle \in \{0,1\}^{\log n}$.
Public Inputs. A source scale $s_1 = \frac{a_1}{b_1}$ and a destination scale $s_2 = \frac{a_2}{b_2}$.
Outputs. A polynomial $B(\vec{X})$ to a vector $\vec{b} = (y_0, y_1, \ldots, y_{n-1})$ under a destination scale $s_2 = \frac{a_2}{b_2}$, such that $B(\langle i \rangle) = y_i$ for all $\langle i \rangle \in \{0,1\}^{\log n}$.
Protocol Execution. This protocol transforms a vector from a source scale to a destination scale.
1) If $a_1 = a_2$ and $b_1 = b_2$, return $([A]; A(\mathbf{X}))$.
2) **Step 1. Output $B(\mathbf{X})$ along with a remainder $r(\mathbf{X})$.**
   a) Define $B(\mathbf{X})$ and $r(\mathbf{X})$ such that $a_2 \cdot b_1 \cdot B(\langle i \rangle) + r(\langle i \rangle) = b_2 \cdot a_1 \cdot A(\langle i \rangle)$ for $\langle i \rangle \in \{0,1\}^{\log n}$.
3) **Step 2. Verify that the remainder $r(\mathbf{X})$ is smaller than the divisor $b_1 \cdot a_2$**
   a) Run $(\mathsf{Cmp}, b_1 \cdot a_2, [r]; r(\mathbf{X})) \to ([z_r]; z_r(\mathbf{X}))$.
   b) Run $(\mathsf{CheckOne}, [z_r], z_r(\mathbf{X}))$.
4) Return $([B]; B(\mathbf{X}))$.

Fig. 5: Protocol for Requantization Π_{Requan}

this condition can be reduced to two separate range proofs: 1) $(2 \cdot y + 1)^2 - 4 \cdot x > 0$ and (2) $4 \cdot x - [(2 \cdot y - 1)^2 - 1] > 0$. In addition, to ensure soundness, we need to ensure that $(2 \cdot y - 1)^2$ is bounded (i.e., $(2 \cdot y - 1)^2 \leq \frac{p-1}{2}$). Specifically, we decompose the protocol Π_{Sqrt} into the following steps: (1) verify that each $x_i > 0$, (2) define $(2 \cdot x + 1)^2$ and $(2 \cdot x - 1)^2$, and run Π_{Cmp} to ensure that $(2 \cdot x + 1)^2$ and $(2 \cdot x - 1)^2$ are less than $\frac{p+1}{2}$, (3) invoke the range check on $y_i = \sqrt{x_i}$, and (4) update the new scale s_2.
Exponentiation. The exponentiation protocol Π_{Exp} is illustrated in Figure 7. Given a value x, we first decompose it into chunks as $x = x^{(k-1)} \parallel x^{(k-2)} \parallel \cdots \parallel x^{(0)}$. We then apply exponentiation to each chunk $x^{(j)}$ individually and aggregate the results to obtain the final output. In addition, since x may be negative, we first invoke the comparison protocol Π_{Cmp} to determine the sign of x. If $x < 0$, we convert it to its absolute value before performing exponentiation. Finally, if x was negative, we take the reciprocal of the result.

Specifically, the protocol consists of five steps: (1) decide the sign s_i of each x_i and get $|x_i|$, (2) decompose each x_i into k digits $x_i^{(k-1)}, x_i^{(k-2)}, \ldots, x_i^{(0)}$, perform exponentiation on each digit $x_i^{(j)}$, and use lookup table to check the correctness of computation, (3) compute the multiplication of each $e^{x_i^{(j)}}$ to e^{x_i}, (4) compute $\frac{1}{e^{x_i}}$ for all negative x_i and (5) compute the result y_i based on the sign s_i of each x_i.
Division. At a high level, to prove $z = \frac{x}{y}$, it suffices to show that $z \cdot y = x + r$ with remainder $r < y$. However, several issues must be addressed: (1) if either x or y is negative, both should first be converted to their absolute values, and the sign of z should be determined based on the original signs of x and y and (2) $|x|$ should be no less than $|y|$, otherwise, the result $|z|$ will always be zero. The protocol for the division protocol Π_{Div} is shown in Figure 8.

Protocol Π_{Sqrt}

Input. A polynomial $q_1(\mathbf{X})$ to a vector $\vec{a} = (x_0, x_1, \ldots, x_{n-1})$ such that $q_1(\langle i \rangle) = a_i$ for $\langle i \rangle \in \{0,1\}^{\log n}$. with scale $s_1 = \frac{a_1}{b_1}$.
Output. A polynomial $q_2(\mathbf{X})$ to a vector $\vec{b} = (y_0, y_1, \ldots, y_{n-1})$ such that $q_2(\langle i \rangle) = y_i$ for $\langle i \rangle \in \{0,1\}^{\log n}$ with scale $s_2 = \frac{a_2}{b_2}$.
Protocol Execution. This protocol proves that $q_2 \approx \sqrt{q_1}$.
1) **Step 1. Verify that each $q_1(\langle i \rangle) = x_i \geq 0$.**
   a) Run $(\mathsf{Cmp}, \frac{p+1}{2}, [q_1]; q_1(\mathbf{X})) \to ([Z_{q_1}]; Z_{q_1}(\mathbf{X}))$.
   b) Run $(\mathsf{CheckOne}, [Z_{q_1}]; Z_{q_1}(\mathbf{X}))$.
2) **Step 2. Define $q_2(\mathbf{X})$, and $(2 \cdot q_2(\mathbf{X}) + 1)^2$ and $(2 \cdot q_2(\mathbf{X}) - 1)^2$ for range proof, and check $(2 \cdot q_2(\mathbf{X}) + 1)^2 < \frac{p+1}{2}$.**
   a) Define $t_1(\mathbf{X}) = (2 \cdot q_2(\mathbf{X}) + 1)^2$ and $t_2(\mathbf{X}) = (2 \cdot q_2(\mathbf{X}) - 1)^2$.
   b) Run $(\mathsf{Cmp}, \frac{p+1}{2}, [t_1]; t_1(\mathbf{X})) \to ([Z_{t_1}]; Z_{t_1}(\mathbf{X}))$ and $(\mathsf{Cmp}, \frac{p+1}{2}, [t_2]; t_2(\mathbf{X})) \to ([Z_{t_2}]; Z_{t_2}(\mathbf{X}))$.
   c) Run $(\mathsf{CheckOne}, [Z_{t_1}]; Z_{t_1}(\mathbf{X}))$ and $(\mathsf{CheckOne}, [Z_{t_2}]; Z_{t_2}(\mathbf{X}))$.
3) **Step 3. Range check for $q_2(\mathbf{X})$.**
   a) Define $t_3(\mathbf{X}) = 4 \cdot \mathbf{X} - t_2(\mathbf{X}) + 1$ and $t_4(\mathbf{X}) = t_1(\mathbf{X}) - 4 \cdot \mathbf{X}$.
   b) Run $(\mathsf{Cmp}, \frac{p+1}{2}, [t_3]; t_3(\mathbf{X})) \to ([Z_{t_3}]; Z_{t_3}(\mathbf{X}))$ and $(\mathsf{Cmp}, \frac{p+1}{2}, [t_4]; t_4(\mathbf{X})) \to ([Z_{t_4}]; Z_{t_4}(\mathbf{X}))$.
   c) Run $(\mathsf{CheckOne}, [Z_{t_3}]; Z_{t_3}(\mathbf{X}))$ and $(\mathsf{CheckOne}, [Z_{t_4}]; Z_{t_4}(\mathbf{X}))$.
4) **Step 4. Define scale $s_2 = \frac{a_2}{b_2}$.**
   a) Compute $a_2 = \lfloor \sqrt{a_1} \rceil$ and $b_2 = \lfloor \sqrt{b_1} \rceil$
5) Return $(a_2, b_2, [q_2]; q_2(\mathbf{X}))$.

Fig. 6: Protocol for Square Root Π_{Sqrt}

Specifically, the division protocol Π_{Div} consists of four steps: (1) decide the sign of x_i and y_i, and get the absolute value $|x_i|$ and $|y_i|$, (2) choose another scale s_2, adjust $|x_i|$ and $|y_i|$ under the scale s_1 to $|x_i'|$ and $|y_i'|$ under s_2 to let $|x_i'| > |y_i'|$, (3) compute $|z_i| = \frac{|x_i'|}{|y_i'|}$ and do range check to ensure the remainder r_i is less than the divisor $|y_i'|$ (i.e., $r_i \leq |y_i'|$, and (4) compute the scale s_3 and decide the final sign of z_i based on the sign of x_i and y_i.
Addition. This protocol is designed to add two values represented under different scaling factors, by computing $s_3 \cdot q_3 = s_1 \cdot q_1 + s_2 \cdot q_2$. The protocol Π_{Add} is shown in Figure 9. Specifically, this protocol consists of two parts: (1) convert each x_i under scale s_1 and y_i under scale s_2 to (x_i', y_i') under scale s_3, and (2) perform addition $z_i = x_i' + y_i'$ under the same scale s_3.

VI. Functions

In this section, we apply the above-mentioned building blocks to mainstream non-linear functions of the LLM inference layer, including GELU, Softmax function for multi-head attention, and normalization. In the design of functions, we adjust the scale for each non-linear function. This maintains

Protocol Π_{Exp}

Parameter. Public scale $s_1 = \frac{a_1}{b_1}$ and $s_2 = \frac{a_2}{b_2}$.
Input. A polynomial $A(\mathbf{X})$ to a vector $\vec{a} = (x_0, x_1, \ldots, x_{n-1})$ such that $A(\langle i \rangle) = x_i$ for $\langle i \rangle \in \{0,1\}^{\log n}$ under scale $s_1 = \frac{a_1}{b_1}$.
Public Input. $L_1^{(j)} = \{x^{(j)}, y^{(j)} = \lfloor e^{x^{(j)} \cdot 2^j \cdot d \cdot s_1} / s_2 \rceil\}_{x^{(j)} \in \{0,1\}^d}$.
Output. A polynomial $B(\mathbf{X})$ to a vector $\vec{b} = (y_0, y_1, \ldots, y_{n-1})$ such that $B(\langle i \rangle) = y_i$ for $\langle i \rangle \in \{0,1\}^{\log n}$ with scale $s_2 = \frac{a_2}{b_2}$
Protocol Execution

1) **Step 1. Decide the sign of x_i and compute $|x_i|$.**
   a) Run $(\mathsf{Cmp}, \frac{p+1}{2}, [A]; A(\mathbf{X})) \to ([Z_A]; Z_A(\mathbf{X}))$.
   b) Define a polynomial $\hat{A}(\mathbf{X}) = (2 \cdot Z_A(\mathbf{X}) - 1) \cdot A(\mathbf{X})$ for $\mathbf{X} \in \{0,1\}^{\log n}$.
2) **Step 2. Decompose $|x_i|$ into k digits and perform over each digit.**
   a) Run $(\mathsf{DigitDec}, [\hat{\mathsf{A}}]; \hat{\mathsf{A}}(\mathbf{X})) \to ([\hat{B}]; \hat{B}(\mathbf{X}, \mathbf{Y}))$.
   b) Define $\hat{C}(\mathbf{X}, \mathbf{Y})$ such that $\hat{C}(\langle i \rangle, \langle j \rangle) = \lfloor e^{\hat{B}(\langle i \rangle, \langle j \rangle) \cdot 2^{j \cdot d} \cdot \frac{a_1}{b_1} / \frac{a_2}{b_2}} \rceil$ for $\langle i \rangle \in \{0,1\}^{\log n}$ and $\langle j \rangle \in \{0,1\}^{\log k}$.
   c) For $\langle i \rangle \in \{0,1\}^{\log n}$, define $t_{C_i}(\mathbf{Y}) = \hat{C}(\langle i \rangle, \mathbf{Y}))$, and randomly choose $\mathbf{r}_1 \leftarrow \mathbb{F}^{\log n}$, compute $t_{C_i}(\mathbf{r}_1)$ and $\hat{C}(\langle i \rangle, \mathbf{r}_1)$, and check $t_{C_i}(\mathbf{r}_1) = \hat{C}(\langle i \rangle, \mathbf{r}_1)$.
   d) For $\langle i \rangle \in \{0,1\}^{\log n}$, define $t_{B_i}(\mathbf{Y}) = \hat{B}(\langle i \rangle, \mathbf{Y}))$, and randomly choose $\mathbf{r}_2 \leftarrow \mathbb{F}^{\log n}$, compute $t_{B_i}(\mathbf{r}_2)$ and $\hat{B}(\langle i \rangle, \mathbf{r}_2)$, and check $t_{B_i}(\mathbf{r}_2) = \hat{B}(\langle i \rangle, \mathbf{r}_2)$.
   e) For $\langle i \rangle \in \{0,1\}^{\log n}$, run $(\mathsf{Lookup}, L_1^{(i)}, [t_{B_i}], [t_{C_i}]; t_{B_i}(\mathbf{Y}), t_{C_i}(\mathbf{Y}))$
3) **Step 3. Compute the multiplication of $\hat{B}(\mathbf{X}, \mathbf{Y})$ for each row (aggregate all results of all digits).**
   a) Define $Z_B^{(1)}(\mathbf{X}) = \prod_{j \in \{0,1\}^{\log k}} \hat{B}(\mathbf{X}, \langle j \rangle)$, sample $\mathbf{r}_3 \leftarrow \mathbb{F}^{\log n}$, and run ProdCheck on $Z_B^{(1)}(\mathbf{r}_3) = \prod_{j \in \{0,1\}^{\log k}} \hat{B}(\mathbf{r}_3, \langle j \rangle)$.
4) **Step 4. Compute $\frac{1}{e^{x_i}}$ for negative x_i.**
   a) Run $(\mathsf{Div}, 1, 1, a_2, b_2, [\mathbf{1}], [Z_B^{(1)}]; \mathbf{1}(\mathbf{X}), Z_B^{(1)}(\mathbf{X})) \to (a_3, b_3, [Z_B^{(2)}]; Z_B^{(2)}(\mathbf{X}))$.
5) **Step 5. Compute the final result of e^{x_i} based on the sign of x_i.**
   a) Run $(\mathsf{Requan}, a_3, b_3, a_2, b_2, [Z_B^{(2)}]; Z_B^{(2)}(\mathbf{X})) \to ([Z_B^{(3)}]; Z_B^{(3)}(\mathbf{X}))$.
   b) $\tilde{B}(\mathbf{X}) = Z_A(\mathbf{X}) \cdot Z_B^{(3)}(\mathbf{X}) + (1 - Z_A(\mathbf{X})) \cdot Z_B(\mathbf{X})$
6) Return $([\hat{B}]; \hat{B}(\mathbf{X}))$.

Fig. 7: Protocol for Exponentiation Π_{Exp}

Protocol Π_{Div}

Parameter. Public scale $s_1 = \frac{a_1}{b_1}$ and $s_2 = \frac{a_2}{b_2}$.
Input. Two polynomials $A(\mathbf{X})$ and $B(\mathbf{X})$ that respectively encodes $\vec{a} = (x_0, x_1, \ldots, x_{n-1})$ and $\vec{b} = (y_0, y_1, \ldots, y_{n-1})$ such that $A(\langle i \rangle) = x_i$ and $B(\langle i \rangle) = y_i$ for $\langle i \rangle \in \{0,1\}^{\log n}$ under scale $s_1 = \frac{a_1}{b_1}$ and $s_2 = \frac{a_2}{b_2}$.
Output. A polynomial $\hat{C}(\vec{X})$ to a vector $\vec{c} = (z_0, z_1, \ldots, z_{n-1})$ such that $C(\langle i \rangle) = z_i$ for $\langle i \rangle \in \{0,1\}^{\log n}$ under scale $s_3 = \frac{a_3}{b_3}$.
Protocol Execution.

1) **Step 1. Decide the sign of each x_i and y_i and get $|x_i|$ and $|y_i|$.**
   a) Run $(\mathsf{Cmp}, \frac{p+1}{2}, [A]; A(\mathbf{X})) \to ([Z_A]; Z_A(\mathbf{X}))$, and $(\mathsf{Cmp}, \frac{p+1}{2}, [B]; B(\mathbf{X})) \to ([Z_B]; Z_B(\mathbf{X}))$.
   b) Define $\hat{A}(\mathbf{X}) = (2 \cdot Z_A(\mathbf{X}) - 1) \cdot A(\mathbf{X})$ and $\hat{B}(\mathbf{X}) = (2 \cdot Z_B(\mathbf{X}) - 1) \cdot B(\mathbf{X})$ for $\mathbf{X} \in \{0,1\}^{\log n}$.
2) **Step 2. Adjust scale s_1 and s_2 to let each $|x_i| > |y_i|$ to prevent the result z_i is 0.**
   a) Run $(\mathsf{Requan}, a_1, b_1, a_1', b_1', [\hat{A}]; \hat{A}(\mathbf{X}) \to ([\tilde{A}]; \tilde{A}(\mathbf{X}))$, and $(\mathsf{Requan}, a_2, b_2, a_2', b_2', [\hat{B}]; \hat{B}(\mathbf{X}) \to ([\tilde{B}]; \tilde{B}(\mathbf{X}))$.
   b) Define $t_1(\mathbf{X}) = \tilde{A}(\mathbf{X}) - \tilde{B}(\mathbf{X})$, and run $(\mathsf{Cmp}, \frac{p+1}{2}, [t_1]; t_1(\mathbf{X})) \to ([Z_1]; Z_1(\mathbf{X}))$.
   c) Run $(\mathsf{CheckOne}, [Z_1]; Z_1(\mathbf{X}))$.
3) **Step 3. Compute $z_i = \frac{x_i}{y_i}$.**
   a) Compute $C(\mathbf{X})$ and $r(\mathbf{X})$ such that $C(\mathbf{X}) \cdot \tilde{B}(\mathbf{X}) + r(\mathbf{X}) = \tilde{A}(\mathbf{X})$ for $X \in \{0,1\}^{\log n}$.
   b) Define $t_2(\mathbf{X}) = \tilde{B}(\mathbf{X}) - r(\mathbf{X})$, run $(\mathsf{Cmp}, \frac{p+1}{2}, [t_2]; t_2(\mathbf{X})) \to ([Z_2]; Z_2(\mathbf{X}))$.
   c) Run $(\mathsf{Checkone}, [Z_2]; Z_2(\mathbf{X}))$.
4) **Step 4. Decide the sign of each z_i based on the sign of x_i and y_i and output the scale $s_3 = \frac{a_3}{b_3}$.**
   a) Compute $a_3 = a_1' \cdot b_2'$ and $b_3 = a_2' \cdot b_1'$.
   b) Compute $\hat{C}(\mathbf{X}) = (2 \cdot Z_A(\mathbf{X}) - 1) \cdot (2 \cdot Z_B(\mathbf{X}) - 1) \cdot C(\mathbf{X})$ for $X \in \{0,1\}^{\log n}$.
5) Return $(a_3, b_3, [\hat{C}]; \hat{C}(\mathbf{X}))$.

Fig. 8: Protocol for Division Π_{Div}

accuracy and consistency across all operations. We prove the security of these components in Appendix.D.

GELU. Given an input x, GELU computes the equation: $y = 0.5 \cdot x \cdot (1 + \mathsf{Tanh}(\sqrt{2/\pi} \cdot (x + 0.044715 \cdot x^3))$, Then, we expand $\mathsf{Tanh}(x) = 2 \cdot \mathsf{sigmoid}(2x) - 1 = \frac{e^{2x}-1}{e^{2x}+1}$. The protocol Π_{GELU} for the GELU layer is shown in Figure 10. It can be decomposed into the following steps.

1) Compute $t_i = \sqrt{2/\pi} \cdot (x + 0.044715 \cdot x^3)$. It can be decomposed into four steps: (1) compute the cubic term x^3 and update the scale, (2) Re-scale x^3 to the same working scale as x, (3) compute two constants encoding $\sqrt{2/\pi}$ and $\sqrt{2/\pi} \cdot 0.044715$ under the current scale, and (4) Form the integer part of t by a linear combination $t = \alpha \cdot x + \beta \cdot x^3$ and update the scale.
2) Compute $\mathsf{Tanh}(t_i) = \frac{e^{2 \cdot t_i} - 1}{e^{2 \cdot t_i} + 1}$. It can be decomposed into four steps: (1) run the exponentiation protocol to evaluate e^{2t}, (2) pick intermediate scales for the numerator $e^{2t} - 1$ and the denominator $e^{2t} + 1$, (3) requantize numerator and denominator to let the numerator larger than the denominator, and (4) run division protocol to get $\mathsf{Tan}(t_i)$.
3) Compute $y_i = 0.5 \cdot x_i + 0.5 \cdot x_i \cdot \mathsf{Tanh}(t_i)$. The method proceeds in two steps: (1) requantize x to the output scale and multiply by the constant $1/2$, and (2) compute the final output by adding the two terms $0.5\,x$ and $0.5\,x\,\tanh(t)$.

Softmax Layer Softmax plays a fundamental role in LLM.

Protocol Π_{Add}

Parameter. Public scale $s_1 = \frac{a_1}{b_1}$ and $s_2 = \frac{a_2}{b_2}$.
Input. Two polynomials $A(\mathbf{X})$ and $B(\mathbf{X})$ that respectively encodes $\vec{a} = (x_0, x_1, \ldots, x_{n-1})$ and $\vec{b} = (y_0, y_1, \ldots, y_{n-1})$ such that $A(\langle i \rangle) = x_i$ and $B(\langle i \rangle) = y_i$ for $\langle i \rangle \in \{0,1\}^{\log n}$ under scale $s_1 = \frac{a_1}{b_1}$ and $s_2 = \frac{a_2}{b_2}$, a scale $s_3 = \frac{a_3}{b_3}$.
Output. A polynomial $C(\vec{X})$ to a vector $\vec{c} = (z_0, z_1, \ldots, z_{n-1})$ under scale $s_3 = \frac{a_3}{b_3}$.
Protocol Execution.
1) **Step 1. Transform x_i under scale s_1 and y_i under s_2 to values under scale s_3.**
   a) $(\mathsf{Requan}, a_1, b_1, a_3, b_3, [A]; A(\mathbf{X})) \to ([\hat{A}]; \hat{A}(\mathbf{X}))$.
   b) $(\mathsf{Requan}, a_2, b_2, a_3, b_3, [B]; B(\mathbf{X})) \to ([\hat{B}]; \hat{B}(\mathbf{X}))$.
2) **Step 2. Perform addition under scale s_3.**
   a) Compute $C(\mathbf{X}) = \hat{A}(\mathbf{X}) + \hat{B}(\mathbf{X})$.
3) Return $([C]; C(\mathbf{X}))$.

Fig. 9: Protocol for Addition Π_{Add}

Given a vector $\vec{a} = (x_0, x_1, \ldots, x_{n-1})$ with size n, then the equation holds: $y_i = \frac{e^{x_i - x_{max}}}{\sum_{j \in [0,n-1]} e^{x_j - x_{max}}}$. The protocol Π_{Softmax} for the Softmax layer is shown in Figure 11. It can be decomposed into the following steps.

1) $\mathsf{max}(x_0, x_1, \ldots, x_n - 1) = x_{max}$. It can be decomposed into two constraints, (1) $t_i = x_{max} - x_i \geq 0$, we invoke comparison protocol on $t_i \leq \frac{p+1}{2}$, and (2) $\prod_{i=0}^{n-1} t_i = 0$, we invoke the product check on the polynomial encoding of t_i.
2) Compute e^{x_i} and e^{t_i}, we invoke the exponentiation protocol on the polynomial encoding of x_i and t_i.
3) Compute $\vec{b} = (y_0, y_1, \ldots, y_{n-1})$ such that $y_i = \frac{e^{x_i}}{\sum_{i=0}^{n-1} e^{t_i}}$. It can be decomposed into three steps: (1) compute the sum of the denominator with the sumcheck protocol, (2) requantize the integer part of e^{x_i} to ensure that the rescaled integer part of e^{x_i} is larger than the denominator, and (3) run division protocol to get the final results.

Normalize. Given an input vector $\vec{a} = (x_0, x_1, \ldots, x_{n-1})$ of size n, it outputs a vector $\vec{b} = (y_0, y_1, \ldots, y_{n-1})$ such that: $y_i := \gamma_i \cdot \frac{x_i - \mu}{\sqrt{\sigma}} + \beta_i$, where (γ_i, β_i) are trained affine transform parameters, $\mu := \frac{\sum_{i \in [0,n-1]} x_i}{n}$, and $\sigma := \frac{\sum_{i \in [0,n-1]} (x_i - \mu)^2}{n}$.

The protocol Π_{Norm} for the normalize layer is shown in Figure 12. It can be decomposed into the following steps.

1) $\mu := \frac{\sum_{i \in [0,n-1]} x_i}{n}$. The method proceeds in two steps: (1) compute the sum of the numerator x_i with the sumcheck protocol, and (2) compute μ.
2) $\sigma := \frac{\sum_{i \in [0,n-1]} (x_i - \mu)^2}{n}$. The method proceeds in two steps: (1) compute the sum of the numerator $x_i - \mu$ with the sumcheck protocol, and (2) compute σ.
3) $y_i := \gamma_i \cdot \frac{x_i - \mu}{\sqrt{\sigma}} + \beta_i$. The method proceeds in two steps: (1) requantize the integer part of $x_i - \mu$ to ensure that it is larger than the integer part of σ, (2) compute the $\frac{x_i - \mu}{\sigma}$ along with a remainder r_i, and check if $r_i < \sigma$, (3) respectively multiply the scale and integer part of γ and $\frac{x_i - \mu}{\sigma}$, and (4) run Π_{Add}w to compute the final result.

Protocol Π_{GELU}

Parameter. Public scale $s_1 = \frac{a_1}{b_1}$ and $s_2 = \frac{a_2}{b_2}$.
Input. A polynomial $A(\mathbf{X})$ to a vector $\vec{a} = (x_0, x_1, \ldots, x_{n-1})$ with size n under scale $s_1 = \frac{a_1}{b_1}$ such that $A(\langle i \rangle) = x_i$ for $\langle i \rangle \in \{0,1\}^{\log n}$.
Output. A polynomial $B(\mathbf{X})$ to a vector $\vec{b} = (y_0, y_1, \ldots, y_{n-1})$ with size n under scale $s_2 = \frac{a_2}{b_2}$ such that $B(\langle i \rangle) = y_i$ for $\langle i \rangle \in \{0,1\}^{\log n}$.
Protocol Execution.
1) **Step 1. Compute** $t = \sqrt{2/\pi} \cdot (x + 0.044715 \cdot x^3)$.
   a) Compute $t_A(\mathbf{X}) = A^3(\mathbf{X})$ for $\mathbf{X} \in \{0,1\}^{\log n}$ with scale $s_{t_A} = \frac{a_{t_A}}{b_{t_A}} = \frac{a_1^3}{b_1^3}$, to compute x^3.
   b) Run $(\mathsf{Requan}, a_{t_A}, b_{t_A}, a_1, b_1, [t_A]; t_A(\mathbf{X})) \to ([t'_A]; t'_A(\mathbf{X}))$.
   c) Compute $q_1 = \lfloor \sqrt{2/\pi} / \frac{a_1}{b_1} \rceil$ and $q_2 = \lfloor \sqrt{2/\pi} \cdot 0.044715 / \frac{a_1}{b_1} \rceil$, to compute $\sqrt{2/\pi} \cdot (x + 0.044715 \cdot x^3)$.
   d) Define $s_t = \frac{a_t}{b_t} = \frac{a_1^2}{b_1^2}$ and $t(\mathbf{X}) = q_1 \cdot A(\mathbf{X}) + q_2 \cdot t'_A(\mathbf{X})$.
2) **Step 2. Compute** $\mathsf{Tanh}(t) = \frac{e^{2t}-1}{e^{2t}+1}$ **($t(\mathbf{X})$ is under scale s_t).**
   a) Choose a'_t and b'_t, and compute $(\mathsf{Exp}, a_t, b_t, a'_t, b'_t, [t]; t(\mathbf{X})) \to ([t_e]; t_e(\mathbf{X}))$.
   b) Choose two intermediate scale $s_{t_1} = \frac{a_{t_1}}{b_{t_1}}$ and $s_{t_2} = \frac{a_{t_2}}{b_{t_2}}$ such that $a_{t_1} < b_{t_1}$ and $a_{t_2} < b_{t_2}$.
   c) Run $(\mathsf{Requan}, a_{t_1}, b_{t_1}, a'_t, b'_t, [t_e]; t_e(\mathbf{X})) \to ([t_1]; t_1(\mathbf{X}))$, define $t'_1(\mathbf{X}) = t_1(\mathbf{X}) - \lfloor \frac{b_{t_1}}{a_{t_1}} \rceil \ (e^{2t} - 1)$.
   d) Run $(\mathsf{Requan}, a_{t_2}, b_{t_2}, a'_t, b'_t, [t_e]; t_e(\mathbf{X})) \to ([t_2]; t_2(\mathbf{X}))$, and define $t'_2(\mathbf{X}) = t_2(\mathbf{X}) + \lfloor \frac{b_{t_2}}{a_{t_2}} \rceil \ (e^{2t} + 1)$.
   e) Define $s_{t_3} = \frac{a_{t_3}}{b_{t_3}} = \frac{a_{t_1} \cdot b_{t_2}}{b_{t_1} \cdot a_{t_2}}$ and $(\mathsf{Div}, a_{t_1}, b_{t_1}, a_{t_2}, b_{t_2}, [t'_1], [t'_2]; t'_1(\mathbf{X}), t'_2(\mathbf{X})) \to (a_{t_3}, b_{t_3}, [t_3]; t_3(\mathbf{X}))$.
3) **Step 3. Compute** $y = 0.5 \cdot x + 0.5 \cdot x \cdot \mathsf{Tanh}(t)$**.**
   a) Define $\frac{a_2}{b_2} = \frac{a_1 \cdot a_{t_3}}{2 \cdot b_1 \cdot b_{t_3}}$ and run $(\mathsf{Requan}, a_1, b_1, a_2, b_2, [A]; A(\mathbf{X})) \to ([\hat{A}]; \hat{A}(\mathbf{X}))$.
   b) Define $B(\mathbf{X}) = \hat{A}(\mathbf{X}) + A(\mathbf{X}) + t_3(\mathbf{X})$.
4) Return $(a_2, b_2, [B]; B(\mathbf{X}))$.

Fig. 10: Protocol for GELU activation function Π_{GELU}

VII. Implementation and Evaluation

A. Benchmark and Environment

Implementation Details. Our implementation is built in C++. Experiments run on two Hygon C86 7390 instances (32-core vCPUs, 512GB memory each) using 16 threads. For the layer-wise scaling experiment (Table IV), we report single-thread results. We instantiate KZG and IPA over the BN254 scalar field, and FRI over the M61 field. Lookup tables are constructed by decomposing inputs into 12-bit digits. We evaluate prover time, verification time, and proof size across multiple dimensions, including: (i) basic operators (Appendix E), (ii) core LLM functions (Table I), (iii) end-to-end GPT-2 inference (Table II), (iv) layer-wise scaling (Table IV), (v) larger LLMs (Table III), and (vi) approximation accuracy (Table V).

Function	Scheme	256 × 256			512 × 512			1024 × 1024		
		Prover (s)	Verify (ms)	Proof	Prover (s)	Verify (ms)	Proof	Prover (s)	Verify (ms)	Proof
GELU	Ours (KZG-BN254)	4.431	4.157	4.99 KB	18.960	8.841	4.99 KB	57.496	22.501	4.99 KB
	Ours (IPA-BN254)	18.584	792.786	7.03 KB	87.698	3062.492	8.66 KB	315.272	12776.097	9.28 KB
	Ours (FRI-M61-2^{64})	1.894	0.716	62.75 KB	3.205	1.153	84.84 KB	10.045	4.170	130.40 KB
	Hao et al. [3] ($p \approx$M61-2^{64})	6.387	6672.400	125.53 MB	15.885	17122.768	534.81 MB	91.827	88083.215	1664.20 MB
Softmax	Ours (KZG-BN254)	5.224	6.982	5.09 KB	19.071	8.675	5.09 KB	60.012	26.702	5.09 KB
	Ours (IPA-BN254)	21.369	970.261	8.03 KB	90.685	3565.400	8.78 KB	334.540	13948.009	9.53 KB
	Ours (FRI-M61-2^{64})	2.656	0.778	93.00 KB	3.557	1.909	107.12 KB	11.556	5.112	135.20 KB
	Hao et al. [3] ($p \approx$M61-2^{64})	18.070	19593.132	154.88 MB	74.751	81354.879	818.02 MB	134.470	119832.940	2209.86 MB
Norm	Ours (KZG-BN254)	3.894	3.095	3.72 KB	12.536	4.752	3.72 KB	38.609	17.359	3.72 KB
	Ours (IPA-BN254)	14.988	709.042	5.28 KB	68.119	2540.154	6.74 KB	253.881	10776.555	8.12 KB
	Ours (FRI-M61-2^{64})	0.767	0.413	61.12 KB	2.811	1.137	76.84 KB	9.306	4.080	102.66 KB
	Hao et al. [3] ($p \approx$M61-2^{64})	12.017	13029.417	86.58 MB	49.709	54099.066	457.27 MB	89.423	79688.915	1235.41 MB

TABLE I: Comparison of LLM functions in terms of prover time, verification time, and proof size with different input sizes.

Scheme	KZG + BN254			IPA + BN254			FRI + M61		
	Prov. (s)	Ver. (ms)	Proof (MB)	Prov. (s)	Ver. (ms)	Proof (MB)	Prov. (s)	Ver. (ms)	Proof (MB)
zkGPT (GPT-2-Small)	N/A	N/A	N/A	88.515	4809.032	0.978	N/A	N/A	N/A
OpenLLM (GPT-2-Small)	29.700	253.590	0.327	63.670	4421.229	0.334	13.280	16.685	2.125
OpenLLM (GPT-2-Medium)	61.651	510.456	0.679	136.465	9854.928	0.848	27.508	34.172	4.353
OpenLLM (GPT-2-Large)	162.653	1487.193	1.178	347.189	22876.186	2.459	59.114	87.558	10.632

TABLE II: End-to-end performance comparison on GPT-2 models of increasing sizes (Small, Medium, and Large), with 117M, 345M, and 774M parameters, respectively. "N/A" indicates that the corresponding configuration is not supported or not implemented in prior work.

Model	Setting	KZG + BN254			IPA + BN254			FRI + M61		
		Prover (s)	Verify (ms)	Proof (MB)	Prover (s)	Verify (ms)	Proof (MB)	Prover (s)	Verify (ms)	Proof (MB)
OpenLLM (SmolLM-2-1.7B)	End-to-End	147.885	735.845	1.925	407.498	29442.080	3.110	87.962	151.864	16.226
	Single Layer	6.202	38.786	0.037	17.799	1001.786	0.116	4.768	3.672	0.595
OpenLLM (Qwen-3-8B)	End-to-End	809.159	2884.302	2.838	1430.496	65035.521	7.941	258.640	329.140	55.652
	Single Layer	23.613	95.047	0.086	49.796	2210.150	0.299	9.012	9.755	1.360

TABLE III: End-to-end and single-layer inference performance on larger LLMs. "Single Layer" reports the cost of proving one Transformer layer, while "End-to-End" corresponds to full-model inference, including all layers and system overhead.

Model	PCS	1	6	12
GPT-2 Small	KZG + BN254	11.764	70.721	140.882
	IPA + BN254	17.781	106.472	213.763
	FRI + M61	0.834	5.013	9.987
GPT-2 Medium	KZG + BN254	12.553	75.222	150.850
	IPA + BN254	19.792	119.015	237.056
	FRI + M61	1.147	6.868	13.792
GPT-2 Large	KZG + BN254	41.254	247.955	494.312
	IPA + BN254	52.394	313.862	629.834
	FRI + M61	3.211	19.303	38.468

TABLE IV: OpenLLM end-to-end wall time (s) under single-thread execution for different numbers of Transformer layers (1, 6, and 12). Wall time includes witness preparation, commitment initialization, transcript handling, batch opening, lookup table setup, thread scheduling, and I/O processing.

Baseline. We compare our operators with the state-of-the-art work [3], which provides comprehensive ZK protocols for non-linear functions in machine learning. We evaluate prover time, verification time, and proof size across multiple input scales and LLM functions under the same experimental setup. In addition, we include comparisons with zkGPT [2] for end-to-end LLM inference to show OpenLLM at the system level.

Scheme	1024 (10^3)			16384 (10^4)			131072 (10^5)		
	Softmax	Norm.	GELU	Softmax	Norm.	GELU	Softmax	Norm.	GELU
OpenLLM	1.16	1.49	18.50	0.423	1.39	16.60	0.0171	1.30	29.30
zkGPT [2]	1.45	1.60	29600	0.526	2.22	6230	0.0175	961	5090
[3]	24.10	221	191	1.14	69.5	177	0.0153	283	182

TABLE V: Accuracy (MSE $\times 10^{-8}$) for Softmax, Normalization, and GELU across different input sizes. Lower values indicate a closer approximation to the original computation.

B. Experimental Results

Results of LLM functions. Table I reports the performance of different polynomial commitment schemes for representative non-linear functions, including GELU, Softmax, and Norm. The FRI-based construction consistently achieves the lowest prover time and verification latency across all input scales; for example, on GELU with input size 1024×1024, the prover time is reduced from 57.496s (KZG) and 315.272s (IPA) to 10.045s. In contrast, [3] incurs extremely large proofs (up to

Protocol Π_{Softmax}

Parameter. Public scale $s_1 = \frac{a_1}{b_1}$ and $s_2 = \frac{a_2}{b_2}$.
Input. A polynomial $A(\mathbf{X})$ to a vector $\vec{a} = (x_0, x_1, \ldots, x_{n-1})$ with size n under scale $s_1 = \frac{a_1}{b_1}$ such that $A(\langle i \rangle) = x_i$ for $\langle i \rangle \in \{0,1\}^{\log n}$.
Output. A polynomial $B(\mathbf{X})$ to a vector $\vec{b} = (y_0, y_1, \ldots, y_{n-1})$ with size n under scale $s_2 = \frac{a_2}{b_2}$ such that $B(\langle i \rangle) = y_i$ for $\langle i \rangle \in \{0,1\}^{\log n}$.
Protocol Execution.

1) **Step 1. Compute** $\mathsf{max}(x_0, x_1, \ldots, x_{n-1}) = x_{max}$**.**
   a) Define $t_A(\mathbf{X}) = x_{\mathsf{max}} - A(\mathbf{X})$.
   b) Run $(\mathsf{Cmp}, \frac{p+1}{2}, [t_A]; t_A(\mathbf{X})) \to ([Z_{t_A}]; Z_{t_A}(\mathbf{X}))$, and $(\mathsf{CheckOne}, [Z_{t_A}]; Z_{t_A})$ for $\mathbf{X} \in \{0,1\}^{\log n}$, to check $x_{\mathsf{max}} - x_i \geq 0$.
   c) Run $(\mathsf{ProdCheck}, 0, [t_A]; t_A(\mathbf{X}))$, to check at least one of the $x_{\mathsf{max}} - x_i = 0$.
2) **Step 2. Compute** e^{x_i} **and** $e^{t_i} = e^{x_{\mathsf{max}} - x_i}$**.**
   a) Choose $s_1 = \frac{a_{t_1}}{b_{t_1}}$ and $s_2 = \frac{a_{t_2}}{b_{t_2}}$.
   b) Run $(\mathsf{Exp}, a_1, b_1, a_{t_1}, b_{t_1}, [A]; A(\mathbf{X})) \to ([B_1]; B_1(\mathbf{X}))$.
   c) Run $(\mathsf{Exp}, a_1, b_1, a_{t_2}, b_{t_2}, [t_A]; t_A(\mathbf{X})) \to ([B_2]; B_2(\mathbf{X}))$.
3) **Step 3. Compute** $\vec{b} = (y_0, y_1, \ldots, y_{n-1})$ **such that** $y_i = \frac{e^{x_i}}{\sum_{i=0}^{n-1} e^{t_i}}$**.**
   a) Run $(\mathsf{SumCheck}, c, [B_2]; B_2(\mathbf{X}))$ such that $\sum_{\langle i \rangle \in \{0,1\}^{\log n}} B_2(\langle i \rangle) = c$.
   b) Run $(\mathsf{Requan}, a_{t_1}, b_{t_1}, a_4, b_4, [B_1]; B_1(\mathbf{X})) \to ([t_{B_1}]; t_{B_1}(\mathbf{X}))$ to let $t_B(\langle i \rangle) > B_2(\langle i \rangle)$ for $\langle i \rangle \in \{0,1\}^{\log n}$, to prevent the result of division equal to 0.
   c) Run $(\mathsf{Div}, a_4, b_4, a_{t_2}, b_{t_2}, [t_{B_1}], [B_2]; t_{B_1}(\mathbf{X}), B_2(\mathbf{X})) \to (a_2, b_2, [B]; B(\mathbf{X}))$.
4) Return $(a_2, b_2, [B]; B(\mathbf{X}))$.

Fig. 11: Protocol for Softmax Layer Π_{Softmax}

Protocol Π_{Norm}

Parameter. Public scale $s_1 = \frac{a_1}{b_1}$ and $s_2 = \frac{a_2}{b_2}$.
Input. A polynomial $A(\mathbf{X})$ to a vector $\vec{a} = (x_0, x_1, \ldots, x_{n-1})$ with size n under scale $s_1 = \frac{a_1}{b_1}$ such that $A(\langle i \rangle) = x_i$ for $\langle i \rangle \in \{0,1\}^{\log n}$.
Two polynomials $\gamma(\mathbf{X})$ and $\beta(\mathbf{X})$ to a vector $\vec{\gamma} = (\gamma_0, \gamma_1, \ldots, \gamma_{n-1})$ and $\vec{\beta} = (\beta_0, \beta_1, \ldots, \beta_{n-1})$ respectively under scale $s_\gamma = \frac{a_\gamma}{b_\gamma}$ and $s_\beta = \frac{a_\beta}{b_\beta}$ such that $\gamma(\langle i \rangle) = \gamma_i$ and $\beta(\langle i \rangle) = \beta_i$ for $\langle i \rangle \in \{0,1\}^{\log n}$.
Output. A polynomial $B(\mathbf{X})$ to a vector $\vec{b} = (y_0, y_1, \ldots, y_{n-1})$ with size n under scale $s_2 = \frac{a_2}{b_2}$ such that $B(\langle i \rangle) = y_i$ for $\langle i \rangle \in \{0,1\}^{\log n}$.
Protocol Execution.

1) **Step 1. Compute the mean** μ
   a) Run $(\mathsf{SumCheck}, \sum_{i=0}^{n-1} x_i, [A]; A(\mathbf{X}))$.
   b) Define $q_\mu = \lfloor \sum_{i=0}^{n-1} x_i / n \rceil$ with $s_\mu = \frac{a_\mu}{b_\mu} = \frac{a_1}{b_1}$.
2) **Step 2. Compute the variance** σ
   a) Define $\hat{A}(\mathbf{X}) = (A(\mathbf{X}) - q_\mu)^2$ for $\mathbf{X} \in \{0,1\}^{\log n}$.
   b) Run $(\mathsf{Cmp}, \frac{p+1}{2}, [\hat{A}]; \hat{A}(\mathbf{X})) \to ([Z_{\hat{A}}]; Z_{\hat{A}}(\mathbf{X}))$ and $(\mathsf{CheckOne}, [Z_{\hat{A}}]; Z_{\hat{A}}(\mathbf{X}))$.
   c) Run $(\mathsf{SumCheck}, \sum_{i=0}^{n-1} (x_i - q_\mu)^2, [\hat{A}]; \hat{A}(\mathbf{X}))$.
   d) Define $q_\sigma = \lfloor \sum_{i=0}^{n-1} (x_i - q_\mu)^2 / n \rceil$ with scale $s_\sigma = \frac{a_\sigma}{b_\sigma} = \frac{a_1^2}{b_1^2}$.
3) **Step 3. Compute** $y_i = \gamma_i \cdot \frac{x_i - \mu}{\sqrt{\sigma}} + \beta_i$**.**
   a) Compute $t_{q_\sigma} = \lfloor \sqrt{q_\sigma} \rceil$ with $s_{t_\sigma} = \frac{a_{t_\sigma}}{b_{t_\sigma}} = \frac{a_1}{b_1}$.
   b) Select $s_1' = \frac{a_1'}{b_1'}$, and run $(\mathsf{Requan}, a_1, b_1, a_1', b_1', [\hat{A}]; \hat{A}(\mathbf{X})) \to ([\tilde{A}]; \tilde{A}(\mathbf{X}))$, such that $\tilde{A}(\langle i \rangle) > t_{q_\sigma}$.
   c) Run $(\mathsf{Cmp}, t_{q_\sigma}, [\tilde{A}]; \tilde{A}(\mathbf{X})) \to ([Z_{\tilde{A}}]; Z_{\tilde{A}}(\mathbf{X}))$, and $(\mathsf{CheckOne}, [Z_{\tilde{A}}]; Z_{\tilde{A}}(\mathbf{X}))$.
   d) Define $s_{t_1} = \frac{a_{t_1}}{b_{t_1}} = \frac{a_1 \cdot b_{t_\sigma}}{b_1 \cdot a_{t_\sigma}}$, and $t_1(\mathbf{X}) \cdot t_{q_\sigma} + r_1(\mathbf{X}) = \tilde{A}(\mathbf{X})$, run $(\mathsf{Cmp}, t_{q_\sigma}, [r_1]; r_1(\mathbf{X})) \to ([Z_{r_1}]; Z_{r_1}(\mathbf{X}))$ and $(\mathsf{CheckOne}, [Z_r]; Z_r(\mathbf{X}))$.
   e) Define $s_{t_2} = s_\gamma \cdot s_{t_1} = \frac{a_{t_2}}{b_{t_2}} = \frac{a_\gamma \cdot a_{t_1}}{b_\gamma \cdot b_{t_1}}$, and $t_2(\mathbf{X}) = \gamma(\mathbf{X}) \cdot t_1(\mathbf{X})$.
   f) Run $(\mathsf{Add}, a_{t_2}, b_{t_2}, a_\beta, b_\beta, a_2, b_2, [t_2], [\beta]; t_2(\mathbf{X}), \beta(\mathbf{X})) \to ([B]; B(\mathbf{X}))$.
4) Return $(a_2, b_2, [B]; B(\mathbf{X}))$.

Fig. 12: Protocol for Normalization Layer Π_{Norm}

GB level) and high verification cost, while KZG- and IPA-based constructions provide a trade-off with smaller proof sizes but higher computational overhead.

Results of end-to-end comparison. Table II shows the end-to-end performance on GPT-2 models of increasing sizes. Compared with [2], our method significantly reduces both prover time and verification latency; for example, on GPT-2-Small, zkGPT requires 88.515s prover time and 4809.032ms verification time, while our FRI-based approach achieves 13.280s and 16.685ms, respectively. Moreover, the results demonstrate stable scalability with respect to model size.

Results on larger LLMs. Table III presents both end-to-end and single-layer inference performance on larger models. For SmolLM-1.7B, the single-layer prover time is 6.202s with a proof size of 0.037 MB, while the end-to-end inference cost remains practical. Similar trends are observed for Qwen-3-8B, demonstrating that the proposed framework scales beyond GPT-2. These results confirm the feasibility of our approach for larger LLMs and highlight the effectiveness of the modular design in handling increasing model complexity.

To further analyze system behavior, Table IV reports layer-wise wall time under a single-thread setting. The results show near-linear scaling with respect to the number of layers; for instance, for GPT-2-Small with FRI, the wall time increases from 0.834s (1 layer) to 9.987s (12 layers), indicating minimal overhead beyond the inherent model computation.

Accuracy. Table V reports the approximation accuracy under different quantization methods using mean squared error (MSE). OpenLLM consistently achieves the lowest MSE across all operators and input scales. For instance, on GELU, our method obtains MSE values of 18.50, 16.60, and 29.30, while [2] reports 29600, 6230, and 5090 under the same settings. These results indicate that our approach maintains high numerical fidelity and remains stable as input size increases.

Ethics Statements and Compliance with the Open Science Policy

Ethics Statements. This work does not involve human subjects, user data, or any interaction with external systems that could pose risks to individuals or organizations. We do not identify any immediate ethical concerns arising from this research. The techniques proposed are intended for improving the security and verifiability of large language model inference, and we do not foresee direct misuse that would lead to harm.

Open Science Policy. We fully adhere to the principles of the Open Science Policy and are committed to promoting transparency and reproducibility in scientific research. We anonymously open-source the code of this work [1].

Generative AI Usage

We used a generative AI tool (ChatGPT) to assist with drafting and revising portions of the paper text, primarily for improving clarity, organization, and presentation, including parts of the abstract and the description of experimental results. The AI tool was not used to design the methodology, conduct experiments, generate data and content, or draw scientific conclusions. All technical content, experimental results, and claims were produced and verified by the authors.

References

[1] A. Riasi, H. Wang, R. Behnia, V. Vo, and T. Hoang, "Zero-knowledge AI inference with high precision," in *Proc. 2025 ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, 2025, pp. 1053–1067.

[2] W. Qu, Y. Sun, X. Liu, T. Lu, Y. Guo, K. Chen, and J. Zhang, "zkGPT: An efficient non-interactive zero-knowledge proof framework for LLM inference," in *Proc. 34th USENIX Security Symp. (USENIX Security 25)*, 2025, pp. 2045–2063.

[3] M. Hao, H. Chen, H. Li, C. Weng, Y. Zhang, H. Yang, and T. Zhang, "Scalable zero-knowledge proofs for non-linear functions in machine learning," in *Proc. 33rd USENIX Security Symp. (USENIX Security 24)*, 2024, pp. 3819–3836.

[4] A. Arun, S. Setty, and J. Thaler, "Jolt: SNARKs for virtual machines via lookups," in *Advances in Cryptology–EUROCRYPT 2024*, Part VI, Lecture Notes in Computer Science, vol. 14656. Cham, Switzerland: Springer, 2024, pp. 3–33.

[5] B. Chen, B. Bünz, D. Boneh, and Z. Zhang, "HyperPlonk: Plonk with linear-time prover and high-degree custom gates," in *Advances in Cryptology–EUROCRYPT 2023*, Part II, Lecture Notes in Computer Science, vol. 14005. Cham, Switzerland: Springer, 2023, pp. 499–530.

[6] B. Bünz, B. Fisch, and A. Szepieniec, "Transparent SNARKs from DARK compilers," in *Advances in Cryptology–EUROCRYPT 2020*, Part I, Lecture Notes in Computer Science, vol. 12105. Cham, Switzerland: Springer, 2020, pp. 677–706.

[7] A. Gabizon, Z. J. Williamson, and O. Ciobotaru, "PLONK: Permutations over Lagrange-bases for oecumenical noninteractive arguments of knowledge," *Cryptology ePrint Archive*, Paper 2019/953, 2019.

[8] A. Chiesa, Y. Hu, M. Maller, P. Mishra, N. Vesely, and N. Ward, "Marlin: Preprocessing zkSNARKs with universal and updatable SRS," in *Advances in Cryptology–EUROCRYPT 2020*, Part I, Lecture Notes in Computer Science, vol. 12105. Cham, Switzerland: Springer, 2020, pp. 738–768.

[9] V. Costan and S. Devadas, "Intel SGX explained," *Cryptology ePrint Archive*, Paper 2016/086, 2016.

[10] Z. Ghodsi, T. Gu, and S. Garg, "SafetyNets: Verifiable execution of deep neural networks on an untrusted cloud," in *Advances in Neural Information Processing Systems 30 (NIPS 2017)*, 2017, pp. 4672–4681.

[11] S. Lee, H. Ko, J. Kim, and H. Oh, "vCNN: Verifiable convolutional neural network based on zk-SNARKs," *IEEE Trans. Dependable Secure Comput.*, vol. 21, no. 4, pp. 4254–4270, Jul./Aug. 2024.

[12] H. Wang and T. Hoang, "ezDPS: An efficient and zero-knowledge machine learning inference pipeline," *arXiv preprint arXiv:2212.05428*, 2022.

[13] B. Feng, L. Qin, Z. Zhang, Y. Ding, and S. Chu, "ZEN: An optimizing compiler for verifiable, zero-knowledge neural network inferences," *Cryptology ePrint Archive*, Paper 2021/087, 2021.

[14] K. Abbaszadeh, C. Pappas, J. Katz, and D. Papadopoulos, "Zero-knowledge proofs of training for deep neural networks," in *Proc. 2024 ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, 2024, pp. 4316–4330.

[15] S. Garg, A. Goel, S. Jha, S. Mahloujifar, M. Mahmoody, G.-V. Policharla, and M. Wang, "Experimenting with zero-knowledge proofs of training," in *Proc. 2023 ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, 2023, pp. 1880–1894.

[16] T. Liu, X. Xie, and Y. Zhang, "zkCNN: Zero knowledge proofs for convolutional neural network predictions and accuracy," in *Proc. 2021 ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, 2021, pp. 2968–2985.

[17] A. Riasi, J. Guajardo, and T. Hoang, "Privacy-preserving verifiable neural network inference service," in *Proc. 2024 Annu. Comput. Security Appl. Conf. (ACSAC)*, 2024, pp. 683–698.

[18] U. Haböck, "Multivariate lookups based on logarithmic derivatives," *Cryptology ePrint Archive*, Paper 2022/1530, 2022.

[19] B.-J. Chen, S. Waiwitlikhit, I. Stoica, and D. Kang, "ZKML: An optimizing system for ML inference in zero-knowledge proofs," in *Proc. 19th Eur. Conf. Comput. Syst. (EuroSys)*, 2024, pp. 560–574.

[20] B. Feng, Z. Wang, Y. Wang, S. Yang, and Y. Ding, "ZENO: A type-based optimization framework for zero knowledge neural network inference," in *Proc. 29th ACM Int. Conf. Architectural Support Program. Lang. Oper. Syst. (ASPLOS), Vol. 1*, 2024, pp. 450–464.

[21] T. Xie, T. Lu, Z. Fang, S. Wang, Z. Zhang, Y. Jia, D. Song, and J. Zhang, "zkPyTorch: A hierarchical optimized compiler for zero-knowledge machine learning," *Cryptology ePrint Archive*, Paper 2025/535, 2025.

[22] A. Kate, G. M. Zaverucha, and I. Goldberg, "Constant-size commitments to polynomials and their applications," in *Advances in Cryptology–ASIACRYPT 2010*, Lecture Notes in Computer Science, vol. 6477. Berlin, Germany: Springer, 2010, pp. 177–194.

[23] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell, "Bulletproofs: Short proofs for confidential transactions and more," in *Proc. 2018 IEEE Symp. Security Privacy (SP)*, 2018, pp. 315–334.

[24] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, "Fast Reed-Solomon interactive oracle proofs of proximity," in *Proc. 45th Int. Colloq. Automata, Lang., Program. (ICALP 2018)*, Leibniz International Proceedings in Informatics, vol. 107. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2018, pp. 14:1–14:17.

[25] S. Setty, J. Thaler, and R. Wahby, "Unlocking the lookup singularity with Lasso," in *Advances in Cryptology–EUROCRYPT 2024*, Part VI, Lecture Notes in Computer Science, vol. 14656. Cham, Switzerland: Springer, 2024, pp. 180–209.

[1] https://anonymous.4open.science/r/OpenLLM-zkp-Anonymous-348A

APPENDIX

A. PRELIMINARIES

Due to the space constraints, we define the interface of each component in Table VI.

A. Multivariate Polynomial

For a function $f : \{0,1\}^\mu \to \mathbb{F}$, there exists a unique multilinear polynomial $\tilde{f} \in \mathbb{F}[X_1, \ldots, X_\mu]$ such that $\tilde{f}(\mathbf{b}) = f(\mathbf{b})$ for all $\mathbf{b} \in \{0,1\}^\mu$. We call $\tilde{f}$ the multilinear extension of f. Moreover, $\tilde{f}(\mathbf{X}) = \sum_{\mathbf{b}\in\{0,1\}^\mu} f(\mathbf{b}) \cdot \mathrm{eq}(\mathbf{b}, \mathbf{X})$, where $\mathrm{eq}(\mathbf{b}, \mathbf{X}) := \prod_{i=1}^{\mu} \big(b_i X_i + (1-b_i)(1-X_i)\big)$.

B. Vector and Matrix Representation

For a matrix A with row size m and column size n, the multivariate polynomial representation of A is $\hat{A}(X, Y)$ for each $\langle i\rangle \in \{0,1\}^{\log m}$ and $\langle j\rangle \in \{0,1\}^{\log n}$, such that $\hat{A}(\langle i\rangle, \langle j\rangle) = A_{1,1}$, in which $\langle i\rangle$ is a binary encoding of i.

In LLM, all computations are performed over a vector. Therefore, we need to iterate over each row of a matrix. For a multivariate polynomial $\hat{A}(X, Y)$ of matrix A, we fix the row i and then iterate over j, and in this case we have $\hat{A}(\langle i\rangle, \langle j\rangle)$ for all $j \in \{0,1\}^{\log n}$. Then, we set the multivariate representation of the vector $\vec{t}$ as $t(Y) = \hat{A}(\langle i\rangle, Y)$ for $Y \in \{0,1\}^{\log n}$.

C. Multivariate Sumcheck Protocol

The multivariate sumcheck is to prove $c = \sum_{\vec{X}\in\{0,1\}^k} f(X_1, \ldots, X_k)$, in which $f(X_1, X_2, \ldots, X_k)$ is an k-variate polynomial. It runs for k rounds. In round j, the prover sends the univariate polynomial $h_j(X_j) = \sum_{x_{j+1},\ldots,x_k\in\{0,1\}} f(r_1, \ldots, r_{j-1}, X_j, x_{j+1}, \ldots, x_k)$. The verifier checks that the degree of $h_j(X_j)$ and enforces the consistency equation $h_1(0) + h_1(1) = c$, and for $j \geq 2$, the verifier checks if $h_j(0) + h_j(1) = h_{j-1}(r_{j-1})$. The verifier then samples a random challenge $r_j \leftarrow \mathbb{F}$, sets $T \leftarrow h_j(r_j)$, and fixes $X_j = r_j$. After k rounds, the claim is reduced to a single evaluation check $f(r_1, \ldots, r_k) = c$.

To combine two instances $c_1 = \sum_{\vec{X}\in\{0,1\}^n} f_1(X_1, \ldots, X_n)$ and $c_2 = \sum_{\vec{X}\in\{0,1\}^n} f_2(X_1, \ldots, X_n)$ into one, a verifier first sends a random challenge β, and then the prover runs sumcheck protocol on $\sum_{\vec{X}\in\{0,1\}^n} f_1(X_1, \ldots, X_n) + \beta \cdot f_2(X_1, \ldots, X_n) = c_1 + \beta \cdot c_2$.

D. Reduction of Basic PIOPs

This subsection introduces several basic PIOPs that can be reduced to the multivariate sumcheck protocol.

Logup Lookup Protocol. We adopt the LogUp lookup argument [18] to prove that a query multiset A is contained in a fixed lookup table B. LogUp represents A and B as multilinear extensions $f_A : \{0,1\}^{\mu_A} \to \mathbb{F}$ and $f_B : \{0,1\}^{\mu_B} \to \mathbb{F}$, which can differ in size. A multiplicity map $m_B : \{0,1\}^{\mu_B} \to \mathbb{F}$ records how many times each table value is used.

The verifier samples a random challenge $\zeta \in \mathbb{F}$, and the prover commits to auxiliary polynomials h_A and h_B, such that $h_A(x)(\zeta - f_A(x)) = 1$ and $h_B(y)(\zeta - f_B(y)) = m_B(y)$, and satisfies $\sum_x h_A(x) = \sum_y h_B(y)$, for all $x \in \{0,1\}^{\mu_A}$ and $y \in \{0,1\}^{\mu_B}$. Furthermore, the logup lookup protocol will be reduced to the sumcheck protocol.

In addition, we denote $(L, A(X))$ to indicate that all elements in $A(X)$ belong to the lookup table L. If L contains n columns, we write $(L, A_1(X), A_2(X), \ldots, A_n(X))$ to represent that each column $A_i(X)$ is matched against a specific column of L. In practice, we can aggregate all columns of L into a single one using a random linear combination, apply the same aggregation to $A_1(X), \ldots, A_n(X)$, and then perform a single Logup lookup argument.

Zero Check. To prove that a polynomial $f(\mathbf{X})$ vanishes over the Boolean hypercube, the ZeroCheck protocol constructs a masked polynomial $\hat{f}(\mathbf{X}) = f(\mathbf{X}) \cdot \mathrm{eq}(\mathbf{X}, \mathbf{r})$, where $\mathbf{r} \in \mathbb{F}^\mu$ is a random point sampled by the verifier and $\mathrm{eq}(\mathbf{X}, \mathbf{r})$ is the multilinear equality polynomial satisfying $\mathrm{eq}(\mathbf{X}, \mathbf{r}) = 1$ if and only if $\mathbf{X} = \mathbf{r}$. This transformation ensures that only the evaluation at $\mathbf{r}$ remains in the sum. The prover then runs a sumcheck protocol to convince the verifier that $\sum_{\mathbf{X}\in\{0,1\}^\mu} \hat{f}(\mathbf{X}) = 0$, which implies $f(\mathbf{r}) = 0$ and thereby that f vanishes over the Boolean domain with high probability.

Product Check. HyperPlonk [5] gives a product check over a rational polynomial, such that $\prod_{\mathbf{X}\in\{0,1\}^n} \frac{f_1(\mathbf{X})}{f_2(\mathbf{X})} = \prod_{\mathbf{X}\in\{0,1\}^n} f(\mathbf{X}) = s$. To apply the product check to a single polynomial $f(\mathbf{X})$, we simply set $f_1(\mathbf{X}) = f(\mathbf{X})$ and $f_2(\mathbf{X}) = 1$. Thus, f_2 is the constant polynomial 1 (degree 0) and is rated as 1 for all inputs, so the 'denominator nonzero' check is automatically satisfied. After that, the protocol is simplified as follows.

1) Define $\tilde{v}(0, \mathbf{X}) = f(\mathbf{X})$.
2) Define $\tilde{v}(1, \mathbf{X}) = \tilde{v}(\mathbf{X}, 0) \cdot \tilde{v}(\mathbf{X}, 1)$.
3) Define $\hat{h}(\mathbf{X}')$ with degree $2n$ such that $\hat{h} = \mathsf{merge}(\hat{f}, \hat{g})$, such that $\hat{g}(\mathbf{X}) = \tilde{v}(0, \mathbf{X}) - f(\mathbf{X})$, and $\hat{f}(\mathbf{X}) = \tilde{v}(1, \mathbf{X}) - \tilde{v}(\mathbf{X}, 0)\tilde{v}(\mathbf{X}, 1)$.
4) Run Zero Check on $\hat{h}(\mathbf{X}')$, and check if $\tilde{v}(1, \ldots, 1, 0) = s$.

Matrix Multiplication. For matrix multiplication, we define

$$\widetilde{C}(\mathbf{u}, \mathbf{v}) = \sum_{\mathbf{y}\in 0,1^{\log N}} \widetilde{A}(\mathbf{u}, \mathbf{y}), \widetilde{B}(\mathbf{y}, \mathbf{v}),$$

where $\widetilde{C}$, $\widetilde{A}$, and $\widetilde{B}$ denote the multilinear extensions of matrices C, A, and B, respectively, and $\mathbf{u}, \mathbf{v}$ are random challenges. When one of the matrices is a vector, the computation becomes a matrix–vector multiplication, expressed as $\tilde{a}(\mathbf{r}) = \sum_{\mathbf{y}\in 0,1^{\log N}} \widetilde{M}(\mathbf{r}, \mathbf{y}) \cdot \tilde{t}(\mathbf{y})$, where $\tilde{a}$, $\tilde{t}$, and $\widetilde{M}$ are the multilinear extensions of a vector a, a vector t, and a matrix M, respectively.

B. SECURITY PROOF OF BUILDING BLOCKS

Theorem 1. *The protocol Π_{DigitDec} as shown in Figure 2 satisfies completeness and knowledge soundness.*

Completeness. Suppose the prover is honest. For each input value $x_i \in \{0, \ldots, 2^{kd} - 1\}$, the prover computes its digit decomposition $x_i = \sum_{j=0}^{k-1} 2^{jd} x_i^{(j)}$, $x_i^{(j)} \in \{0, \ldots, 2^d - 1\}$. Then the prover defines a multilinear polynomial $B(\mathbf{X}, \mathbf{Y})$ such that $B(\langle i\rangle, \langle j\rangle) = x_i^{(j)}$ for all i, j, and extends it multilinear extension to the domain $\{0,1\}^{\log n} \times \{0,1\}^{\log k}$.

	Definition	Relation
Basic PIOPs		
Sumcheck	$(\mathsf{Sumcheck}, c, [A]; A(\mathbf{X}))$	$\sum_{X\in\{0,1\}^{\log n}} A(\mathbf{X}) = c$
Product Check	$(\mathsf{ProdCheck}, c, [A]; A(\mathbf{X}))$	$\prod_{X\in\{0,1\}^{\log n}} A(\mathbf{X}) = c$
ZeroCheck	$(\mathsf{ZeroCheck}, [A]; A(\mathbf{X}))$	$A(\mathbf{X}) = 0$ for $X \in \{0,1\}^{\log n}$
Lookup	$(\mathsf{Lookup}, L, [A_1], \ldots, [A_n]; A_1(\mathbf{X}), \ldots, A_n(\mathbf{X}))$	$\sum_{i=1}^{n} \beta^{i-1} \cdot A_i \in \sum_{i=1}^{n} \beta^{i-1} \cdot L_i$
Building Blocks (For $\langle i\rangle \in \{0,1\}^{\log n}$ and $\langle j\rangle \in \{0,1\}^{\log k}$)		
Digital Decomposition	$(\mathsf{DigitDec}, [A]; A(\mathbf{X})) \to ([B]; B(\mathbf{X}, \mathbf{Y}))$	$A(\langle i\rangle) = B(\langle i\rangle, \langle k-1\rangle) \| \ldots \| B(\langle i\rangle, \langle 0\rangle)$
Comparison	$(\mathsf{Cmp}, c, [A]; A(\mathbf{X})) \to ([Z_A]; Z_A(\mathbf{X}))$	If $A(\langle i\rangle) < c$, then $Z_A(\langle i\rangle) = 1$, else $Z_A(\langle i\rangle) = 0$
CheckOne	$(\mathsf{CheckOne}, [Z_A]; Z_A(\mathbf{X})) \to \mathsf{accept/reject}$	If $Z_A(\langle i\rangle) = 1$, output accept; else output reject
Requantization	$(\mathsf{Requan}, a_1, b_1, a_2, b_2, [A]; A(\mathbf{X})) \to ([B]; B(\mathbf{X}))$	Transform $A(\langle i\rangle)$ under $\frac{a_1}{b_1}$ to $B(\langle i\rangle)$ under $\frac{a_2}{b_2}$
Math (For $\langle i\rangle \in \{0,1\}^{\log n}$ and $\langle j\rangle \in \{0,1\}^{\log k}$)		
Square Root	$(\mathsf{Sqrt}, a_1, b_1, [q_1]; q_1(\mathbf{X})) \to (a_2, b_2, [q_2]; q_2(\mathbf{X}))$	$\frac{a_1}{b_1} \cdot q_1(\langle i\rangle) \approx \sqrt{\frac{a_2}{b_2} \cdot q_2(\langle i\rangle)}$
Addition	$(\mathsf{Add}, a_1, b_1, a_2, b_2, a_3, b_3, [A], [B]; A(\mathbf{X}), B(\mathbf{X})) \to ([C]; C(\mathbf{X}))$	$\frac{a_1}{b_1} A(\langle i\rangle) + \frac{a_2}{b_2} B(\langle i\rangle) \approx \frac{a_3}{b_3} C(\langle i\rangle)$
Division	$(\mathsf{Div}, a_1, b_1, a_2, b_2, [A], [B]; A(\mathbf{X}), B(\mathbf{X})) \to (a_3, b_3, [C]; C(\mathbf{X}))$	$\frac{a_1}{b_1} A(\langle i\rangle) / \frac{a_2}{b_2} B(\langle i\rangle) \approx \frac{a_3}{b_3} C(\langle i\rangle)$
Exponential	$(\mathsf{Exp}, a_1, b_1, a_2, b_2, [A]; A(\mathbf{X})) \to ([B]; B(\mathbf{X}))$	$\frac{a_2}{b_2} B(\langle i\rangle) \approx \mathsf{Exp}(\frac{a_1}{b_1} A(\langle i\rangle))$
Function (For $\langle i\rangle \in \{0,1\}^{\log n}$ and $\langle j\rangle \in \{0,1\}^{\log k}$)		
GELU	$(\mathsf{GELU}, a_1, b_1, [A]; A(\mathbf{X})) \to (a_2, b_2, [B]; B(\mathbf{X}))$	$\frac{a_2}{b_2} B(\langle i\rangle) \approx \mathsf{GELU}(\frac{a_1}{b_1} A(\langle i\rangle))$
Softmax	$(\mathsf{Softmax}, a_1, b_1, [A]; A(\mathbf{X})) \to (a_2, b_2, [B]; B(\mathbf{X}))$	$\frac{a_2}{b_2} B(\langle i\rangle) \approx \mathsf{Softmax}(\frac{a_1}{b_1} A(\langle i\rangle))$
Normalize	$(\mathsf{Norm}, a_1, b_1, a_\gamma, b_\gamma, a_\beta, b_\beta, [A], [\gamma], [\beta]; A(\mathbf{X}), \gamma(\mathbf{X}), \beta(\mathbf{X})) \to (a_2, b_2, [B]; B(\mathbf{X}))$	$\frac{a_2}{b_2} B(\langle i\rangle) \approx \mathsf{Norm}(\frac{a_1}{b_1} A(\langle i\rangle))$

TABLE VI: Interface of Each Component.

We now verify the checks of the protocol.

Step 2(b). For each i, the prover sets $t_i(\mathbf{Y}) = B(\langle i\rangle, \mathbf{Y})$. Thus, for any random $\mathbf{r}_1$, the equality $t_i(\mathbf{r}_1) = B(\langle i\rangle, \mathbf{r}_1)$ holds deterministically.

Step 2(c). For each $\langle j\rangle$, we have $t_i(\langle j\rangle) = B(\langle i\rangle, \langle j\rangle) = x_i^{(j)} \in L_1$.

Step 3. Since $\widehat{e}(\langle j\rangle) = 2^{jd}$, we obtain

$$\sum_{\mathbf{Y}} B(\mathbf{X}, \mathbf{Y})\, \widehat{e}(\mathbf{Y}) = \sum_{j=0}^{k-1} B(\mathbf{X}, \langle j\rangle)\, 2^{jd} = \sum_{j=0}^{k-1} x^{(j)}(\mathbf{X})\, 2^{jd}.$$

By construction, for every $\mathbf{X} = \langle i\rangle$ this sum equals $x_i = A(\langle i\rangle)$, and by multilinearity the equality holds on the entire hypercube. Therefore, for any random $\mathbf{r}_2$ the sumcheck verification passes.

Knowledge Soundness. Assume the lookup argument is knowledge-sound, the commitment to B is binding and extractable, and sumcheck is sound. Let $\mathcal{E}$ be the extractor.

From Step 2(c), by lookup knowledge soundness, $\mathcal{E}$ obtains values $\tilde{x}_i^{(j)} \in \{0, \ldots, 2^d - 1\}$. Step 2(b) together with commitment binding ensures that $t_i(\mathbf{Y})$ and $B(\langle i\rangle, \mathbf{Y})$ coincide as multilinear polynomials, hence $B(\langle i\rangle, \langle j\rangle) = \tilde{x}_i^{(j)}$. Consider $F(\mathbf{X}) = \sum_{\mathbf{Y}} B(\mathbf{X}, \mathbf{Y})\, \widehat{e}(\mathbf{Y}) - A(\mathbf{X})$. If $F \not\equiv 0$, then by sumcheck soundness the verifier accepts with at most negligible probability. Since the prover is accepted with non-negligible probability, we conclude $F \equiv 0$. Evaluating at $\mathbf{X} = \langle i\rangle$ gives $A(\langle i\rangle) = \sum_{j=0}^{k-1} B(\langle i\rangle, \langle j\rangle)\, 2^{jd} = \sum_{j=0}^{k-1} \tilde{x}_i^{(j)}\, 2^{jd}$. Thus $\mathcal{E}$ extracts a valid decomposition $\{\tilde{x}_i^{(j)}\}$ satisfying the relation. Therefore, Π_{DigitDec} is knowledge-sound.

Theorem 2. *The protocol Π_{Cmp} as shown in Figure 3 satisfies completeness and knowledge soundness.*

Completeness. Suppose the prover is honest and follows the protocol specification. Given $A(\mathbf{X})$ encoding $\vec{a} = (x_0, \ldots, x_{n-1})$, the prover first runs Π_{DigitDec} to obtain a correct digit decomposition $B(\mathbf{X}, \mathbf{Y})$ such that $B(\langle i\rangle, \langle j\rangle) = x_i^{(j)}$. By completeness of Π_{DigitDec}, this step succeeds.

Step 2. For each digit $x_i^{(j)}$ and the public threshold digit $c^{(j)}$, the prover correctly computes $Z(\langle i\rangle, \langle j\rangle) = 2^{k+j} \cdot 1\{x_i^{(j)} < c^{(j)}\} + 2^j \cdot 1\{x_i^{(j)} = c^{(j)}\}$, so that Z encodes the lexicographic comparison. The checks at Step 2(b) and 2(c) hold deterministically, because $t_{B_j}(\mathbf{Y}) = B(\langle j\rangle, \mathbf{Y})$ and $t_{Z_j}(\mathbf{Y}) = Z(\langle j\rangle, \mathbf{Y})$. The lookup checks in Step 2(d) succeed since the values lie in the lookup table $L_1^{(j)}$ by construction.

Step 3. Define $\hat{Z}(\mathbf{X}) = \sum_{j=0}^{k-1} Z(\mathbf{X}, \langle j\rangle)$. For any $\langle i\rangle$, the sum $\hat{Z}(\langle i\rangle)$ correctly encodes whether $x_i < c$. Thus, the sumcheck in Step 3(b) passes, and the polynomial $t_y(\mathbf{X})$ correctly outputs $t_y(\langle i\rangle) = 1$ if $x_i < c$, otherwise 0. The lookup verification in Step 3(d) also succeeds by construction.

Knowledge Soundness. Assume that Π_{DigitDec}, the lookup argument, and the sumcheck protocol are all knowledge-sound, and that the commitments to $B(\mathbf{X})$, $Z(\mathbf{X}, \mathbf{Y})$, and $t_y(\mathbf{X})$ are binding and extractable. Suppose a prover convinces the verifier with non-negligible probability.

From Step 1, by knowledge soundness of Π_{DigitDec}, the extractor obtains digits $\tilde{x}_i^{(j)} \in \{0, \ldots, 2^d - 1\}$ such that $A(\langle i\rangle) = \sum_{j=0}^{k-1} 2^{jd}\, \tilde{x}_i^{(j)}$. Hence the extractor has the canonical digit decomposition of each x_i.

From Step 2, the extractor obtains values $Z(\langle i\rangle, \langle j\rangle)$ consistent with the lookup tables $L_1^{(j)}$, certifying that each digit comparison with $c^{(j)}$ is correct. Commitment binding and random-point checks ensure these Z values coincide with the

committed polynomial.

In Step 3, the sumcheck guarantees $\hat{Z}(\mathbf{X}) = \sum_{j=0}^{k-1} Z(\mathbf{X}, \langle j \rangle)$, and the extractor learns that $t_y(\mathbf{X})$ is consistent with $\hat{Z}(\mathbf{X})$, so that $t_y(\langle i \rangle) = 1$ iff $x_i < c$ and 0 otherwise.

Theorem 3. *The protocol Π_{CheckOne} for CheckOne Figure 4 satisfies completeness and knowledge soundness.*

Completeness. Suppose the prover is honest and provides a polynomial $y(\mathbf{X})$ such that $y(\langle i \rangle) = 1$ for every $i \in \{0, \ldots, n-1\}$. The verifier samples a random challenge α and defines $\hat{\alpha}(\mathbf{X})$ as the multilinear extension of the map $\langle i \rangle \mapsto \alpha^i$.

By construction,

$$\sum_{\mathbf{x} \in \{0,1\}^{\log n}} y(\mathbf{x}) \cdot \hat{\alpha}(\mathbf{x}) = \sum_{i=0}^{n-1} y(\langle i \rangle) \cdot \alpha^i$$

$$= \sum_{i=0}^{n-1} 1 \cdot \alpha^i = 1 + \alpha + \alpha^2 + \cdots + \alpha^{n-1}$$

Therefore the polynomial identity checked in Step 2 is always correct, and the sumcheck verification accepts with probability 1. Hence the protocol is complete.

Knowledge Soundness. Assume the sumcheck protocol is sound: if the claimed identity is false, then the acceptance probability is at most negligible in λ. Suppose a prover convinces the verifier with non-negligible probability.

Define the polynomial $F(\alpha) = \sum_{\mathbf{x} \in \{0,1\}^{\log n}} y(\mathbf{x}) \cdot \hat{\alpha}(\mathbf{x}) - \left(1 + \alpha + \alpha^2 + \cdots + \alpha^{n-1}\right)$. If there exists some i such that $y(\langle i \rangle) \neq 1$, then $F(\alpha)$ is a non-zero polynomial in α of degree at most $n-1$. For a random challenge $\alpha \in \mathbb{F}$, the probability that $F(\alpha) = 0$ is at most $(n-1)/|\mathbb{F}|$, which is negligible when the field is large. In this case, the sumcheck protocol would reject with overwhelming probability. Therefore, acceptance with non-negligible probability implies that $y(\langle i \rangle) = 1$ holds.

Theorem 4. *The protocol Π_{Requan} as shown in Figure 5 satisfies completeness and knowledge soundness.*

Completeness. Suppose the prover is honest. Given $A(\mathbf{X})$ encoding $\vec{a} = (x_0, \ldots, x_{n-1})$ under the source scale $s_1 = \frac{a_1}{b_1}$, the prover computes the target vector $\vec{b} = (y_0, \ldots, y_{n-1})$ under the destination scale $s_2 = \frac{a_2}{b_2}$ by solving $a_2 \cdot y_i + r_i = b_2 \cdot a_1 \cdot x_i, \quad 0 \leq r_i < b_1 \cdot a_2$, for each i. Define the multilinear extensions $B(\mathbf{X})$ and $r(\mathbf{X})$ such that $a_2 \cdot B(\langle i \rangle) + r(\langle i \rangle) = b_2 \cdot a_1 \cdot A(\langle i \rangle)$.

Step 2. Since $0 \leq r_i < b_1 \cdot a_2$ for each i, the comparison protocol Cmp verifies the bound correctly and outputs $z_r(\langle i \rangle) = 1$ for all i. Then the CheckOne protocol confirms that indeed $z_r(\mathbf{X}) \equiv 1$ on the hypercube.

Knowledge Soundness. Assume that both sub-protocols Cmp and CheckOne are knowledge-sound, and that commitments to $B(\mathbf{X})$ and $r(\mathbf{X})$ are binding and extractable. Suppose a prover convinces the verifier with non-negligible probability.

From the binding property, there exist unique multilinear extensions $B(\mathbf{X})$ and $r(\mathbf{X})$ consistent with the commitments. Step 2(a) enforces the relation $a_2 \cdot B(\langle i \rangle) + r(\langle i \rangle) = b_2 \cdot a_1 \cdot A(\langle i \rangle)$, so the extractor $\mathcal{E}$ recovers $B(\langle i \rangle)$ and $r(\langle i \rangle)$ that satisfy this identity.

By knowledge soundness of Cmp, the extractor learns that each remainder value is bounded as $0 \leq r(\langle i \rangle) < b_1 \cdot a_2$. By knowledge soundness of CheckOne, the extractor confirms that the verification polynomial $z_r(\mathbf{X})$ is identically 1, certifying the bound holds for all indices.

Consequently, $\mathcal{E}$ outputs a valid witness consisting of $B(\mathbf{X})$ and $r(\mathbf{X})$ such that the required relation and range condition are satisfied.

C. Security Proof of Mathematic Functions

Theorem 5. *The protocol Π_{Sqrt} as shown in Figure 6 satisfies completeness and knowledge soundness.*

Completeness. In Step 1, running Π_{DigitDec} produces a correct digit decomposition $B(\mathbf{X}, \mathbf{Y})$ such that $B(\langle i \rangle, \langle j \rangle) = x_i^{(j)}$ and $A(\langle i \rangle) = \sum_{j=0}^{k-1} 2^{jd} x_i^{(j)}$.

In Step 2, the prover defines $Z(\langle i \rangle, \langle j \rangle) = 2^{k+j} \cdot 1\{x_i^{(j)} < c^{(j)}\} + 2^j \cdot 1\{x_i^{(j)} = c^{(j)}\}$. This correctly encodes the digit-wise comparison between $x_i^{(j)}$ and $c^{(j)}$. Checks at Step 2(b) and 2(c) pass since $t_{B_j}(\mathbf{Y}) = B(\langle j \rangle, \mathbf{Y})$ and $t_{Z_j}(\mathbf{Y}) = Z(\langle j \rangle, \mathbf{Y})$ by construction. The lookup checks at Step 2(d) succeed because the values are in the public tables $L_1^{(j)}$.

In Step 3, the polynomial $\hat{Z}(\mathbf{X}) = \sum_{j=0}^{k-1} Z(\mathbf{X}, \langle j \rangle)$ is formed. For any $\langle i \rangle$, $\hat{Z}(\langle i \rangle)$ determines whether $x_i < c$. The sumcheck verification passes, and the polynomial $t_y(\mathbf{X})$ is defined so that $t_y(\langle i \rangle) = 1$ if $x_i < c$ and 0 otherwise. The lookup in Step 3(d) also succeeds.

Knowledge Soundness. Assume Π_{DigitDec}, lookup arguments, and sumcheck are knowledge-sound, and that commitments are binding and extractable. Let the prover convince the verifier with non-negligible probability.

From Step 1, the extractor of Π_{DigitDec} outputs digits $\tilde{x}_i^{(j)} \in \{0, \ldots, 2^d - 1\}$ with $A(\langle i \rangle) = \sum_{j=0}^{k-1} 2^{jd} \tilde{x}_i^{(j)}$, thus recovering the correct decomposition.

From Step 2, the extractor obtains values $Z(\langle i \rangle, \langle j \rangle)$ that are certified by the lookup tables $L_1^{(j)}$ to represent the comparison outcomes of $\tilde{x}_i^{(j)}$ against $c^{(j)}$. Binding of commitments and random-point checks ensure consistency of Z.

In Step 3, the sumcheck guarantees $\hat{Z}(\mathbf{X}) = \sum_{j=0}^{k-1} Z(\mathbf{X}, \langle j \rangle)$, and the extractor learns that $t_y(\mathbf{X})$ matches this relation. Hence $t_y(\langle i \rangle) = 1$ iff $x_i < c$, and 0 otherwise.

Thus, from any accepting transcript the extractor recovers valid digits $\{\tilde{x}_i^{(j)}\}$ and comparison results $\{t_y(\langle i \rangle)\}$ satisfying the intended relation.

Theorem 6. *The protocol Π_{Exp} as shown in Figure 7 satisfies completeness and knowledge soundness.*

Completeness. Assume the prover is honest and all sub-protocols are run as specified.

Step 1 (sign and $|x_i|$). Running Π_{Cmp} with threshold $(p+1)/2$ yields a Boolean polynomial $Z_A(\mathbf{X})$ such that $Z_A(\langle i \rangle) = 1$ iff $x_i < 0$ (in the canonical representative in $\mathbb{F}_p$), and 0 otherwise. Define $\widehat{A}(\mathbf{X}) = (2Z_A(\mathbf{X}) - 1) \cdot A(\mathbf{X})$; then for every i, $\widehat{A}(\langle i \rangle) = |x_i|$. By completeness of Π_{Cmp}, Step 1 succeeds.

Step 2 (digit decomposition and per–digit lookups). Run Π_{DigitDec} on $\widehat{A}$ to obtain $B(\mathbf{X},\mathbf{Y})$ with $B(\langle i\rangle,\langle j\rangle) = |x_i|^{(j)}$, the j-th d-bit digit of $|x_i|$. For each j, the prover forms $\widehat{C}(\mathbf{X},\mathbf{Y})$ with $\widehat{C}(\langle i\rangle,\langle j\rangle) = \left\lfloor \exp\!\left(|x_i|^{(j)}\cdot 2^{jd}\cdot\frac{a_1}{b_1}\right) \Big/ \frac{a_2}{b_2}\right\rfloor$, and proves correctness against the public lookup table $L_1^{(j)}$ using the standard lookup argument. The random-point equalities $t_C(\mathbf{r}_1) = \widehat{C}(\mathbf{r}_1)$ and $t_B(\mathbf{r}_2) = B(\mathbf{r}_2)$ hold by construction, and the lookup checks succeed because each value lies in $L_1^{(j)}$. Hence Step 2 accepts.

Step 3 (row-wise aggregation). For each i, define $Z_B^{(1)}(\langle i\rangle) = \prod_{j=0}^{k-1}\widehat{B}(\langle i\rangle,\langle j\rangle)$, where $\widehat{B}$ is the requantized (destination-scale) version of the per–digit exponentiation outputs (implicit in Step 2 lookups). By construction this equals a correctly rounded/scaled approximation of $\exp(|x_i|\cdot a_1/b_1)$ under scale a_2/b_2. The ProdCheck at a random $\mathbf{r}_3$ verifies the per–row product identity, so Step 3 passes.

Step 4 (reciprocal for negative inputs). For those i with $Z_A(\langle i\rangle) = 1$ (negative x_i), the prover runs Π_{Div} to compute $Z_B^{(2)}(\mathbf{X}) = 1/Z_B^{(1)}(\mathbf{X})$ (under the stated scales), with a remainder bounded as required by the division sub-protocol. By completeness of Π_{Div}, Step 4 passes.

Step 5 (final selection and requantization). Run Π_{Requan} (if needed) to align scales, obtaining $Z_B^{(3)}(\mathbf{X})$ at destination scale $s_2 = a_2/b_2$. Define the output polynomial $\widehat{B}(\mathbf{X}) = Z_A(\mathbf{X})\cdot Z_B^{(3)}(\mathbf{X}) + \big(1 - Z_A(\mathbf{X})\big)\cdot Z_B^{(1)}(\mathbf{X})$. For each i this equals $\exp(x_i\cdot a_1/b_1)$ represented at scale a_2/b_2, because for $x_i < 0$ we used the reciprocal and for $x_i \geq 0$ the direct product.

Knowledge Soundness. Let a prover be accepted with non-negligible probability. We build an expected polynomial-time extractor $\mathcal{E}$.

E1 (sign bits and $|x_i|$). From Π_{Cmp} in Step 1, $\mathcal{E}$ extracts $Z_A(\langle i\rangle) \in \{0,1\}$ for all i, and using commitment binding and the random-point equality, obtains $\widehat{A}(\langle i\rangle) = (2Z_A(\langle i\rangle) - 1)\,A(\langle i\rangle) = |x_i|$.

E2 (digits of $|x_i|$). By knowledge soundness of Π_{DigitDec}, $\mathcal{E}$ extracts $B(\langle i\rangle,\langle j\rangle) = |x_i|^{(j)} \in \{0,\ldots,2^d-1\}$ with $|x_i| = \sum_j 2^{jd}|x_i|^{(j)}$. The random-point checks and commitment binding tie t_B and B (and likewise t_C and $\widehat{C}$) as identical multilinear polynomials.

E3 (per–digit exponentiation values). From the lookup proofs against $L_1^{(j)}$, $\mathcal{E}$ extracts $\widehat{C}(\langle i\rangle,\langle j\rangle)$ that must equal the table value corresponding to $|x_i|^{(j)}$ at scale s_1 and destination scale s_2. Hence every per–digit exponentiation/quantization output is fixed.

E4 (row-wise product). ProdCheck soundness implies the identity $Z_B^{(1)}(\mathbf{X}) = \prod_{j=0}^{k-1}\widehat{B}(\mathbf{X},\langle j\rangle)$ holds as a polynomial (except with negligible probability). Thus $\mathcal{E}$ obtains the per–row aggregates $Z_B^{(1)}(\langle i\rangle)$ that are the (scaled) products of the per–digit factors.

E5 (reciprocal when $x_i < 0$). From Π_{Div}, $\mathcal{E}$ extracts $Z_B^{(2)}(\langle i\rangle)$ and the bounded remainder witnessing $Z_B^{(2)}(\langle i\rangle)\cdot Z_B^{(1)}(\langle i\rangle) = 1$ (at the declared scales). Hence for negative indices the reciprocal is fixed.

E6 (requantization and final selection). By knowledge soundness of Π_{Requan}, $\mathcal{E}$ extracts $Z_B^{(3)}(\langle i\rangle)$ at scale s_2. Finally, from the committed linear combination $\widehat{B}(\mathbf{X}) = Z_A(\mathbf{X})\cdot Z_B^{(3)}(\mathbf{X}) + \big(1 - Z_A(\mathbf{X})\big)\cdot Z_B^{(1)}(\mathbf{X})$, and commitment binding / random-point equality, $\mathcal{E}$ fixes $\widehat{B}(\langle i\rangle)$ for every i. By construction these values coincide with the destination-scale representation of $\exp(x_i\cdot a_1/b_1)$.

Combining E1–E6, $\mathcal{E}$ outputs a witness consisting of the sign bits Z_A, the digits of $|x_i|$, all per–digit lookup values, the row-wise product $Z_B^{(1)}$, the reciprocal $Z_B^{(2)}$ for negative inputs, the requantized product $Z_B^{(3)}$, and the final output $\widehat{B}$, all mutually consistent and satisfying the protocol's relations.

Theorem 7. *The protocol Π_{Div} as shown in Figure 8 satisfies completeness and knowledge soundness.*

Completeness. Assume the prover is honest and follows the protocol.

Step 1. Running Π_{Cmp} with threshold $(p+1)/2$ on $A(\mathbf{X})$ and $B(\mathbf{X})$ yields $Z_A(\mathbf{X})$ and $Z_B(\mathbf{X})$, which correctly record the signs of x_i and y_i, respectively. Then $\widehat{A}(\mathbf{X}) = (2Z_A(\mathbf{X}) - 1)\cdot A(\mathbf{X})$ and $\widehat{B}(\mathbf{X}) = (2Z_B(\mathbf{X}) - 1)\cdot B(\mathbf{X})$ equal $|x_i|$ and $|y_i|$ on the hypercube.

Step 2. The requantization protocols Π_{Requan} on $\widehat{A}$ and $\widehat{B}$ align scales so that $|x_i| \geq |y_i|$ and division does not produce a trivial zero. The difference $t_1(\mathbf{X}) = \widehat{A}(\mathbf{X}) - \widehat{B}(\mathbf{X})$ is nonnegative, and Π_{Cmp} together with Π_{CheckOne} verify this property correctly. Thus Step 2 passes.

Step 3. The prover computes the quotient $C(\mathbf{X})$ and remainder $r(\mathbf{X})$ satisfying $C(\mathbf{X})\cdot\widehat{B}(\mathbf{X}) + r(\mathbf{X}) = \widehat{A}(\mathbf{X})$. The difference $t_2(\mathbf{X}) = \widehat{B}(\mathbf{X}) - r(\mathbf{X})$ is nonnegative, and Π_{Cmp} and Π_{CheckOne} confirm that $0 \leq r(\langle i\rangle) < \widehat{B}(\langle i\rangle)$ for all i. Hence Step 3 is satisfied.

Step 4. The sign of each $z_i = x_i/y_i$ is determined by the signs of x_i and y_i, and Z_A and Z_B guarantee correctness. The final polynomial $\widehat{C}(\mathbf{X}) = (2Z_A(\mathbf{X})-1)\cdot(2Z_B(\mathbf{X})-1)\cdot C(\mathbf{X})$ therefore encodes the signed quotient at the output scale $s_3 = a_3/b_3$.

Knowledge Soundness. Assume Π_{Cmp}, Π_{Requan}, and Π_{CheckOne} are knowledge-sound, and that commitments are binding and allow extraction.

From Step 1, by knowledge soundness of Π_{Cmp}, the extractor obtains $Z_A(\mathbf{X}), Z_B(\mathbf{X})$ that correctly encode the signs of x_i, y_i, and hence $\widehat{A}(\langle i\rangle) = |x_i|$ and $\widehat{B}(\langle i\rangle) = |y_i|$.

From Step 2, by knowledge soundness of Π_{Requan}, the extractor learns the aligned versions of $\widehat{A}$ and $\widehat{B}$ under consistent scales. The proof of $t_1(\mathbf{X}) \geq 0$ via Π_{Cmp} and Π_{CheckOne} ensures these adjusted values satisfy the nontriviality condition.

From Step 3, extraction of $C(\mathbf{X})$ and $r(\mathbf{X})$ is possible from the commitment. The polynomial identity $C(\mathbf{X})\cdot\widehat{B}(\mathbf{X}) + r(\mathbf{X}) = \widehat{A}(\mathbf{X})$ is enforced by random-point and sumcheck arguments. The check $t_2(\mathbf{X}) = \widehat{B}(\mathbf{X}) - r(\mathbf{X}) \geq 0$ proved via Π_{Cmp} and Π_{CheckOne} guarantees the bound $0 \leq r(\langle i\rangle) < \widehat{B}(\langle i\rangle)$.

Finally, Step 4 extracts $\widehat{C}(\mathbf{X})$, which is defined deterministically from $C(\mathbf{X})$ and the sign bits Z_A, Z_B. Thus $\widehat{C}(\langle i\rangle) = x_i/y_i$ under the destination scale s_3.

Theorem 8. *The protocol Π_{Add} as shown in Figure 9 satisfies completeness and knowledge soundness.*

Completeness. Assume the prover is honest. In Step 1 the prover runs Π_{Requan} twice and obtains $\widehat{A}(\mathbf{X})$ and $\widehat{B}(\mathbf{X})$ such that, for every $i \in \{0,\dots,n-1\}$, $\widehat{A}(\langle i\rangle) = x_i$ at $s_3 = \frac{a_3}{b_3}$, $\widehat{B}(\langle i\rangle) = y_i$ at s_3. By completeness of Π_{Requan} these identities hold on the entire hypercube. In Step 2 the prover defines $C(\mathbf{X}) = \widehat{A}(\mathbf{X}) + \widehat{B}(\mathbf{X})$, hence for every i, $C(\langle i\rangle) = \widehat{A}(\langle i\rangle) + \widehat{B}(\langle i\rangle) = x_i + y_i$ at s_3.

Knowledge Soundness. Assume Π_{Requan} is knowledge-sound and the commitment scheme is binding with extractable openings; also assume the standard random-point equality test for multilinear polynomials (Schwartz–Zippel soundness).

Let a prover be accepted with non-negligible probability. From Step 1, by knowledge soundness of Π_{Requan}, an extractor $\mathcal{E}$ recovers polynomials $\widehat{A}(\mathbf{X})$ and $\widehat{B}(\mathbf{X})$ such that for all i, $\widehat{A}(\langle i\rangle)$ (resp. $\widehat{B}(\langle i\rangle)$) is x_i (resp. y_i) represented at scale s_3. By commitment binding and the random-point equality test applied to the final commitment $[C]$, $\mathcal{E}$ also fixes $C(\mathbf{X})$ and, except with negligible probability, obtains the polynomial identity $C(\mathbf{X}) \equiv \widehat{A}(\mathbf{X}) + \widehat{B}(\mathbf{X})$. Evaluating at $\mathbf{X} = \langle i\rangle$ yields $C(\langle i\rangle) = \widehat{A}(\langle i\rangle) + \widehat{B}(\langle i\rangle) = x_i + y_i$ under scale s_3 for every i.

Hence from any accepting transcript $\mathcal{E}$ extracts a valid witness consisting of $\widehat{A}$, $\widehat{B}$, and C that satisfy the required relation. Therefore, Π_{Add} is knowledge-sound.

D. Security Proof of LLM Functions

Theorem 9. *The protocol Π_{GELU} as shown in Figure 10 satisfies completeness and knowledge soundness.*

Completeness. *Step 1.* For each i, the prover computes $t_A(\mathbf{X}) = A^3(\mathbf{X})$, then requantizes t_A to scale $s_{t_A} = \frac{a_t}{b_t} = \frac{a_1^3}{b_1^3}$. By definition of q_1, q_2, the linear combination $t(\mathbf{X}) = q_1 \cdot A(\mathbf{X}) + q_2 \cdot t_A(\mathbf{X})$ correctly represents $t = \sqrt{2/\pi}\,(x + 0.044715 \cdot x^3)$ at scale $s_t = \frac{a_t}{b_t}$. Thus Step 1 is satisfied.

Step 2. The prover then computes $\tanh(t)$ under scale s_t. This involves invoking Π_{Exp} to compute e^{2t}, followed by two requantizations and one division to obtain $t_3(\mathbf{X}) = \frac{e^{2t(\mathbf{X})}-1}{e^{2t(\mathbf{X})}+1}$, all under scale s_{t_3}. By completeness of Π_{Exp}, Π_{Requan}, and Π_{Div}, Step 2 succeeds.

Step 3. Finally, the prover computes $B(\mathbf{X}) = \frac{1}{2} \cdot A(\mathbf{X}) \cdot (1 + t_3(\mathbf{X}))$, requantized to scale $s_2 = \frac{a_2}{b_2}$. Thus, for each i, $B(\langle i\rangle) = \frac{1}{2} \cdot x_i \cdot \Big(1 + \tanh(\sqrt{2/\pi}(x_i + 0.044715 x_i^3))\Big)$, which equals $\mathrm{GELU}(x_i)$ under s_2.

Knowledge Soundness. Assume $\Pi_{\mathsf{Requan}}, \Pi_{\mathsf{Exp}}, \Pi_{\mathsf{Div}}$ are knowledge-sound, and that all commitments are binding and extractable. Suppose a prover convinces the verifier with non-negligible probability.

From Step 1, the extractor obtains $A(\mathbf{X})$, $t_A(\mathbf{X})$, and $t(\mathbf{X})$, all consistent with A and its cubic term. Thus the extractor reconstructs $\sqrt{2/\pi}(x + 0.044715x^3)$ at scale s_t.

From Step 2, the extractor obtains $t_3(\mathbf{X})$, which is certified by $\Pi_{\mathsf{Exp}}, \Pi_{\mathsf{Requan}}, \Pi_{\mathsf{Div}}$ to equal $\tanh(t(\mathbf{X}))$ at the required scale.

From Step 3, binding and random-point equality checks guarantee that $B(\mathbf{X}) = \frac{1}{2} \cdot A(\mathbf{X}) \cdot (1 + t_3(\mathbf{X}))$. Thus the extractor recovers $B(\mathbf{X})$ which correctly encodes $\mathrm{GELU}(x)$ at scale s_2.

Therefore, from any accepting transcript, the extractor outputs a valid witness (A, B, t_A, t, t_3) satisfying all protocol relations.

Theorem 10. *The protocol Π_{Softmax} as shown in Figure 11 satisfies completeness and knowledge soundness.*

Completeness. *Step 1 (maximum computation).* The prover defines $t_A(\mathbf{X}) = x_{\max} - A(\mathbf{X})$. By construction, $t_A(\langle i\rangle) = x_{\max} - x_i \geq 0$ for all i, and there exists at least one i such that $t_A(\langle i\rangle) = 0$. The Π_{Cmp} and Π_{CheckOne} subprotocols confirm these conditions, while ProdCheck ensures at least one index achieves the maximum.

Step 2 (exponentials). The prover computes exponentials under different scales: $B_1(\mathbf{X})$ encodes e^{x_i} and $B_2(\mathbf{X})$ encodes $e^{x_{\max}-x_i}$. By completeness of Π_{Exp}, both computations are correct.

Step 3 (normalization). The prover computes the denominator $c = \sum_{i=0}^{n-1} e^{x_{\max}-x_i}$, via sumcheck on $B_2(\mathbf{X})$. The requantization ensures $t_{B_1}(\mathbf{X}) > B_2(\mathbf{X})$, so the denominator is positive. Finally, Π_{Div} computes $B(\langle i\rangle) = \frac{e^{x_i}}{\sum_{j=0}^{n-1} e^{x_j}}$, at the destination scale.

Knowledge Soundness. $\Pi_{\mathsf{Cmp}}, \Pi_{\mathsf{CheckOne}}, \Pi_{\mathsf{Exp}}, \Pi_{\mathsf{Div}}, \Pi_{\mathsf{Requan}}$, Product Check, and SumCheck are knowledge-sound, and that commitments are binding and extractable. Suppose a prover convinces the verifier with non-negligible probability.

From Step 1, by soundness of Π_{Cmp} and Π_{CheckOne}, the extractor obtains $t_A(\mathbf{X}) = x_{\max} - A(\mathbf{X})$ with $t_A(\langle i\rangle) \geq 0$ for all i, and ProdCheck certifies at least one i with $t_A(\langle i\rangle) = 0$. Thus the extractor recovers $x_{\max} = \max_i x_i$.

From Step 2, the extractor obtains $B_1(\mathbf{X})$ and $B_2(\mathbf{X})$, certified by Π_{Exp} as exponentials e^{x_i} and $e^{x_{\max}-x_i}$.

From Step 3, sumcheck guarantees $c = \sum_i B_2(\langle i\rangle)$, and requantization ensures $t_{B_1}(\mathbf{X}) > B_2(\mathbf{X})$, preventing division by zero. Π_{Div} then certifies $B(\langle i\rangle) = \frac{B_1(\langle i\rangle)}{\sum_j B_2(\langle j\rangle)}$. Thus the extractor recovers $B(\mathbf{X})$ encoding the normalized probabilities, which equal the softmax outputs.

Theorem 11. *The protocol Π_{Norm} as shown in Figure 12 satisfies completeness and knowledge soundness.*

Completeness. *Step 1 (mean).* The prover computes the mean $\mu = \frac{1}{n}\sum_{i=0}^{n-1} x_i$ represented as $q_\mu(\mathbf{X})$ at scale $s_\mu = \frac{a_1}{b_1}$, so $q_\mu(\langle i\rangle) = \mu$ for all i. This is correct by completeness of Sumcheck.

Step 2 (variance). Define $\widehat{A}(\mathbf{X}) = A(\mathbf{X}) - q_\mu(\mathbf{X})$. Run Π_{Cmp} and Π_{CheckOne} to check validity of the intermediate values. Then apply Sumcheck to compute $q_\sigma(\mathbf{X}) = \frac{1}{n}\sum_{i=0}^{n-1}(x_i - \mu)^2$ at scale $s_\sigma = \frac{a_1^2}{b_1^2}$, so $q_\sigma(\langle i\rangle) = \sigma$. This step succeeds by completeness.

Step 3 (normalization and affine map). The prover computes $t_{q_\sigma} = \sqrt{q_\sigma}$ at scale $s_{t_\sigma} = \frac{a_1}{b_1}$. By requantization and comparison checks, division by zero is avoided. Next, $t_1(\mathbf{X})$ and $t_2(\mathbf{X})$ are computed such that $t_1(\langle i\rangle) = (x_i - \mu)/\sqrt{\sigma}$ and $t_2(\langle i\rangle) = \gamma_i \cdot t_1(\langle i\rangle)$. Finally, the bias is added to form $B(\mathbf{X}) = t_2(\mathbf{X}) + \beta(\mathbf{X})$, hence $B(\langle i\rangle) = \gamma_i \cdot (x_i - \mu)/\sqrt{\sigma} + \beta_i$ at scale s_2.

Function	Scheme	256 × 256			512 × 512			1024 × 1024		
		Prover (s)	Verify (ms)	Proof	Prover (s)	Verify (ms)	Proof	Prover (s)	Verify (ms)	Proof
Cmp	Ours (KZG-BN254)	1.008	1.355	1.94 KB	4.121	1.694	1.94 KB	15.635	1.772	1.94 KB
	Ours (IPA-BN254)	12.684	467.864	3.53 KB	57.529	2080.193	4.01 KB	134.161	6724.695	4.41 KB
	Ours (FRI-M61-2^{64})	0.332	0.087	32.38 KB	0.753	0.162	39.42 KB	2.439	0.591	58.50 KB
	Hao et al. [3] (p≈M61-2^{64})	0.059	56.948	21.87 MB	0.384	327.794	87.48 MB	1.139	1011.150	349.90 MB
DigitDec	Ours (KZG-BN254)	0.530	0.967	1.92 KB	1.951	1.173	1.92 KB	10.572	1.386	1.92 KB
	Ours (IPA-BN254)	11.490	376.187	3.11 KB	41.305	1476.539	3.47 KB	106.040	7813.379	3.89 KB
	Ours (FRI-M61-2^{64})	0.092	0.059	29.10 KB	0.362	0.071	30.42 KB	1.035	0.098	40.97 KB
	Hao et al. [3] (p≈M61-2^{64})	0.043	51.180	17.65 MB	0.296	252.960	67.51 MB	0.798	708.140	245.03 MB
Requant	Ours (KZG-BN254)	2.026	1.504	2.62 KB	6.652	1.995	2.62 KB	19.140	2.308	2.62 KB
	Ours (IPA-BN254)	14.011	568.199	4.95 KB	63.647	2389.116	5.37 KB	254.829	8607.959	6.01 KB
	Ours (FRI-M61-2^{64})	0.452	0.102	41.87 KB	0.857	0.239	45.10 KB	2.935	0.351	61.25 KB

TABLE VII: Comparison of OpenLLM basic building blocks (Cmp, DigitDec, and Requant) in terms of prover time, verification time, and proof size.

Function	Scheme	256 × 256			512 × 512			1024 × 1024		
		Prover (s)	Verify (ms)	Proof	Prover (s)	Verify (ms)	Proof	Prover (s)	Verify (ms)	Proof
Add	Ours (KZG-BN254)	2.326	1.677	2.16 KB	7.591	2.037	2.16 KB	28.210	2.408	2.16 KB
	Ours (IPA-BN254)	15.259	638.168	5.62 KB	66.052	2563.847	6.24 KB	276.033	9808.816	6.73 KB
	Ours (FRI-M61-2^{64})	0.494	0.162	47.38 KB	1.144	0.368	53.42 KB	3.866	0.471	63.42 KB
Div	Ours (KZG-BN254)	2.704	3.381	2.94 KB	8.024	4.976	2.94 KB	31.652	14.942	2.94 KB
	Ours (IPA-BN254)	16.127	670.999	6.53 KB	74.254	2772.331	6.97 KB	281.960	10821.032	7.41 KB
	Ours (FRI-M61-2^{64})	0.559	0.226	52.38 KB	1.612	0.813	56.42 KB	4.079	1.794	60.47 KB
	Hao et al. [3] (p≈M61-2^{64})	0.406	414.381	153.92 MB	1.475	1502.940	614.81 MB	3.852	3691.106	2472.74 MB
Sqrt	Ours (KZG-BN254)	3.771	2.896	3.16 KB	7.903	4.702	3.16 KB	30.494	10.737	3.16 KB
	Ours (IPA-BN254)	16.860	634.297	7.09 KB	78.598	2650.736	7.55 KB	291.530	9921.639	7.89 KB
	Ours (FRI-M61-2^{64})	0.528	0.187	54.82 KB	1.822	0.513	66.71 KB	4.516	0.917	73.06 KB
	Hao et al. [3] (p≈M61-2^{64})	0.509	512.146	137.40 MB	2.436	2014.102	589.7 MB	4.975	5302.196	2052.19 MB
Exp	Ours (KZG-BN254)	3.903	4.927	4.12 KB	9.138	6.055	4.12 KB	34.005	18.013	4.12 KB
	Ours (IPA-BN254)	17.061	771.637	7.78 KB	82.166	2959.412	8.03 KB	308.724	11884.751	8.93 KB
	Ours (FRI-M61-2^{64})	0.847	0.401	59.50 KB	1.958	1.060	72.56 KB	5.372	3.147	89.20 KB
	Hao et al. [3] (p≈M61-2^{64})	0.570	593.106	92.17 MB	1.879	1963.941	475.3 MB	7.514	7902.894	1812.55 MB

TABLE VIII: Comparison of OpenLLM mathematical functions (Add, Div, Sqrt, and Exp) in terms of prover time, verification time, and proof size.

Knowledge Soundness. Assume the sumcheck protocol, Π_{Cmp}, Π_{CheckOne}, and Π_{Requan} are knowledge-sound, and commitments are binding with extractable openings.

From Step 1, the extractor obtains $q_\mu(\mathbf{X})$ such that $q_\mu(\langle i \rangle) = \mu$ for all i. From Step 2, the extractor reconstructs $A(\mathbf{X}) = A(\mathbf{X}) - q_\mu(\mathbf{X})$ and extracts $q_\sigma(\mathbf{X})$, which encodes the variance.

From Step 3, the extractor obtains $t_{q_\sigma}(\mathbf{X})$ as $\sqrt{\sigma}$, and then $t_1(\mathbf{X})$ and $t_2(\mathbf{X})$ with the relations $t_1(\langle i \rangle) = (x_i - \mu)/\sqrt{\sigma}$ and $t_2(\langle i \rangle) = \gamma_i \cdot t_1(\langle i \rangle)$. Finally, it extracts $B(\mathbf{X}) = t_2(\mathbf{X}) + \beta(\mathbf{X})$, so $B(\langle i \rangle) = \gamma_i \cdot (x_i - \mu)/\sqrt{\sigma} + \beta_i$.

E. Experiments of Basic Operators

Results of building blocks. Table VII reports the performance of core building blocks, including Cmp, DigitDec, and Requant. The results show that the FRI-based construction consistently achieves the lowest prover time and verification latency across all input scales. For example, on Cmp with input size 1024×1024, the prover time is 2.439s, compared with 15.635s for KZG and 134.161s for IPA. In contrast, the baseline [3] incurs significantly higher verification cost and extremely large proof sizes. KZG- and IPA-based constructions provide smaller proofs but exhibit higher computational overhead, demonstrating a clear trade-off between efficiency and proof size.

Results of mathematical functions. Table VIII presents the performance of fundamental mathematical functions, including Add, Div, Sqrt, and Exp. Similar trends are observed across all operators: the FRI-based construction achieves the best performance in terms of prover time and verification latency, while KZG- and IPA-based constructions offer more compact proofs. For instance, for Exp at input size 1024×1024, the prover time is 5.372s, compared with 34.005s for KZG and 308.724s for IPA, while maintaining low verification cost. The baseline [3] again exhibits substantially larger proofs (up to GB level) and higher verification overhead. These results demonstrate that our operator-level design provides efficient and scalable building blocks for higher-level LLM computations.