DeepProve: Verifiable End-to-End Large Language Model Inference

Nicolas Gailly[1], Ismael Hishon-Rezaizadeh[1], Tianyi Liu[1,2,†], Nicholas Mainardi[1,†], Dimitrios Papadopoulos[3], Charalampos Papamanthou[1,4], Christodoulos Pappas[3], Shravan Srinivasan[1,†], Zack Youell[1,†], and Yupeng Zhang[1,2*†]

[1]Lagrange Labs
[2]University of Illinois Urbana-Champaign
[3]Hong Kong University of Science & Technology
[4]Yale University

Abstract

Large Language Models (LLMs) are frontier deep learning systems that have achieved remarkable success across a wide range of AI services. However, their substantial computational and memory requirements make them difficult to deploy and run on local hardware. Due to these resource requirements, users often rely on untrusted cloud infrastructure providers to perform model inference. However, outsourcing introduces the challenge of verifying that the returned output is the genuine result of the specified model. In this work, we present `DeepProve`, the *first* system to enable efficient *end-to-end* verification of full LLM inference (i.e., for all generated tokens of a prompt) on untrusted cloud servers using zero-knowledge proofs (ZKPs). In contrast, prior work either provides only a proof-of-concept partial implementation for a single token (zkGPT, USENIX'25), or focuses exclusively on specific components of the inference pipeline, such as Softmax (zkLLM, CCS'24).

`DeepProve` achieves end-to-end verification by *certifying* the correctness of the output sequence rather than encoding the expensive inference computation in-circuit, an approach that would require either circuit size quadratic in the sequence length or costly in-circuit modelling of RAM operations. The core building blocks of `DeepProve` are sum-check protocol and lookup arguments, which enable efficient proof of correctness of *all* operators needed for GPT-2 and Gemma 3, such as multi-head attention and layer normalization for GPT-2, and grouped-query attention, root mean square normalization, and rotary positional embeddings for Gemma 3. Our evaluation shows that `DeepProve` can prove inference of GPT-2 and Gemma 3 at approximately 174 and 86 tokens per minute, respectively, which is $20-60\times$ faster than the state of the art, without *any* significant loss in accuracy. Verification takes only 1 to 3.7 seconds. By distributing proof computation across multiple nodes, `DeepProve` can further improve the prover time while reducing the memory requirements for individual machines. With distributed proving, `DeepProve` can scale the throughput to 1855 tokens per minute. Our work represents the first full system for end-to-end LLM inference verification, thus paving the way for secure and trustworthy AI services.

1 Introduction

Large Language Models (LLMs) are powerful deep learning models that are based on the fabled transformer architecture [VSP+17], predicting the next word in a sequence to generate human-like text and answer queries by learning patterns from vast amounts of training data. They have found countless real-world applications including chatbots [Ope25], programming assistance [ea21], healthcare [MS25], search engines, defense [Loc24], and many more [MMN+24]. This wide applicability across numerous sectors of critical importance raises the question of preserving the *integrity* of the result of LLM inference, as it can potentially affect crucial decision-making steps.

For example, consider the following AI-based services:

- **Premium enterprise support:** Enterprises often pay a premium to access advanced LLM capabilities for customer support bots. However, AI providers could return responses from weaker models to cut costs, degrading the promised service quality [QSL+25, Sal23, Com24, Com23].
- **Programming assistants:** These systems handle operations such as writing code, editing files, and executing command-line instructions [Git26], thus making it essential to ensure that the actions originate from the trusted intended LLM rather than a malicious actor, as malicious code can be unwittingly executed by the integrated development environment (IDE) on the user's machine.
- **Government procurement and services:** Sovereign states (e.g., see [Wik25a]) are exploring deploying AI systems with the intention to reduce corruption in procurements and enhance transparency in public services. In this case, ensuring that decisions and actions originate from the authorized AI system, rather than from a potentially malicious actor, is essential.

At the same time, LLM inference is highly resource-intensive, requiring substantial computational and memory resources [YJK+22]. As a result, AI service providers may themselves wish to outsource the task of inference to external service providers. In this case, again, one needs to ensure the correctness of returned results, as the service providers may be motivated to alter them.

The above has strongly motivated the need to *verify* deep learning inference results in the literature (e.g., [LXZ21, SLZ24, QSL+25, CTK25]), i.e., to ensure that the returned inference outcome is the result of faithfully executing the model without deviating from the computation.

Succinct non-interactive arguments of knowledge (SNARKs) [GGPR13, BSCTV14, GKR08, CMT12] are powerful cryptographic primitives that allow a prover to convince a verifier about the correctness of a computation by producing a small and easy-to-verify proof. This is usually done by representing the computation as an arithmetic

*Authors are listed alphabetically.
†Work done while affiliated with Lagrange Labs.

circuit whose correct evaluation can be reduced to checking a set of numerical constraints. There has been a vast line of works that apply SNARKs to protect the integrity of ML tasks such as training and inference of multilayer perceptron (MLP), convolutional neural network (CNN), decision trees, and other deep learning models (e.g., [APKP24, GGJ+23, BFGV+23, LKKO24, LXZ21, LWQ+24, KHSS22, CCYY25, MY25, ZFZS20, PP24, zko24, TGM+25, ea25a]).

A smaller number of recent works [SLZ24, QSL+25, CTK25] have specifically focused on building SNARK-based protocols for verifying LLM inference. Unfortunately, while these prior works have made considerable progress towards this goal, they all suffer from some crucial limitations. First, they only focus on proving the correctness of a single-token inference and do not explain how to extend to the realistic scenario of multiple tokens. As we discuss next, this is not trivial and may lead to prohibitive prover overheads. Furthermore, existing works either focus on proposing efficient provers only for a specific part of LLMs (e.g., zkLLM [SLZ24] proposes a protocol for efficiently proving a Softmax layer, while zkGPT [QSL+25] a protocol for efficiently proving matrix multiplications) and omit details on other layers, or construct an end-to-end system using black-box proving systems thus becoming concretely inefficient and massively memory-consuming (e.g., the experimental evaluation of [CTK25] reports results for a machine with 4 TB RAM). Finally, the most efficient schemes to date [SLZ24, QSL+25] only offer proof-of-concept implementations which omit specific parts of the prover (see Section 6 for a detailed discussion).

While their results are a necessary step towards our goal, focusing on individual building blocks or parts of the inference process, hides several key challenges that arise when trying to build an *end-to-end* verifiable system for the entire LLM inference process in real-world LLMs–addressing these and building an efficient end-to-end system will be the focus of this work.

1.1 Challenges in End-to-End Proving

Here we discuss some of the key challenges that need to be addressed to successfully build a full-fledged system for verifiable LLM inference, related to scalability, efficiency, and accuracy.

Quadratic blowup from vanilla inference. A key part of LLM inference is the self-attention mechanism [VSP+17], which recomputes the attention scores over all previous tokens in the sequence, when generating each new token, enabling the model to dynamically focus on the most relevant parts of the input. Computed "naively" this would require time quadratic in the length of the input sequence. Existing LLM systems overcome this quadratic blowup by trading computation for memory. This approach is known as Key-Value (KV) caching [Wik25c]. In KV-caching, the inference algorithm stores the large tensors called keys and values in memory and reuses them for generating future tokens without recomputing the attention scores over all previous tokens.

When it comes to proving LLM inference, following the naive approach above for a sequence of length t and assuming a specialized circuit for proving the correctness of a single token is of size $|C|$, would lead to prover time $O(t^2|C|)$. This is prohibitively large for long token sequences, thus limiting the scalability of any proof system based on this approach. We stress that prior works in verifiable LLM inference do not address this problem as they focus on proving only parts of the inference process (e.g., single-token inference [QSL+25]).

To overcome that issue, one could try to emulate KV-caching [Wik25c] techniques within the SNARK. While this may lead to asymptotically faster provers, it is still an undesirable solution. This is because, to realize KV-caching, the prover has to emulate read/write RAM operations within the SNARK circuit. This complicates the prover's logic and leads to large concrete overheads even when using state-of-the-art protocols for emulating RAM (e.g., via lookup arguments) as they require the prover to commit to additional auxiliary data that scales linearly to the number of reads/writes.

Need for pure-integer inference. Most existing SNARK constructions operate over finite fields. In contrast, LLM computations are defined over real-valued numbers and implemented using floating-point arithmetic. Bridging the gap is non-trivial and can introduce significant inefficiencies and/or potential accuracy loss. This is a well-known issue in SNARK literature and in particular, when proving ML tasks [HCL+24, SSPP25, RWB+25]. Prior works show that emulating floating-point arithmetic inside a SNARK system remains quite impractical [GJJZ22, EZC+25], hence most existing works [QSL+25, CTK25, SLZ24, zko24, HCL+24, RWB+25] adopt fixed-point arithmetic via quantization over a given domain (e.g., 8 bits). However, certain inputs might not be able to be represented correctly as fixed point if they're too small, or on the opposite, too large inputs might blow up the values. Given lookup table range needs to be fixed in advance, some inputs might not be provable at all. Interestingly enough, independently of cryptographic integrity, deep learning practitioners already quantize model weights for reduced memory footprint and faster inference, but typically keep activations in floating-point format during the inference [SSZ21] to mitigate lossiness due to quantization and accuracy degradation. However, to cryptographically prove the correctness of inference, both weights and activations must be in integers as well as all operations between them. Therefore, an *accurate* integer-only inference algorithm is necessary for efficient proof generation.

Outlier activations. Outlier activations in LLMs are unusually large magnitude values that occur in the activation tensors of certain layers during inference [RSZ25]. These extreme values, often concentrated in specific outlier feature channels, or caused by reasons such as layer normalization and residual connections [AMC+24, LXW+24, vBBWN25] and can cause significant quantization errors. This poses an even more significant challenge when performing pure-integer inference as outlier activations can cause the quantization bin to be too wide, leading to typical activations getting rounded to the same quantized value. For instance, consider an activation $[9.1, 3.9, 11.5, 1000.0]$. If we quantize this activation to 8-bits, all typical activations will get quantized to either 0 or 1, leading to total accuracy loss. Thus, any pure-integer inference algorithm must address the outlier activation problem to preserve accuracy.

Requantization strategy. Modern prover-efficient SNARKs [HLP24, Tea, PP25] can benefit from being able to operate directly over small finite fields, closer to standard CPU word sizes, (e.g., 64-bit Goldilocks field [Tea]). While this allows for faster prover times, in our verifiable LLM inference application it introduces an additional challenge. During quantized integer inference we need to perform *requantization* after every few operations (mainly multiplications)

to prevent overflow in the finite field, i.e., to bring the values back into the quantization range. However, requantization itself is a lossy operation and can further degrade accuracy.

Batch proving of operations. Proving linear operations such as matrix multiplications are efficiently handled by SNARK systems. However, non-linear operations such as Softmax, LayerNorm, and GeLU that are needed in LLMs, are "SNARK-unfriendly" and require specialized protocols to achieve the best efficiency and accuracy [SLZ24, SSPP25]. Moreover, these non-linear operations are used extensively and multiple times during LLM inference (as opposed to simpler ML tasks such as neural networks). As well, a full trace of inference requires to commit and open many polynomials over different points. Last but not least, LLM inference is autoregressive, e.g. it runs *multiple* times a forward pass to produce a vector of trace. Naively proving each trace separately would incur a prohibitive cost. Thus, to increase efficiency, a practical system must be designed to batch the proving of multiple instances of the non-linear layers, the PCS openings, and the multiple traces.

1.2 Our Contributions

In this work, we design and implement `DeepProve`, which, to the best of our knowledge, is the first system that enables efficient end-to-end verifiable LLM inference of transformer-based LLMs. Specifically, we fully support verifiable inference for: (i) GPT-2 (124 million parameters) [RWC+19], and (ii) Gemma 3 (270 million parameters) [Tea25]. Our key contributions are as follows:

(1) We build specialized efficient PIOPs for a variety of LLM layers, including (a) re-quantization—a fundamental operation in quantized inference, (b) normalization layers that do not require the prover to show the correct computation of square roots and inverses (dramatically improving prover time and enhancing accuracy), (c) multi-head layer attention, (d) rotary positional embeddings (see Section 4.3). This is in stark contrast with prior works that focus only on specific components of LLMs, such as proving Softmax [SLZ24] or efficient matrix-to-matrix multiplications [QSL+25], or only prove single layer attention [QSL+25]. We formally prove that each component is round-by-round (knowledge) sound [CCH+19], which enables us to argue about the soundness of our scheme when making it non-interactive via the Fiat-Shamir transformation.
(2) Using the above protocols as building blocks, we show how to plug them together in order to construct a PIOP for proving the correct inference of an entire LLM model. While this is relatively straightforward when proving the correctness of a single token [SLZ24, QSL+25], based on our prior discussion, it is unclear how to efficiently achieve this in realistic scenarios with more tokens. Instead, we propose a certification algorithm which is given the input and output tokens and validates the correctness of the latter by simply *re-executing the inference step only for the final token*. This directly reduces the prover overhead from quadratic in the number of tokens to linear (see Section 3.1). We then combine the resulting PIOP with a polynomial commitment scheme (PCS) to build `DeepProve`, the first end-to-end verifiable LLM inference system, and apply it to the GPT-2 and Gemma 3 models.
(3) We propose several practical optimizations to reduce prover overheads, e.g., delayed re-quantization, or clamping outlier activations to improve the efficiency without sacrificing accuracy.
(4) We implemented in Rust, open-sourced `DeepProve` [cod] and extensively evaluated performance (see Section 5). Its *peak* proving throughput is approximately 174 tokens per minute for GPT-2 and 86 tokens per minute for Gemma 3 (on a single machine), which is $20-60\times$ faster than the state-of-the-art [QSL+25]. Verification times are around 1.2 seconds for GPT-2 and 4 seconds for Gemma 3, while proofs are 7-27MB and 16-54MB for GPT-2 and Gemma 3, respectively, yielding reasonable deployment overheads for verifiers. Finally, we design and evaluate a distributed proving algorithm for `DeepProve` by horizontally scaling across multiple machines (see Section 4.5). In our experimental evaluation, we show that distributed `DeepProve` can scale the proving throughput to 1855 tokens per minute with 17 machines.
(5) In terms of accuracy, `DeepProve` achieves comparable scores (Perplexity and cosine similarity) with that of floating-point inference. For instance, our quantized integer inference algorithm with 12-bit quantization maintains accuracy within 0.54% of the floating-point baseline for GPT-2. Unlike prior works [SLZ24, QSL+25], our quantization approach is based on standard ML quantization techniques without any custom modifications.
(6) As an independent contribution, we include in the `DeepProve` pipeline a type of "proof-of-non-hallucination" (see Section 4.4). In more detail, we propose a protocol for the proof of the correct execution of a state-of-the-art hallucination detection technique, LLM-Check [SBS+24], extended to work on quantized inference with minimal accuracy loss. Thus, `DeepProve` convinces the verifier, not only about the correctness of the LLM inference but also that the produced result passed the hallucination detection algorithm. To our knowledge, this is the first approach to cryptographically prove LLM non-hallucination in the literature.

Overall, we believe our work takes a significant step towards more practical verifiable LLMs, since it not only is the first fully realized open-sourced system capable of producing proofs for the entire inference process, but our prover is also 20-60× faster than prior works. That said, verifiability still introduces a substantial overhead to the inference process. For comparison, plaintext GPT-2 inference on our experimental testbed, using PyTorch [SSZ21] and running on CPU, achieves 71.4 tokens-per-second (TPS) for prompts with 128 input and 128 output tokens (sequence length 256), compared to 3 TPS for our prover—-roughly 24× slowdown. For other queries with the same total sequence length but much smaller output token size, plaintext inference achieves more than 2K TPS, almost three orders of magnitude faster than our prover. Besides, this gap can grow even further in practice when plaintext inference leverages additional optimizations such as GPU execution or batching strategies.

Organization. The rest of the paper is organized as follows. In Section 2, we present some necessary preliminaries on LLMs and in Section 3, we present a technical highlights of our approach. Section 4 contains the core design of `DeepProve`, including our protocols for different layers and their end-to-end integration, as well as our approach to distribution. In Section 5, we present our evaluation results. Finally, in Section 6, we discuss related works.

2 Preliminaries

We include here some necessary background information for quantization and LLMs. Cryptographic definitions are in Appendix A.

2.1 Quantization

We use standard affine quantization techniques to convert the model weights and activations from floating-point to integer representations. The standard formula [JKC+18] to convert a floating-point value x into a quantized integer value $\overline{x}$ is $x = s \cdot (\overline{x} - z)$, where s is the scale factor and z is the zero-point.

Linear operations [JKC+18] and requantization. Consider the matrix operation $\mathbf{X}_4 = \mathbf{X}_1\mathbf{X}_2 + \mathbf{X}_3$, where $\mathbf{X}_1 \in \mathbb{R}^{m\times\ell}$, $\mathbf{X}_2 \in \mathbb{R}^{\ell\times n}$, and $\mathbf{X}_3, \mathbf{X}_4 \in \mathbb{R}^{m\times n}$ represented as floating-point values. We denote entries of each matrix as $x_\alpha[i,j]$ for $\alpha \in \{1,2,3,4\}$. The quantization relations for each entry in the matrices are given by: $x_\alpha[i,j] = s_\alpha(\overline{x}_\alpha[i,j] - z_\alpha)$. From the definition of matrix multiplication with bias addition: $s_4(\overline{x}_4[i,k] - z_4) = \sum_{j=1}^{\ell} s_1(\overline{x}_1[i,j] - z_1)s_2(\overline{x}_2[j,k] - z_2) + s_3(\overline{x}_3[i,k] - z_3)$, and $\overline{x}_4[i,k] = z_4 + M\sum_{j=1}^{\ell}(\overline{x}_1[i,j] - z_1)(\overline{x}_2[j,k] - z_2) + B(\overline{x}_3[k] - z_3)$, where $M := \frac{s_1 s_2}{s_4}$ and $B := \frac{s_3}{s_4}$. In practice, ML frameworks, like PyTorch [For25], set $s_3 = s_1 s_2$. Since, M is in $(0,1)$, we set $M = 2^{-n}\epsilon$ for some small $\epsilon > 0$ and integer n.

2.2 Large Language Models

While `DeepProve` is generic over models representable as a graph of operations (MLPs, CNNs, etc), particular attention have been put into making it efficient on LLMs which mainly consist of:

- an embedding layer, mapping the user one-dimensional input to the multi-dimensional space the LLM operates on
- multiple sequential *transformer* blocks reasoning about the inputs in the multi-dimensional space, each time changing the hidden state, e.g. the state of the transformed input
- at the end, an un-embedding layer that chooses the next token according to the hidden state.

Auto-regressive models: Users of an LLM first give the whole input to the LLM, generate the next token, and append this token to the input, and run the LLM again. This loop stops after a predetermined number of iterations or until the LLM indicates it has nothing else to produce. GPT-2 [RWC+19] is one of the first LLM models that were able to produce readable English. See Fig. 1 for a detailed view of a GPT-2 transformer block, and Fig. 4 for Gemma 3.

3 Technical Overview

In this section, we present an overview of the key techniques that enable `DeepProve` to achieve both high accuracy and high performance in verifiable LLM inference.

3.1 Certification of the LLM Inference

First, we propose a certification algorithm for verifying that the output tokens were correctly computed by the LLM model given the input tokens. Our certification algorithm for this is extremely simple: It just feeds the entire sequence of input and output tokens into the model at once, essentially performing a single inference step. In practice, this translates to proving only the last inference step leading to prover times that are only linear to the input and output tokens rather than quadratic, as in the vanilla inference discussed in the introduction section. Correctness stems from the way attention mechanism works, in that the model generates token predictions for every position in the sequence. If the model's prediction at position t matches the claimed token at position $t+1$, then the token at position $t+1$ is certified as correctly computed. Because of the Attention Mask, the model's calculation for the t-th token cannot use the values of the $(t+1)$-th token. Moreover, argmax in the final layer of GPT-2 (Fig. 1) and Gemma 3 (Fig. 4) provides determinism to the execution. Consider the following example:

- **Prompt:** "The cat"
- **Claimed output:** "sat on the mat"
- **Concatenated sequence:** [The, cat, sat, on, the, mat]
- **Predicted sequence:** [cat, sat, on, the, mat, $\perp$]

Thus, our certification algorithm simply checks if the model's prediction at each position matches the claimed token at the next position until the end of the sequence.

3.2 Accurate Integer Inference

The linear operators of LLMs are rather straightforward to implement using quantized integer arithmetic, as described in Section 2.1. However, non-linear operations such as Softmax, GeLU, LayerNorm, and RMSNorm are inherently challenging as basic (field) algebraic operations are not enough to evaluate the functions [Wik25b]. To get around this challenge, we use a combination of approximations and lookup tables [SLZ24] to implement these non-linear functions accurately using quantized integer arithmetic. However, these techniques alone are insufficient to obtain high accuracy in LLMs (e.g., the outlier activations problem [AMC+24, LXW+24, vBBWN25]). Here we highlight the techniques that we use in `DeepProve` to achieve efficient and high accuracy in integer LLM inference.

Mixed-precision inference. `DeepProve` can be parametrized by any quantization bit-length. We observe that certain layers benefit from keeping values in higher precision. For example, a residual layer simply adds its two inputs and return the result as output. However, the input to an attention block or feed forward network is normalised first so the value range on the residual and the block output can vary dramatically (See Figs. 1 and 4). To add these two quantized tensors, they must have the same scaling factor. In low precision, the process of rescaling the tensor with the smaller value range to have the scaling factor matching the one with larger value range incurs a substantial loss in accuracy. `DeepProve` deals with this issue by using a larger bit length for this layer, specifically such that the quantization grid is fine enough that values from both tensors can be captured.

Outlier smoothing. As noted in prior works ([AMC+24, LXW+24, vBBWN25]), the output of LLM layers can contain large outlier values. One of the main challenges in producing accurate quantization at smaller bit lengths is handling the quantization of these large outlier values. To reduce the size of these outlier values, `DeepProve` makes use of smoothing techniques discussed in [AMC+24, LXW+24, vBBWN25]. To smooth these values, we use *orthonormal matrices*. An *orthonormal matrix* $M \in \mathbb{R}^{n\times n}$, is a matrix satisfying the properties $M \cdot M^\top = \mathrm{Id}_n$ and $\det(M) = 1$. If we write $X \in \mathbb{R}^{k\times n}$ for the input to a linear layer and $W \in \mathbb{R}^{n\times l}$ for the weight matrix we can observe that $X \cdot W = (X \cdot M) \cdot (M^\top \cdot W)$, where $X \cdot M$ and $M^\top \cdot W$ are both much easier to quantize (see [CCKDS23] for more

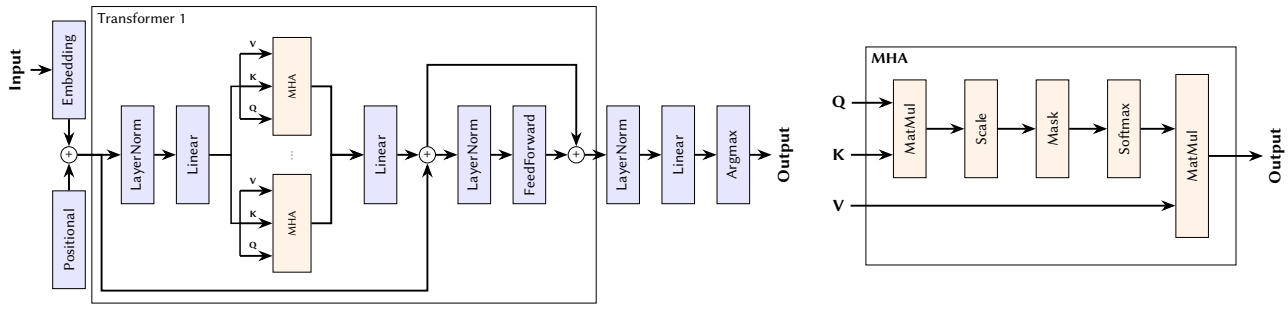

Figure 1: GPT-2 architecture. For simplicity, we illustrate only a single transformer block. The subfigure details the operations of the multi-head attention mechanism.

details). To avoid ever materializing the hard-to-quantize X, the right multiplication by M is absorbed into the previous linear layer. By this, we mean if $X := X' \cdot W'$ with $X' \in \mathbb{R}^{k \times b}$ and $W' \in \mathbb{R}^{b \times n}$ then we modify W' by multiplying on the right by M. This transformation has no effect on the floating-point model's output, but is substantially easier to quantize as the range of values observed as linear layer outputs is far smaller.

Requantization. The bit length of the product of n q-bit integers is at most nq bits, and addition of n q-bit integers produces at most a $q + \log n$ bit integer. Thus, without any intervention, the bit-length of the intermediate values in LLM inference would grow rapidly, leading to computation slowdown and even potential field overflow. To address this, `DeepProve` employs *re-quantization* after every few layers to reduce the bit-length of intermediate values back to a manageable size. `DeepProve` uses post-training static quantization [SSZ21] to perform re-quantization, which involves computing the optimal scaling factor and zero-point for intermediate values based on their distribution in floating-point inference.

3.3 Efficient Proving of LLM Inference

We design specialized protocols to efficiently prove the correctness of each operator used in transformer-based LLMs, such as GPT-2 and Gemma 3. Recall from Section 3.1 that we certify the correctness of the entire output sequence by performing inference on the concatenated input and output tokens. Since this generates multiple predictions in a single forward pass, we can leverage *batched* variants of our protocols to prove the correctness of multiple outputs.

The primary cryptographic tools we use to design our operator protocols are: Specialized sum-check protocols [LFKN90] for proving linear layers, and lookup arguments based on LogUp [PH23] for proving non-linear layers. Similar to the fabled GKR protocol [GKR08], we compose these operator protocols in a layer-by-layer manner to obtain an end-to-end proof of the entire LLM inference. Note that, in this way, our approach does not commit to intermediate values computed during the inference beyond model parameters and the witness values needed for lookup arguments in non-linear layers.

Finally, we deploy several optimizations in `DeepProve`, such as delayed-requantization. Recall that we must perform re-quantization after every few layers to shorten the bit-length of intermediate values. Since re-quantization involves scaling and rounding operations, we can merge the non-linear operations with the subsequent re-quantization step. This allows us to prove both operations together using a single lookup argument. In the literature [QSL+25], this approach is typically called operator fusion. We refer readers to Appendices C and B for other optimizations such as clamping, and to Section 4.4 for hallucination detection.

4 DeepProve

Here, we describe the overall system architecture of `DeepProve`, split into pre-processing and proving phases (Fig. 2).

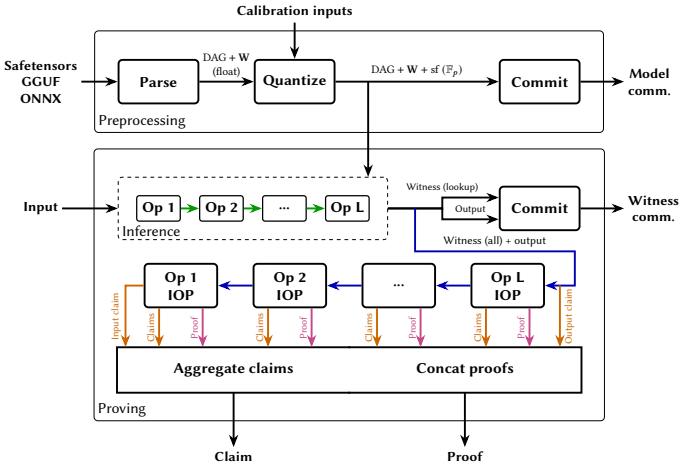

Figure 2: System Architecture of DeepProve

4.1 Pre-processing

The pre-processing phase in `DeepProve` is a *one-time* step for any specific ML model (see Fig. 2). It consists of three steps: (1) Parsing, (2) Quantization, and (3) Commitment to the model weights and operators graph. Here, we describe each of these steps in detail.

Parsing. In the pre-processing phase, we input the LLM model file in Safe Tensors [Fac25b], GGUF [Fac25a] or ONNX [dev19] format. The model is parsed to extract the operator graph and weights in floating-point precision. Thus at the end of parsing, we have a directed acyclic graph of operators where each node represents an operator (e.g., MatMul, Conv, etc.) and edges represent the flow of tensors between operators. The weights associated with each operator are also extracted and stored in the respective nodes. If the model is already quantized, e.g. GGUF, we dequantize the weights first using the provided scaling factors.

Quantization. This step of pre-processing takes a model with weights in floating-point representation (e.g., FP32) and first converts them to q-bit signed integer representation (e.g., in range $[-2^{q-1}+1; 2^{q-1}-1]$) using affine quantization (see Section 2.1).

Our system uses a *symmetric* quantization centered around zero for both weights and activations, agnostic to the choice of quantization bit length. Thus, we can easily adapt our system to use different bit lengths (e.g., 8-bit, 12-bit, or 16-bit) as needed. Specifically, we perform *post-training static quantization* to convert the model weights from floating-point to signed integer representation [SSZ21]. To quantize the model we need to compute the scaling factors for the weights and activations, for which we use a *calibration dataset* consisting of representative inputs to the model. We evaluate the model using the calibration dataset and collect the activation and weight statistics for each layer *as well as the input/output* of each layer. Using these statistics, we compute the scaling factors for activations, weights and input/output in each layer of the ML model as $s = \frac{2a}{2^b-1}$, where b is the bit width, x_{max} and x_{min} are the maximum and minimum activation (resp. weights) values observed during calibration. Since we use symmetric quantization, $a = \max(|x_{\max}|, |x_{\min}|)$. Whenever the activations are outside the calculated range during inference, we *clamp* them to be within the range $[-a, a]$. See Appendix C for more information about the statistics we take to obtain the most accurate scaling factors.

Say a signed integer value is $-x < 0$. To represent this value in a finite field $\mathbb{F}_p$, we compute its equivalent representation as $p - x$, where p is the order of the field. This way we convert all signed integer weights (and activations) to their corresponding finite field representations. Thus, at the end of this step, we have the ML model weights represented as finite field values, the per-layer quantization scaling factors, and the operator graph extracted during parsing.

Integer to Fields. During model commitment and proving (and *not* during inference), `DeepProve` maps the integer weights to the fields chosen for proving. Specifically, an integer $i \in \mathbb{Z}$ is mapped to a field $f \in \mathbb{F}$ in the following way. (a) If $i \in [0; 2^{q-1}-1]$ then f is mapped to the same value in the field ($|\mathbb{F}|$ is assumed to be much larger than the quantization bit-length, e.g. 64 or 256 etc.). (b) If $i \in [-2^{q-1}-1; 0]$, then f is mapped to $ord(\mathbb{F}) - 1 - i$, i.e., it is mapped to the *higher end* of the field range. Using this method, it can be shown that the arithmetic actually holds and produces the correct sign output.

Model Commitment. In this step, we commit to the model weights using a polynomial commitment scheme (PCS). Our system and protocols are agnostic to the choice of PCS.

Let W_i denote the weight tensor for layer i of the ML model and (m, n, ℓ) be the dimensions of the tensor. We represent W_i as a function $f_i : \{0,1\}^{\log m} \times \{0,1\}^{\log n} \times \{0,1\}^{\log \ell} \to \mathbb{F}$, where $f_i(\mathbf{a}, \mathbf{b}, \mathbf{c}) = W_i[a, b, c]$. Let the model have L layers. Each layer i contains a collection of parameter tensors $\{W_{i,1}, \ldots, W_{i,C_i}\}$ (for example, a dense layer has both a weight matrix and a bias vector). Each tensor $W_{i,j}$ is represented as a multilinear function $f_{i,j}$. The full model is therefore represented by the functions $\{f_{i,j} \mid 1 \le i \le L,\ 1 \le j \le C_i\}$. If the PCS doesn't support homomorphism, then we compute a single commitment to all model parameters using the batch commit operation of the PCS. However, if the PCS supports homomorphism, then we compute a commitment to each polynomial separately.

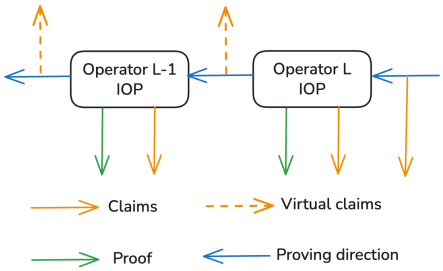

Figure 3: Operator proving in DeepProve

Thus, at the end of this step, we have commitments to the model weights and quantization scaling factors that will be used both by the prover and verifier.

4.2 Proving

The proving phase is executed for *each* input to the LLM. It consists of three main steps: (1) Quantized inference, (2) Witness commitment, and (3) Operator proving. In this subsection, we describe each of these steps in detail.

Quantized Inference. Given an input to the ML model from the user, as mentioned in Section 3.2, we perform quantized inference using the quantization scaling factors computed during the pre-processing phase. The inference is executed operator-by-operator. For each operator, we perform the necessary computations using integer arithmetic. This includes: (1) clamping activations to ensure they remain within the quantization range, (2) smoothing out outliers, (3) using mixed-precision arithmetic for residual layers, and (4) re-quantizing the outputs as needed if the output range grows too large and cannot be easily proven. The output of each operator serves as the input to the subsequent operator in the graph. During this process, we record the execution trace, which includes all intermediate activations and outputs produced at each operator node in the graph. We also record any additional values needed for the prover, such as the shift values applied before the Softmax operation, as in [SLZ24]. This execution trace forms the *witness* for the proof generation step.

Witness Commitment. During the pre-processing phase, we committed to the static information such as model weights and quantization scaling factors. In this step, we commit to *some* non-deterministic information. For example, we use lookup arguments to verify non-linear computations, specifically employing Logup GKR as the lookup argument [PH23]. Logup GKR requires committing to the lookup table and the witness values as a part of the statement. Thus, in this step, we commit to these witness values that depend on the input to the model. We employ a similar approach to model commitments and use the batched polynomial commitment scheme to commit only to the witness values that explicitly need to be committed and cannot be *virtualized* [Tha25].

4.3 Operator Proving

Suppose the operator graph has L layers. The proving process is carried out layer-by-layer, similar to the GKR protocol [GKR08, CMT12], proceeding from the output layer to the input layer of the ML model, e.g., in *backwards* order.

In the first step of the proving phase, the prover makes a claim about the output of the ML model, i.e., the output of operator L. This claim is actually derived from the Fiat-Shamir transformation after all commitments and auxiliary information have been correctly integrated into the transcript. Since the verifier cannot evaluate the model directly, the prover and verifier engage in a PIOP protocol that proves the correctness of operator L. At the end of this protocol, the verifier obtains (as illustrated in Fig. 3): (1) a proof of correctness for this operator, (2) claims about the model parameters used in this operator, (3) claims about the witness of this layer if any, and (4) most importantly, claims about the *input* to operator L. This way, the verifier converts the claim about the output of operator L into a claim about the output of operator $L-1$, a *virtual* claim [Tha25]. This process is repeated until the prover and verifier reach the input layer of the model. At this point, the verifier can directly evaluate the input layer operator. Using PCS evaluation proof from the prover, the verifier checks the claims about of the model parameters and witness values at each layer. If all evaluations claims are correct, along with the operator proofs, the verifier accepts the final output of the ML model as correct.

As an optimization, the verifier can delegate the evaluation of model and witness claims to the prover. The prover in turn uses the batching opening feature of the PCS to produce a single evaluation proof for all claims and witness claims across all layers.

Next, we present proving protocols for the main building blocks of LLMs. Due to space limitations, we describe here the proving logic and provide all detailed constructions in Appendix B, where we also include formal proofs for the round-by-round (knowledge) soundness of all our protocols which, enables us to show that our protocol is knowledge sound when making it non-interactive using the BCS transformation [BCS16].

4.3.1 Proving Embeddings. The embedding layer is used to map a token with a vector—from a pre-trained embedding table—which will be used as input during LLM inference. In more detail, let the vocabulary consist of t tokens with identifiers $\mathcal{V} = \{0, 1, \ldots, t-1\}$, and let each token be represented by a vector of dimension d, called the embedding dimension. Given an input sequence $\mathbf{x} = (x_0, x_1, \ldots, x_{s-1}) \in \mathcal{T}^s$ and an embedding matrix $\mathbf{T} \in \mathbb{F}^{v \times d}$, the embedding of $\mathbf{x}$ is defined as:

$$\mathbf{Y} = \text{Embed}(\mathbf{T}, \mathbf{x}) = \left(\mathbf{T}[x_0], \mathbf{T}[x_1], \ldots, \mathbf{T}[x_{s-1}]\right) \in \mathbb{F}^{s \times d}$$

In order to prove the above, someone could directly use any state-of-the-art lookup argument to prove the above [PH23]. In practice, however, this would result in large prover overhead since it would have to additionally commit and open to a polynomial of size $\Omega(sd)$. Instead, we observe that the verifier has direct access to the index vector $\mathbf{x}$ meaning that the verification cost can be proportional to $|\mathbf{x}| = s$. We thus propose a lightweight protocol that requires no additional commitments. Our key observation, is that we can re-write the Embed function as a matrix to matrix product which is much more efficient to prove using the sumcheck protocol [Tha13]. In particular, $\text{Embed}(\mathbf{T}, \mathbf{x}) = \mathbf{H}_{\mathbf{x}} \cdot \mathbf{T}$, where $\mathbf{H}_{\mathbf{x}} \in \mathbf{F}^{s \times t}$, is a sparse matrix containing s non-zero elements such that $\mathbf{H}_{\mathbf{x}}[i, \mathbf{x}_i] = 1$ and 0 everywhere else, for all $i \in [s]$. Note that since $\mathbf{H}_{\mathbf{x}}$ has only s non-zero elements, the prover does not have to commit to its MLE since the verifier can evaluate it locally in $O(s \log n)$ field operations.

4.3.2 Proving Re-quantization. Re-quantization is a fundamental operation in quantized inference, which enables the inference algorithm to turn a q'-bit quantized integer $x \in \mathbb{Z}$ into a q-bit quantized integer $y \in \mathbb{Z}$. As we discuss in C, this operation is usually invoked after fixed point multiplications to "lift" an integer to the original quantization level and to avoid overflows. At a very high level, re-quantization takes as input a vector $\mathbf{x} \in \mathbb{F}^n$ of values expected to be q'-bit and outputs $\mathbf{y} \leftarrow \text{ReQuant}(\mathbf{x}) \in \mathbb{F}^n$ such that:

$$\mathbf{y}[i] \leftarrow \begin{cases} 2^{q-1}, \text{ if } \mathbf{x}[i] \geq 2^{q'-1}, \\ -2^{q-1}+1, \text{ if } \mathbf{x}[i] \leq -2^{q'-1}+1 \\ \left\lfloor \frac{2^f \cdot \epsilon \cdot \mathbf{x}[i] + 2^{f+\delta-1}}{2^{f+\delta}} \right\rfloor - 2^{q-1}, \text{otherwise} \end{cases}$$

where δ and f are constants provided in a pre-processing phase.

Before we explain our final result, we first provide a high level idea of our protocol. We will initially discuss how to handle the simple case where all values of $\mathbf{x}$ are q'-bit numbers and later we will discuss the more general one.

Case 1: $\mathbf{x}$ has Correct Range. In this case, (which corresponds to the only scenario considered by prior works [SLZ24, QSL+25]), $\mathbf{y} = \left\lfloor \frac{2^f \cdot \epsilon \cdot \mathbf{x} + 2^{f+\delta-1}}{2^{f+\delta}} \right\rfloor - 2^{q-1}$. At first glance, this formula seems "SNARK-unfriendly" as it requires one division per element. Observe, however, that the divisor is a power of two number, and thus the division can be considered as a right shift which is easier to prove. Hence, the prover only has to show the correctness of a linear operation with constant scalars followed by a right shift operation of $f+\delta$ bits which can be efficiently proven using a PIOP for lookup arguments over decomposable tables [STW23, AST24] (see Appendix A for detailed definition). In particular, the PIOP prover decomposes the input to k-bit numbers, range-checks each chunk and finally keeps the q/k most significant chunks. For the rest of this section, we set $k = q$.

Case 2: Input has Incorrect Range. In this case, we need to construct a PIOP for the original ReQuant function. To achieve this, we will still rely on lookup arguments from decomposable tables. First, assume the largest element of $|\mathbf{x}|$ is a q''-bit number. Next, the prover decomposes $\mathbf{x}' = \text{Quant}(2^f \epsilon) \cdot \mathbf{x} + 2^{f+\delta-1}$ in $s = q''/q$ chunks $(\mathbf{x}'_1, \ldots, \mathbf{x}'_s)$ such that $\mathbf{x}' = \sum_{i \in [s]} 2^{(i-1) \cdot q} \mathbf{x}'_i$. It then uses these chunks as input into the lookup argument with tables defined as follows: The first $i \in [q'/q]$ tables simply range check the chunks (e.g., $\mathbf{T}_i[j] = j$ for all $j \in [2^q]$) while all other the tables are used to identify the elements that exceed the expected range. As we will elaborate in more detail next, these tables output 1 or -1 if a number has more than q'-bits and 0 otherwise. We then use the outputs of these tables to compute $\mathbf{y}$.

Protocol Description. Based on the above, we have everything we need to construct our PIOP for proving the correctness of requantization. Initially, the prover computes $\mathbf{x}' = \text{Quant}(2^f \epsilon) \cdot \mathbf{x} + 2^{f+\delta-1}$ and both parties interact following the lookup PIOP on input $\mathbf{x}'$. As discussed above, during this PIOP, the prover decomposes $\mathbf{x}'$ in s chunks $(\mathbf{x}'_1, \ldots, \mathbf{x}'_s)$ and passes the i-th chunk through the table $\mathbf{T}_i$ defined as follows:

(1) $\mathbf{T}_i[j] = j$, for all $j \in [2^q]$ and all $i \in [q'/q]$. These tables range check all chunks that belong to the expected range.
(2) $\mathbf{T}_i[j] = 0$, for all $j \in [2^q] \backslash \{0\}$ and $\mathbf{T}_i[0] = 1$, for all $i \in \{q'/q, s-1\}$. These tables check if an element belongs to

an incorrect range by returning 0 if a chunk is non-zero, else 1.

(3) $\mathbf{T}_s[j] = -1$ if $j \in \{1, \ldots, 2^{q-1} - 1\}$, $\mathbf{T}_s[j] = 1$ if $j \in \{2^{q-1} - 1, 2^q - 1\}$ and $\mathbf{T}_s[0] = 0$. Note that this table, applied to the most significant bit, not only checks if an element $\mathbf{x}'_1[j]$ is in the correct range but also, if it is, then it outputs -1 or 1 depending on whether $\mathbf{x}_1[j]$ is negative or positive.

Furthermore, the lookup PIOP shows the correctness of the decomposition by proving that $\mathbf{x} = \sum_{i\in[s]} 2^{(i-1)q}\mathbf{x}'_i$ and the correctness of the requantized output $\mathbf{y}$ by showing that:

$$\mathbf{y} = \text{InRange}(\{\mathbf{T}_i, \mathbf{x}_i\}_{i\in[q'/q,s])} \cdot (\mathbf{T}_{q'/q}[\mathbf{x}_{q'/q}] - 2^{q-1}) + (1 - \text{InRange}(\{\mathbf{T}_i, \mathbf{x}_i\}_{i\in[q'/q,s])}) \cdot (\frac{2^n - 1}{2}\mathbf{T}_s[\mathbf{x}_s]) + \frac{1}{2})$$

where the InRange function outputs 1 if $\mathbf{x}'[i]$ is a q' bit number and 0 if it is larger. More formally, InRange is defined as:

$$\text{InRange}(\{\mathbf{T}_i, \mathbf{x}_i\}_{i\in[q'/q,s])}) = (1 - \mathbf{T}_s[\mathbf{x}_s]^2) \prod_{i=q'/q+1}^{i=s-1} \mathbf{T}_i[\mathbf{x}_i]$$

It is not hard to see that the above function outputs 1 if and only if the most significant $s - q'/q$ chunks of a number $\mathbf{x}'[j]$ q'/q are zero. Finally, observe that if $\mathbf{x}'[j]$ is a q'-bit number then Comb_y outputs $\left\lfloor \frac{2^f \cdot \epsilon \cdot \mathbf{x}[j] + 2^{f+\delta-1}}{2^{f+\delta}} \right\rfloor - 2^{q-1}$. Furthermore, if $\mathbf{x}'[j]$ is larger then Comb_y outputs $-2^{n-1} + 1$ or 2^{n-1}, depending on the sign of $\mathbf{x}[j]$ decided by the table $\mathbf{T}_s$.

4.3.3 Proving RMSNorm. The RMSNorm layer takes as input a matrix $\mathbf{X} \in \mathbb{F}^{s\times d}$ and outputs $\mathbf{Y} \leftarrow \text{RMSNorm}(\mathbf{X})$ where:

$$\mathbf{Y}[i,j] = \frac{\mathbf{X}[i,j]}{\sqrt{\frac{1}{d}\sum_{j\in[d]} \mathbf{X}[i,j]^2}}, \forall i, j \in [s] \times [d]$$

The "standard" approach of proving the above, which was also adopted by all prior works [SLZ24, QSL+25], is to first construct a circuit that directly computes the RMSNorm layer and subsequently invoke a PIOP for proving the correct computation of that circuit. Unfortunately, this approach not only incurs significant prover overheads since it contains many "SNARK-unfriendly" operations such as square roots and inverses, but also leads to a significant accuracy loss. Namely, these techniques usually require representing the square root and the inverse as quantized values, which introduces additional rounding error at each step. To overcome these limitations, we propose an alternative approach that is significantly more efficient as it does not involve square roots or inverses and also incurs negligible accuracy loss.

To achieve this, the prover computes an additional non-deterministic witness $\mathbf{d} \in \mathbb{F}^s$ where $\mathbf{d}[i] = \text{Quant}(1/\sqrt{\frac{1}{d}\sum_{j\in[d]} \mathbf{X}[i,j]^2})$ and uses it to prove the correct computation of $\mathbf{Y}$ by showing that $\mathbf{Y}[i,j] = \mathbf{X}[i,j] \cdot \mathbf{d}[i]$, for all $i, j \in [s] \times [d]$. Unfortunately, this is not enough since the verifier has no clue on whether $\mathbf{d}$ has been computed honestly. To check the well-formedness of $\mathbf{d}$, the prover shows to the verifier that $\sum_{j\in[d]} \mathbf{Y}[i,j]^2 \approx n\frac{\sum \mathbf{X}[i,j]^2}{(\sqrt{\sum \mathbf{X}[i,j]^2})^2} = 1$.

Protocol Description. Based on the above discussion, we construct our PIOP which we present in **Protocol 10**. For completeness we will present a slightly modified variant of the protocol described above which also performs requantization of the output $\mathbf{Y}$ in the desired quantization level. With that in mind, we partition our protocol in two basic phases. In the first phase the prover shows the correct computation of $\mathbf{Y}$ while in the second phase the prover shows the correctness of $\mathbf{d}$.

Phase 1: In this phase, the prover shows that $\mathbf{Y} = \left\lfloor \frac{2^{\delta_1} \cdot \mathbf{X} + 2^{\delta_2}}{2^{\delta_3}} \right\rfloor - 2^{q-1}$, where $\delta_1, \delta_2, \delta_3$ are constants the depend on the scaling factors of the specific layer and $\mathbf{X}'[i,j] = \mathbf{X}[i,j] \cdot \mathbf{d}[i]$, for all $i, j \in [s] \times [d]$. To achieve this, the prover first computes $\mathbf{d}$ and $\mathbf{X}'$ and sends $[\![f_{\mathbf{d}}]\!]$ to the verifier. Next, both parties run the PIOP for $\mathcal{R}_{requant}$ to prove the correct computation of $\mathbf{Y}$. Note that the verifier does not hold an oracle to the MLE of $\mathbf{X}'$. Instead, it only receives an evaluation claim $y_{\mathbf{X}'}$ of $f_{\mathbf{X}'}$ which is reduced to an evaluation claim of $f_{\mathbf{X}}$ by running the sumcheck protocol for the instance:

$$y_{\mathbf{X}'} = \sum_{\mathbf{x}_1\in\{0,1\}^{\log s}} f_{\mathbf{d}}(\mathbf{x}_1) \sum_{\mathbf{x}_2\in\{0,1\}^{\log d}} f_{\mathbf{X}}(\mathbf{x}_1, \mathbf{x}_2)$$

At the end of that protocol, the verifier uses $[\![f_{\mathbf{d}}]\!]$ and $[\![f_{\mathbf{X}}]\!]$ to validate the correctness of the received evaluation claims.

Phase 2: In this phase, the prover shows the well-formedness of $\mathbf{d}$ by showing that $\sum_{j\in[d]} \mathbf{Y}[i,j] \approx \text{Quant}(\frac{d}{\delta_4})$, where δ_4 a constant selected by the prover in a pre-processing phase. For this, the prover first computes the vector $\mathbf{y}_s \in \mathbb{F}^s$ such that $\mathbf{y}_s[i] = \sum_{j\in[d]} \mathbf{Y}[i,j]$ and then uses it as input in the lookup PIOP to show that each element of $\mathbf{y}_s$ belongs to the correct range. We have an analysis on the error bound to enforce in Appendix C. At the end of that protocol, the verifier holds an evaluation claim y_s of the MLE of $\mathbf{y}_s$ and reduces it to an evaluation claim of the MLE of $\mathbf{Y}$ by interacting over the sumcheck protocol for the instance :

$$y_s = \sum_{\mathbf{i}\in\{0,1\}^{\log s}} \beta(\mathbf{i}, \mathbf{r}_s) \sum_{\mathbf{j}\in\{0,1\}^{\log s}} f_{\mathbf{Y}}(\mathbf{i}, \mathbf{j})$$

Finally, the verifier validates that claim using $[\![f_{\mathbf{Y}}]\!]$.

4.3.4 Proving LayerNorm. The LayerNorm layer takes as input a matrix $\mathbf{X} \in \mathbb{F}^{s\times d}$ and outputs: $\mathbf{Y} \leftarrow \text{LayerNorm}(\mathbf{X})$ such that

$$\mathbf{Y}[i,j] = \frac{\mathbf{X}[i,j] - \mu_i}{\sqrt{\frac{1}{d}\sum_{j\in[d]} (\mathbf{X}[i,j] - \mu_i)}}, \forall i, j \in [s] \times [d]$$

where μ_i, the mean of the i-th row of $\mathbf{X}$.

As a starting point, note that for every $i \in [s]$, it holds that:

$$\text{LayerNorm}(\mathbf{X}[i,:]) = \text{RMSNorm}(\mathbf{X}[i,:] - \mu_i)$$

Hence, to prove the the correct computation of a LayerNorm, it is enough to construct a PIOP for showing (1) the correct computation of RMSNorm with input $\mathbf{X}'[i,j] = \mathbf{X}[i,j] - \mathbf{d}[i]$ and (2) that $\mu_i = \frac{1}{d}\sum_{j\in[d]} \mathbf{X}[i,j]$. Since we explained how to prove (1) in the previous Section, we will focus on showing how to efficiently prove the correctness of each μ_i. For this, the prover will provide the vector $\mathbf{m} = (\mu_1, \ldots, \mu_s)$ as a non-deterministic input and then will prove its well-formedness by showing that $\text{Quant}(\frac{1}{d})\sum_{j\in[d]} \mathbf{X}[i,j] \approx \mu_i$ for all $i \in [s]$—via a lookup argument.

4.3.5 Proving Multi-Head Attention Layers. Before delving into the details of our main protocol, recall that a multi-head attention layer (MHA) consists of multiple invocations of a single-head attention layer with inputs consecutive columns of the matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{F}^{s\times d}$. In more detail, given the input matrix $\mathbf{X} \in \mathbb{F}^{s\times d}$, we want to

prove the correct computation of $\mathbf{Y} = [\mathbf{Y}_1, \ldots, \mathbf{Y}_h]$ such that for all $i \in [h]$, $\mathbf{Y}_i = \text{Softmax}(\mathbf{Q}_i \cdot \mathbf{K}_i^T / \sqrt{d/h})\mathbf{V}_i$, where $\mathbf{Q} = [\mathbf{Q}_1, \ldots, \mathbf{Q}_h] = \mathbf{X} \cdot W_Q$ (and likewise we define $\mathbf{K}$ and $\mathbf{V}$).

To ease explanation, we first describe the protocol for proving an MHA layer when d/h is a power of two. Later we will show what modifications we need to apply in order to overcome that restriction. With that in mind, one can prove the correct computation of an MHA layer by *batch proving* h instances of a single head attention layer (see **Protocol 14** of Appendix B.8). This can be done in the following way: First, the prover computes $\mathbf{T}_{p,i} = \mathbf{Q}_i \cdot \mathbf{K}_i^T$, $\mathbf{T}_{s,i} = \text{Softmax}(\mathbf{T}_{p,i}/\sqrt{d/h})$ and sends $[\![f_{T_{s,i}}]\!]$ for all $i \in [h]$.

Then, the prover shows to the verifier that $\mathbf{Y}_i = \mathbf{T}_{p,i} \cdot \mathbf{V}_i, \forall i \in [h]$. Note that at the end of this step, the verifier will end up with evaluation claims of $y_{\mathbf{Y}_i}$, for the MLE of each $\mathbf{Y}_i$ at a random point $\mathbf{r}$. However, it can not directly check their correctness since it only has the oracle of the MLE of $\mathbf{Y}$—the concatenation of all $\mathbf{Y}_i$. Hence, it has to derive an evaluation claim of $f_{\mathbf{Y}}$ using the claims of $f_{\mathbf{Y}_i}$. Since d/h is a power of two, this can be done by computing $y_Y = \sum_{i \in [h]} \beta(\mathbf{i}, \mathbf{r}_h) y_{\mathbf{Y}_i}$, for a randomly sampled point $\mathbf{r}_h \in \mathbb{F}^{\log h}$.

Subsequently, both parties interact over the batched variant of the PIOP for proving softmax (see **Protocol 13** Appendix B.7 for more details) to prove the correctness of $\mathbf{T}_{s,i} = \text{Softmax}(\mathbf{T}_{p,i}/\sqrt{d/h})$. At the end of this protocol, the prover ends up with an evaluation claim for each $f_{T_{p,i}}$ which are used in step to prove the correctness of $\mathbf{T}_{p,i} = \mathbf{Q}_i \cdot \mathbf{K}_i^T$ for all $i \in [h]$ using the PIOP for batched matrix multiplication. At this point, the verifier ends up with evaluation claims of the MLEs of $\mathbf{Q}_i, \mathbf{K}_i^T$ and $\mathbf{V}_i$ for all $i \in [h]$. It then recombines these claims into one for the MLE of $\mathbf{Q}, \mathbf{K}^T$ and $\mathbf{V}$ in the same way as it did for $\mathbf{Y}$.

Finally, the prover uses the evaluation claims of the matrices $\mathbf{Q}, \mathbf{K}^T$ and $\mathbf{V}$ to prove that $\mathbf{Q} = \mathbf{X} \cdot \mathbf{W}_Q$ (and likewise for $\mathbf{K}$, $\mathbf{V}$) using the PIOP for proving matrix multiplications [Tha13].

Handling the Case When d/h is not a Power of Two. In the above protocol, we assumed that $d_k = d/h$ was a power of two, which allowed the verifier to efficiently compute the evaluation claims of the MLEs of Y, Q, K^T and V from the corresponding submatrices. Unfortunately, this algorithm does not work when $d_k = d/h$ is not power of two—which is usually the case.

To overcome this issue, the prover first needs to pad the matrices $Y, Q, K, V \in \mathbb{F}^{s \times d}$ to $\hat{Q}, \hat{K}, \hat{V} \in \mathbb{F}^{s \times \tilde{d_k} \cdot h}$, where $\tilde{d_k}$ the next power of two of d_k, such that $\hat{Y}_i, \hat{Q}_i, \hat{K}_i, \hat{V}_i \in \mathbb{F}^{s \times \tilde{d_k}}$ where the last $\tilde{d_k} - d_k$ columns are zero. In addition, it has to show that these matrices have been correctly padded. This is done by using the PIOP for matrix multiplication showing that (i) $\hat{Y} = Y \cdot P_1$ (and likewise for all other matrices), where $P_1 \in \{0,1\}^{d \times \tilde{d_k} \cdot h}$ is a padding matrix consisting of exactly d non-zero elements and properly re-arranges the columns of Q. Note that since P_1 has only d non-zero elements, the verifier can evaluate its MLE using only $\tilde{O}(d)$ field operations.

4.3.6 Handling Non-Determinism. So far, we have described our prover under the assumption that inference is deterministic. However, unlike many standard ML models such as CNNs or MLPs, LLM inference can be randomized. For example, during decoding, the model may sample the next token from the top-k most probable candidates instead of deterministically selecting the highest-probability token. In practice, this is not a limitation since any randomized algorithm can be "de-randomized" by providing a single *purely* random seed as additional input. Consequently, to handle non-deterministic inference, `DeepProve` simply generates a proof for our certification algorithm (the correctness of which is guaranteed as long as it takes as the initial random seed) using the PIOPs we described above together with the techniques proposed by [KPPS20, PP24] for verifying the de-randomized parts of the inference. Typically, those parts correspond to the final decoding stage, which incurs a negligible prover overhead. In reality, the main challenge comes from actually showing that the initial random seed has been sampled correctly. This can be ensured by either using a public verifiable source of randomness like a blockchain or through an interactive protocol with the verifier (Section 4.3 of [PGL+26]). We note that we do not include this in our implementation, same as all prior works on verifiable LLM inference [SLZ24, QSL+25].

4.4 Proofs of LLM Non-Hallucination

Despite LLM's impressive generative capabilities, they often generate hallucinated content [JLF+23, ZPM+24], i.e., outputs that are syntactically plausible yet semantically false or unsupported by evidence. Such hallucinations undermine the trust of users and limit their applicability in scientific, medical, and legal settings.

To address the issue of hallucinations, we develop a module in DeepProve that proves the response generated by the LLM model is not hallucinated in zero-knowledge, together with the proof of inference. There is a rich literature on hallucination detection algorithms for LLM [MLG23, SBS+24, AKKP25, NWZ+24, OTG+25, GCX+25], see [ea25b] for a survey of the topic. These algorithms can be classified into several categories depending on the settings and information needed. For example, there are detection algorithms with black-box access to the LLM with only the output or the logits of the generated tokens [MLG23, FKKG24], while others need white-box access with internal states of LLM [CLC+24, SBS+24]. There are algorithms utilizing external reference databases via retrieval-augmented generation (RAG) [MAB+24, LYZ24], and there are algorithms with post-processing possibly by another LLM [FKKG24].

We first identify the appropriate category of hallucination detection algorithms that is friendly to prove. In our setting, as the prover knows all the information of the LLM and already computes the internal states, we utilize the algorithms with white-box access to improve the accuracy of detection. Moreover, as we aim to generate the proof of non-hallucination together with the proof of inference, those algorithms with external databases or post-processing are not desirable. Therefore, we choose the state-of-the-art white-box hallucination detection algorithm without external knowledge in [SBS+24] named LLM-check in our protocol. Our technique can be readily generalized to other detection algorithms in this category as the field of hallucination detection continues to evolve.

In LLM-check [SBS+24], detection is based on the *Attention Score*[1], with kernel similarity of Q and K in the attention layer defined as $Ker(Q, K) = \text{softmax}(\frac{Q \cdot K^T}{\sqrt{d}})$. Note that it is the same as the matrix T_s computed in Appendix B.8 to prove the attention layer. The attention score is then defined as $\log \det(Ker(Q, K)) =$

[1]The paper also proposes the hidden score, but its performance is worse than the attention score in the experiments.

$\sum_{i=0}^{s-1} \log Ker_{i,i}$, the logarithm of the determinant of matrix $Ker(Q, K)$. As the matrix is lower triangular after masking, its determinant is the product of the elements on the diagonal, denoted as $Ker(i, i)$. For multi-head attention, the attention score is aggregated from the kernel similarity of all heads by taking the average. The attention score is then compared with a threshold to determine whether the response is hallucinated or not. The computation of the attention score can be easily proven as a sub-protocol in DeepProve. As mentioned above, the matrix Ker has already been computed during the inference. We use a lookup argument to approximate the logarithm function, followed by a sumcheck protocol to compute the attention score as a summation. The protocol is presented in Appendix B.9.

4.5 Distributed DeepProve Prover

Here, we discuss how the DeepProve proving process can be distributed across multiple machines that jointly produce the proof of the inference process. This not only reduces the per-machine workload but also the corresponding memory footprint, moving towards more scalable solutions.

In a nutshell, we choose to distribute the proving workload horizontally by partitioning the DAG representing the LLM into subsets of *consecutive* layers, hereby called *chunks*. Each chunk can be considered as a standalone DAG, which can be proven as in the single-prover case. The key insight for DeepProve's distribution mechanism is that the three steps of proving (witness commitment, proving claims, and commitment opening) can be performed *on a subset* of the graph with little added cost, and without relying on specific features of the underlying cryptographic primitives. By committing to inputs/outputs of *each chunk*, proving of each chunk becomes independent from all the other chunks, which means that all the chunks can be proven simultaneously.

In practice, the server holding the inference trace (*coordinator*) splits it into subtraces—where each subtrace corresponds to a specific partition—and distributes them to *workers*. Upon receiving the subtrace, each worker runs the proving algorithm of DeepProve as if the partition is a full graph in itself. Note that this step produces opening proofs *both* for the witness corresponding to the subgraph *and* the model polynomials *used* to produce the subtrace. Finally, the final proof is simply the concatenation of all sub-proofs –every worker sends it sub-proof to the coordinator as soon as it is done.

Note that each worker commits to the *inputs* and *outputs* of each chunk. To correctly bind together all the chunks, it is necessary for the verifier to check that each input of each chunk corresponds to an output of the next chunk (and similarly for outputs). To enforce this property, the verifier needs to check that (i) The partitioning of the graph chosen by the prover is consistent with the original model DAG (e.g., the prover did not omit nodes or edges in the chunks), and (ii) each commitment to an input of a chunk corresponds to the commitment of the corresponding output of the next chunk (resp. for outputs). Our horizontal partitioning strategy offers two very useful advantages. First, observe that once the inference trace for all the layers in a chunk is ready, the coordinator can immediately send this portion of the trace to the corresponding worker to start proving. This asynchronous communication can reduce the latency required to send the trace to the workers (except for the last chunk of the model). Second, unlike many existing distributed prover schemes (e.g., [XZC+22, LXZ+24]), our approach requires a single round of interaction between workers and coordinator, which can again significantly reduce latency, especially over WAN.

Model	PCS	Seq. Len	Context (min)	RAM (GiB)	Prove (min)	Verify (s)	π (MiB)	TPM
GPT-2	BaseFold	64	< 1.38	18.47	0.8	1.72	20.68	80.5
		128		20.95	1.22	1.85	21.83	104.63
		256		27	2.18	1.97	23.04	117.59
		512		41.98	4.44	2.12	24.33	115.26
	HKZG	64	< 0.86	24.41	1.09	1.81	8.98	58.68
		128		30.63	1.44	1.99	9.89	89.03
		256		45.01	2.23	2.2	10.83	114.87
		512		78.7	4.02	3.22	11.8	127.23
Gemma 3	BaseFold	64	< 3.86	45.52	1.85	2.45	36.31	34.62
		128		51.66	2.82	2.52	38.59	45.37
		256		63.55	4.77	2.66	41.07	53.62
		512		96.27	8.56	2.88	43.6	59.8
	HKZG	64	< 3.02	57.65	2.62	2.59	12.62	24.45
		128		69.92	3.33	2.78	13.98	38.46
		256		94.52	4.77	2.98	15.25	53.67
		512		147	7.93	3.43	16.66	64.53

Table 1: Performance of DeepProve on our primary machine. TPM denotes the throughput in tokens per minute.

5 Evaluation

In this section, we evaluate DeepProve's performance and accuracy.

Benchmarking setup. Our primary benchmarking machine has AMD EPYC 9254 with 24 cores (2.9 GHz) and 504 GB RAM. However, for smaller models and sequence lengths, we also benchmark on a secondary machine with AMD Ryzen 9 7950X3D with 16 cores (4.2 GHz) and 128 GB RAM. Observe that our primary machine has significantly more memory and CPU cores, which allows us to benchmark larger models and longer sequence lengths. On the other hand, the CPU on our secondary machine is faster, leading to higher throughput (Table 2). Unless otherwise specified, all experiments are run on the primary machine.

We implemented the non-interactive variant of DeepProve in Rust using Scroll Technologies' Ceno library [Scr24], which is based on the Goldilocks field. We instantiate DeepProve using the BaseFold [ZCF24] PCS (BF). Furthermore, we extend the library to also support BN254 elliptic curve and HyperKZG [ZSCZ25] PCS (HKZG), so we overall have two versions of DeepProve with different trade-offs (e.g., the Basefold version is plausible post-quantum secure, while the HyperKZG version achieves shorter proofs). We provide an implementation of DeepProve in the following repository [cod].

Experiments. For various maximum sequence lengths, we run inference on the GPT-2 [RWC+19] and Gemma 3 [Tea25] models (with 124M and 270M parameters respectively) using DeepProveusing 12-bit quantization level. We measure the following parameters: (1) time taken to generate the cryptographic context, which includes computing the PCS parameters and committing to the model weights; (2) time taken to prove the end-to-end inference, excluding the inference time itself since various ML libraries are already optimized for inference; (3) verification time; (4) proof size; (5)

Model	PCS	Seq. Len	Context (min)	RAM (GiB)	Prove (min)	Verify (s)	π (MiB)	TPM
GPT-2	BaseFold	64	< 0.91	18.33	0.57	1.35	20.66	112.86
		128		20.82	0.85	1.45	21.82	150.98
		256		27.02	1.46	1.55	23.02	175.08
		512		41.85	2.94	1.65	24.33	174.32
	HKZG	64	< 0.79	24.34	0.91	1.45	8.98	70.55
		128		30.28	1.2	1.59	9.89	106.96
		256		44.73	1.88	1.74	10.83	135.9
		512		78.49	3.51	2.59	11.8	146.03
Gemma 3	BaseFold	64	< 2.64	45.38	1.36	1.91	36.21	46.95
		128		51.77	1.99	2.02	38.6	64.38
		256		62.96	3.26	2.08	41.04	78.59
		512		96.02	5.9	2.13	43.6	86.76
	HKZG	64	< 2.47	57.59	2.08	2.04	12.62	30.73
		128		69.54	2.71	2.19	13.91	47.17
		256		96.59	4.03	2.34	15.25	63.45
		512		OOM				

Table 2: Performance of `DeepProve` on our secondary machine. TPM and OOM denote the throughput in tokens per minute and memory exhaustion respectively.

space-utilization and (6) throughput in tokens per minute (TPM). Especially, this last metric is very useful for comparison of different systems, as most works in the area are not fully open-sourced. The results of these experiments can be found in Table 1 and Table 2 for primary and secondary machine, respectively. Finally, in Table 7 we also include a more detailed analysis of the prover cost by presenting a layer-wise breakdown of the prover time.

Prover time and context generation time. Overall, our prover time scales linearly with the model size. At the same time, our prover time increases with larger sequence length—as expected—but this increase is not strictly linear. This stems from the sub-linear prover cost of certain operations like matrix-to-matrix multiplications. As we will discuss next, this results in faster TPMs when sequence length grows leading to a lower amortized prover cost per token. We observe that HyperKZG (HKZG) has better scaling properties than BaseFold as the sequence length increases. This is due to HKZG being more efficient to compute commitments to witness polynomials, which become more relevant on the overall proving time with higher context lengths; our implementation of HKZG is also optimized for small bitlength scalars, which is exactly what the trace contains in the case of LLMs. Furthermore, since HKZG has homomorphic properties, `DeepProve` actually randomly combine all polynomials to be opened in a single one first before opening, while Basefold still deals with independent polynomials during opening. Regardless of the PCS, Gemma 3 has a higher prover time than GPT-2 due to its larger model size. On the other hand, context generation time is relatively stable for variable sequence lengths since PCS-related operations dominate this phase.

Throughput. We observe that the throughput (in TPM) increases with the sequence length for both GPT-2 and Gemma 3. Our peak throughput on the primary machine is 127 TPM for GPT-2 and 64 TPM for Gemma 3, both at sequence length 512 using HyperKZG. However, on the secondary machine, we achieve a throughput of 174 TPM and 146 TPM for GPT-2 at sequence lengths 512 using BaseFold and HyperKZG, respectively (Table 2)! This is because the CPU on the secondary machine has a higher clock speed, leading to faster computation despite having fewer cores and less memory. At comparable sequence lengths, the throughput on the secondary machine is around 1.5× higher than the primary machine. Remarkably, the best TPM we achieved in our experiments is 175, obtained for GPT-2 at sequence length 256 using the secondary machine.

Breakdown of Prover Cost. Table 7 reports a detailed prover cost breakdown of `DeepProve` with BaseFold for GPT-2 and Gemma 3 inference at sequence length 512. Specifically, it reports (1) the prover time spent on the PIOP and PCS components (top), and (2) a finer-grained decomposition of these costs across different layer types (middle-bottom). We observe that, as expected, PCS constitutes the main bottleneck of the prover; however, it does not fully dominate the total runtime. This is primarily due to our optimized GKR-like PIOPs, which reduce the amount of data that must be committed, thereby lowering commitment overhead. Another interesting observation is that in both PIOP and PCS, more than 50% of the prover cost comes from activation functions, Softmax, and requantization, with the latter being the most expensive component overall, accounting for 27% of total prover time. Consequently, optimizing any of these components would yield substantial improvements in overall prover performance.

Verifier time and proof size. The verifier time marginally increases with the sequence length, but is still under 3.2 seconds for GPT-2 and under 3.5 seconds for Gemma 3, even at the maximum sequence length. Our proof sizes are between 9-43MiB depending on the model, PCS, and sequence length. Naturally, HyperKZG consistently produces smaller proofs than BaseFold.

Memory Utilization. Table 2 also reports the memory utilization of our scheme. Overall, memory consumption increases with both the sequence length and, as expected, it scales linearly with the model size. This agrees with the space complexity of our prover which is proportional to the circuit size being proven. Furthermore, using HKZG results in higher memory utilization than BaseFold due to its larger field size. Concretely, for the largest evaluated instance, our scheme consumes 96GB and 147GB of RAM with BaseFold and HKZG respectively. In fact, the latter was not able to run in the secondary machine. This makes memory utilization the main bottleneck of our prover (e.g., proving models with more than 3 billion parameters, such as GPT-J, would likely require over a terabyte of RAM). We leave as an interesting future direction efficiently reducing total memory utilization using space-efficient [PP24, PP25] or disk-based [BMM+24] SNARKs.

Baseline comparison with prior works. The ideal way to benchmark `DeepProve`'s performance against prior works such as zkGPT [QSL+25] and zkTorch [CTK25] would be running all of them on the same hardware and language model. However, the full source code for these works is not available. Likewise, zkLLM [SLZ24] provides access only to their attention mechanism source code; this would require significant engineering effort to extend it to a full LLM inference proof system. Under these constraints, and similar to prior works [QSL+25], we compare `DeepProve`'s performance based on the numbers reported in their papers, while highlighting the differences in the experimental setup, in Table 3.

Tool	Model	CPU	System RAM	# Th.	TPM
zkGPT [QSL+25]	GPT-2	2.6 GHz 16 cores	200 GB	32	2.75
zkTorch [CTK25]	GPT-J	2.6 GHz 32 cores	4 TB	64	0.09
Ours	GPT-2	2.9 GHz 24 cores	504 GB	48	127.23
	GPT-2	4.2 GHz 16 cores	128 GB	32	175.08

Table 3: Baseline throughput and hardware setup from [CTK25, QSL+25]. We exclude the comparison with zkLLM [SLZ24] since their performance numbers rely on GPU acceleration and are not replicable in CPU environments. TPM is tokens per minute and # Th is the number of threads used. Here RAM refers to maximum available system RAM, which is not necessarily the RAM used by the implementation. Neither zkGPT [QSL+25] nor zkTorch [CTK25] reports the actual RAM usage.

Accuracy. Here, we test the accuracy that DeepProve achieves. Recall that the main variable affecting this is the quantization bit-length, hence we report accuracy for three different lengths: 8, 10, and 12. Table 4, reports the accuracy statistics on the quantized output logits and compare them with the baseline from running the initial floating-point model on PyTorch. We compute the perplexity over the Wikitext-2 dataset by running a forward pass with the first 256 tokens of 25 documents in the dataset. We then compute the perplexity score of the first 255 rows of the logit tensor output by the model for these 256 tokens, using the input tokens (shifted by 1 and omitting the first token) as the ground truth, for both the baseline and DeepProve's quantized engine. Naturally, the higher quantization bit lengths provide better perplexity measures (lower is better), and DeepProve is close to the baseline. We note that for Gemma 3, anything below 12-bit quantization performs poorly for accuracy. The activations in Gemma 3 are susceptible to *extremely* large outliers, and due to the presence of additional RMSNorm layers in the transformer blocks, traditional smoothing techniques cannot be applied.

Impact of quantization in prover performance. For the three quantization levels described above, we also investigated the impact of quantization in prover complexity. Overall, increasing the quantization level from 8 to 12 bits adds only a 0.8% additional overhead in prover time. This is because quantization mainly affects prover time when the number of lookup tables used to prove activation and re-quantization layers increase which is not the case when moving from 8 to 12 bits. Simply put, our quantization optimizations, enable us to obtain as fast prover times as if we used 8 bit quantization while simultaneously attaining near-perfect perplexity.

Proof of non-hallucination. To demonstrate the performance of our proof of non-hallucination, we test it on the Fava Annotation dataset [MAB+24]. As introduced in [SBS+24], the dataset itself spans four different data sources: Knowledge-intensive queries sampled from the Open Assistant dataset [ea23], Open QA prompts from the No Robots dataset [RTB+23], Instruction-following and WebNLG [GSNPB17] datasets. It has 460 samples in total. We compute the attention score of each attention layer, and present standard threshold-based detection metrics such as Accuracy and Area under the ROC curve (AUROC) on balanced datasets with an equal number of positive and negative results. Our experiments are in the same setting as those in [SBS+24].

Model	Quant. (bits)	**Cosine Similarity**	**Perplexity (Baseline)**	**Perplexity (Test)**
GPT-2	8	0.9967	49.2198	60.9273
	10	0.9993		50.0806
	12	0.9966		49.4863
Gemma 3	8	0.2035	515.46	> 1150
	10	0.6998		1150.43
	12	0.9587		454.72

Table 4: Accuracy of DeepProve's quantized inference engine compared to the baseline floating-point model.

The performance of our scheme is presented in Table 5. We report the best AUROC and accuracy from all layers, which was done in [SBS+24]. As shown in the table, the highest AUROC is 68.69% (L8) in our quantized GPT-2 model and 65.84% (L18) in our quantized Gemma 3 model, while the highest accuracy is 67.96% for GPT-2 and 64.16% for Gemma 3. These numbers are close to the performance of the hallucination detection algorithm for Llama-2 and Llama-3 as reported in [SBS+24], confirming the effectiveness of the detection.

To understand the impact of our quantization, we also tested the detection algorithms on the corresponding models over floating-point numbers obtained from HuggingFace [RWC+19, Tea25]. In the original GPT-2 model, the highest AUROC is 68.29% and the highest accuracy is 65.62%. Our performance becomes slightly better after quantization. For the original Gemma3 model, the highest AUROC is 70.5% and the highest accuracy is 65.87%. The performance becomes slightly worse after quantization.

We also evaluate the prover time of the hallucination detection on machine 1. Because it is integrated as a node in the model, and the witnesses are committed together with all other nodes, we only measure the prover time of the sumcheck PIOP. As shown in Table 5, the prover time for sequence length 512 is only tens of milliseconds. The overhead of this gadget is only a tiny portion of the prover time of the verifiable inference, as shown in Tables 1 and 2.

Distributed proving. We simulate the performance of distributed DeepProve on 8 and 16 workers and a coordinator using our secondary machine configuration as the engineering effort to set up a distributed environment is non-trivial. We measure the *maximum* time to prove any chunk and add the time to transfer each individual proof back to the coordinator, assuming a LAN connection at 1 Gb/s. In this way, we get the worst-case performance of the distributed setup.

Table 6 reports our findings when runing our distributed prover with 8 (top) and 16 (bottom) workers for GPT-2 with BaseFold, and sequence lengths ranging from 128-1024. Overall distributed proving significantly improves the prover time and memory usage of each worker. First, note that distributed proving allows proving bigger models with the same computational resources, e.g., GPT-2 at sequence length 512 is proven at a remarkable throughout of

Model	AUROC (%)	Accuracy (%)	Prove (ms)
GPT-2	68.69	67.96	98.1
Gemma 3	65.84	64.16	21.1

Table 5: Performance of proof of non-hallucination in DeepProve– 512 sequence length.

# Mach.	Model	PCS	Seq. Len	Context (min)	Prove (s)	Verify (s)	π (MiB)	TPM	Single TPM	Speedup
9	GPT-2	BaseFold	128	< 1.25	9.83	1.9	40.62	780.9	150.98	5.17×
			256		14.58	2	41.9	1053.25	175.08	6.02×
			512		28.45	2.11	43.28	1079.71	174.32	6.19×
			1024		68.5	2.26	44.88	896.2	–	–
17	GPT-2	BaseFold	128	< 1.24	9.75	2.15	50.39	788	150.98	5.22×
			256		9.69	2.28	51.97	1585.4	175.08	9.05×
			512		16.56	2.39	53.81	1855.1	174.32	10.64×
			1024		39.5	2.5	55.49	1547.1	–	–

Table 6: Simulated performance of distributed `DeepProve` using worker machines with our secondary machine configuration. TPM denotes the throughput in tokens per minute, Single TPM is the single-machine throughput, and Speedup is TPM/Single TPM.

1079TPM for 8 workers and 1855TPM for 16—achieving 6.74× and 10.64× improvement, respectively. At the same time, each prover needs significantly less memory than the non-distributed one.

Most importantly, however, our distributed prover has a minimal communication overhead. Excluding the setup step where the coordinator distributes the trace, it only requires a single round of interaction and the total communication cost corresponds to the proof size which is 43.2MB for 8 workers and 53.81MB for 16 workers. This makes our system suitable for WAN settings. Indeed, for a WAN with lower bandwidth (100 Mb/s), `DeepProve` achieves 1054TPM at context lengths 512 and 8 machines—only a 2% slowdown! We leave the full-fledged implementation of distributed `DeepProve` as future work.

6 Related Work

Here we discuss related work on cryptographically verifiable machine learning. Orthogonal to ours, some works focus on proving training correctness for neural networks [APKP24], logistic regression [GGJ+23], and decision trees [PP24], or proving proximity to an optimal model for convex-loss algorithms [TGM+25]. We next discuss works focusing on verifiable inference which are more relevant to ours.

"Classical" ML models. Early works on verifiable inference focused on simpler ML models (as opposed to LLMs), such as decision trees [WYX+21, ZFZS20, PP24] and classical neural networks with few layers [KHSS22, FQZ+21]. Subsequent works proved the correctness of convolutional neural network inference [BFGV+23, LKKO24, LXZ21]. The techniques used in these works range from encoding the CNN computation in an arithmetic circuit [BFGV+23, LKKO24, ea25a] to using specialized sum-check protocols [LXZ21] that are closer in spirit to our approach, albeit for simpler models. Another line of work focused on building generic frameworks for verifiable machine learning inference [LWQ+24, CCYY25, MY25, zko24]. Broadly, these works trade off performance for generality, and suffer significant overheads while proving large models such as LLMs.

Large language model inference. With the advent of large language models, a smaller number of works have proposed techniques to prove the correctness of LLM inference [CWSK24, SLZ24, CTK25, QSL+25]. ZKML encodes a (slightly modified) version of GPT-2 in an arithmetic circuit and uses Halo2 [Zca20] to prove the correctness of inference. Unfortunately, this "generic" approach leads in reduced efficiency and massive architecture requirements when applied to LLMs. zkPyTorch [XLF+25] takes a similar approach, but relies on GKR [GKR08]. Subsequent works such as zkLLM [SLZ24], zkGPT [QSL+25], and zkTorch [CTK25], are faster than ZKML [CWSK24] by one to two orders of magnitude.

zkLLM [SLZ24] was the first work to efficiently prove LLM components such as attention and softmax using specialized sum-check

	GPT-2	Gemma 3
Layer	BaseFold	BaseFold
Prover split-up (%)		
Witness gen.	5.97	5.51
Witness commit	56.85	57.73
Open	1.04	1.11
PIOP	35.25	34.71
Others	0.89	0.94
Total	100	
Witness commit (%)		
Activation	22.96	10.21
Logits	9.19	18.11
Requant	27.49	41.60
RMS Norm	10.59	21.86
Softmax	29.68	8.13
Multiplicity	0.08	0.04
Others	0.01	0.05
Total	100	
PIOP split-up (%)		
Activation	13.96	6.08
Add	1.03	0.83
Attention Mask	3.96	0.97
Einsum	14.91	15.46
Embeddings	2.69	5.78
Logits	7.83	16.10
Positional	0.06	0.83
Requant	27.30	29.96
RMS Norm	6.02	15.24
Softmax	19.39	5.40
Tables	0.08	0.03
Aggregation	1.79	1.95
Others	0.98	1.37
Total	100	

Table 7: Layer-wise breakdown of witness commitment and PIOP time (%) for sequence length 512 and 48 threads.

protocols. However, that work only focuses on proving these specific layers and their proof-of-concept implementation returns individual proofs for each layer without asserting the correctness of tensors passed between them. Conceptually, we extend their approach for proving softmax correctness to the batch setting, and our system is end-to-end.

zkTorch [CTK25] aims for a middle ground between generality and specialization by decomposing LLMs into a set of operators, similar to what we do. However, the authors focus on generating proofs of correctness for one output token at a time. Due to the generality of their approach, as remarked in Section 5, the performance of zkTorch is generally lower than `DeepProve`. Moreover, their reported experimental runs on a machine with 64 cores and 4 TB RAM. A final drawback of that approach is that the use of scaling factors is limited to powers-of-2, which incurs greater precision loss than the flexible quantization approach adopted in `DeepProve`.

zkGPT [QSL+25] is currently the state-of-the-art system for proving LLM inference. However, like the above works, that system is only designed to prove a single output token. Moreover, the authors prove the correctness of a modified version of GPT-2 that doesn't contain residual connections, attention masks, and the final projection layer. These relaxations significantly circumvent the outlier issues that we discussed in Section 1. Similar to zkTorch [CTK25], zkGPT uses fixed-point multiplication with scaling factors for quantization. While this technique is slightly simpler to prove, it comes with a trade-off: The major downside is that since re-quantization and non-linear functions are usually proven via lookup arguments, using a fixed scaling factor without clamping can lead to situations where the values generated during inference are too large to be looked up, thus forcing a larger lookup table to commit.

Finally, we stress that ours is the only work to provide accuracy measurements for results achieved with the exact codebase of our LLM inference prover. To the best of our knowledge, prior works did not integrate their accuracy experiments with their proving codebase. As shown in Appendix C, many subtle nuances arise when trying to prove end-to-end LLM traces (such as requantization, lookup table size selection, clamping, etc.). Hence, it is not entirely clear whether prior works [QSL+25, SLZ24, XLF+25] would achieve their reported accuracy results if they were proving a full LLM inference trace.

References

[AKKP25] Deema Alnuhait, Neeraja Kirtane, Muhammad Khalifa, and Hao Peng. Factcheckmate: Preemptively detecting and mitigating hallucinations in lms, 2025.

[AMC+24] Saleh Ashkboos, Amirkeivan Mohtashami, Maximilian L. Croci, Bo Li, Pashmina Cameron, Martin Jaggi, Dan Alistarh, Torsten Hoefler, and James Hensman. Quarot: Outlier-free 4-bit inference in rotated llms, 2024.

[APKP24] Kasra Abbaszadeh, Christodoulos Pappas, Jonathan Katz, and Dimitrios Papadopoulos. Zero-Knowledge Proofs of Training for Deep Neural Networks. In *ACM CCS*, 2024.

[AST24] Arasu Arun, Srinath Setty, and Justin Thaler. Jolt: SNARKs for Virtual Machines via Lookups. In *EUROCRYPT*, 2024.

[BCS16] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. Interactive oracle proofs. In *TCC*, pages 31–60, 2016.

[BFGV+23] David Balbás, Dario Fiore, Maria Isabel González Vasco, Damien Robissout, and Claudio Soriente. Modular Sumcheck Proofs with Applications to Machine Learning and Image Processing. In *ACM CCS*, 2023.

[BMM+24] Anubhav Baweja, Pratyush Mishra, Tushar Mopuri, Karan Newatia, and Steve Wang. Scribe: Low-memory snarks via read-write streaming. *IACR Cryptol. ePrint Arch.*, 2024.

[BSCTV14] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture. In *USENIX Security*, 2014.

[CBBZ23] Binyi Chen, Benedikt Bünz, Dan Boneh, and Zhenfei Zhang. HyperPlonk: Plonk with Linear-Time Prover and High-Degree Custom Gates. In *EUROCRYPT*, 2023.

[CCC+25] Matteo Campanelli, Mario Carrillo, Ignacio Cascudo, Dario Fiore, Danilo Francati, and Rosario Gennaro. On the Power of Polynomial Preprocessing: Proving Computations in Sublinear Time, and More. Cryptology ePrint Archive, 2025.

[CCH+19] Ran Canetti, Yilei Chen, Justin Holmgren, Alex Lombardi, Guy N. Rothblum, Ron D. Rothblum, and Daniel Wichs. Fiat-Shamir: from practice to theory. In *STOC*, 2019.

[CCKDS23] Jerry Chee, Yaohui Cai, Volodymyr Kuleshov, and Christopher De Sa. Quip: 2-bit quantization of large language models with guarantees. NIPS '23, 2023.

[CCYY25] Mingshu Cong, Sherman S. M. Chow, Siu Ming Yiu, and Tsz Hon Yuen. Scalable zkSNARKs for Matrix Computations: A Generic Framework for Verifiable Deep Learning. Cryptology ePrint Archive, 2025.

[CFS17] Alessandro Chiesa, Michael A. Forbes, and Nicholas Spooner. A zero knowledge sumcheck and its applications. *CoRR*, abs/1704.02086, 2017.

[CHM+20] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS. In *EUROCRYPT*, 2020.

[CLC+24] Chao Chen, Kai Liu, Ze Chen, Yi Gu, Yue Wu, Mingyuan Tao, Zhihang Fu, and Jieping Ye. INSIDE: LLMs' internal states retain the power of hallucination detection. In *ICLR*, 2024.

[CMT12] Graham Cormode, Michael Mitzenmacher, and Justin Thaler. Practical verified computation with streaming interactive proofs. ITCS, 2012.

[cod] Deepprove implementation. https://anonymous.4open.science/r/deep-prove-private-C86D/README.md.

[Com23] OpenAI Community. Openai did made gpt-3.5 more stupid, 2023.

[Com24] OpenAI Community. Gpt-4 turbo more stupid/lazy? it's not a gpt-4, 2024. OpenAI Community Forum.

[CTK25] Bing-Jyue Chen, Lilia Tang, and Daniel Kang. ZKTorch: Compiling ML Inference to Zero-Knowledge Proofs via Parallel Proof Accumulation, 2025.

[CWSK24] Bing-Jyue Chen, Suppakit Waiwitlikhit, Ion Stoica, and Daniel Kang. ZKML: An Optimizing System for ML Inference in Zero-Knowledge Proofs. EuroSys '24, 2024.

[dev19] ONNX Runtime developers. Onnx: Open neural network exchange, 2019. Accessed: 2025-11-07.

[ea21] Mark Chen et. al. Evaluating Large Language Models Trained on Code, 2021.

[ea23] Köpf Andreas et. al. Openassistant conversations - democratizing large language model alignment. In *NeurIPS*. Curran Associates Inc., 2023.

[ea25a] Dmytro Zakharov et. al. Bionetta: Efficient Client-Side Zero-Knowledge Machine Learning Proving, 2025.

[ea25b] Huang Lei et. al. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *TOIS*, 43:1–55, 2025.

[EZC+25] Jens Ernstberger, Chengru Zhang, Luca Ciprian, Philipp Jovanovic, and Sebastian Steinhorst. Zero-Knowledge Location Privacy via Accurate Floating-Point SNARKs. In *IEEE SP*, 2025.

[Fac25a] Hugging Face. Gpt-generated unified format (gguf), 2025. Accessed: 2025-11-07.

[Fac25b] Hugging Face. Safetensors documentation, 2025. Accessed: 2025-12-07.

[FKKG24] Sebastian Farquhar, Jannik Kossen, Lorenz Kuhn, and Yarin Gal. Detecting hallucinations in large language models using semantic entropy. *Nature*, pages 625–630, 2024.

[For25] PyTorch Discussion Forum. Is bias quantized while doing pytorch static quantization?, 2025. Accessed: 2025-12-07.

[FQZ+21] Boyuan Feng, Lianke Qin, Zhenfei Zhang, Yufei Ding, and Shumo Chu. ZEN: An optimizing compiler for verifiable, zero-knowledge neural network inferences. Cryptology ePrint Archive, 2021.

[GCX+25] Cheng Gao, Huimin Chen, Chaojun Xiao, Zhiyi Chen, Zhiyuan Liu, and Maosong Sun. H-neurons: On the existence, impact, and origin of hallucination-associated neurons in llms, 2025.

[GGJ+23] Sanjam Garg, Aarushi Goel, Somesh Jha, Saeed Mahloujifar, Mohammad Mahmoody, Guru-Vamsi Policharla, and Mingyuan Wang. Experimenting with Zero-Knowledge Proofs of Training. In *ACM CCS*, 2023.

[GGPR13] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic Span Programs and Succinct NIZKs without PCPs. In *EUROCRYPT*, 2013.

[Git26] GitHub Copilot. https://docs.github.com/en/copilot/concepts/agents/about-copilot-cli#security-considerations, 2026.

[GJJZ22] Sanjam Garg, Abhishek Jain, Zhengzhong Jin, and Yinuo Zhang. Succinct Zero Knowledge for Floating Point Computations. In *ACM CCS*, 2022.

[GKR08] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: interactive proofs for muggles. In *STOC*, 2008.

[GSNPB17] Claire Gardent, Anastasia Shimorina, Shashi Narayan, and Laura Perez-Beltrachini. The WebNLG challenge: Generating text from RDF data. In Jose M. Alonso, Alberto Bugarín, and Ehud Reiter, editors, *Proceedings of the 10th International Conference on Natural Language Generation*, pages 124–133. Association for Computational Linguistics, September 2017.

[Hab22] Ulrich Haböck. Multivariate lookups based on logarithmic derivatives. Cryptology ePrint Archive, 2022.

[HCL+24] Meng Hao, Hanxiao Chen, Hongwei Li, Chenkai Weng, Yuan Zhang, Haomiao Yang, and Tianwei Zhang. Scalable Zero-knowledge Proofs for Non-linear Functions in Machine Learning. In *USENIX Security*, 2024.

[HLP24] Ulrich Haböck, David Levit, and Shahar Papini. Circle STARKs. Cryptology ePrint Archive, 2024.

[HLSS24] Intak Hwang, Hyeonbum Lee, Jinyeong Seo, and Yongsoo Song. Practical Zero-Knowledge PIOP for Maliciously Secure Multiparty Homomorphic Encryption. Cryptology ePrint Archive, 2024.

[JKC+18] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew G. Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In *CVPR*, 2018.

[JLF+23] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. Survey of hallucination in natural language generation. *ACM computing surveys*, pages 1–38, 2023.

[KHSS22] Daniel Kang, Tatsunori Hashimoto, Ion Stoica, and Yi Sun. Scaling up Trustless DNN Inference with Zero-Knowledge Proofs, 2022.

[KPPS20] Ahmed E. Kosba, Dimitrios Papadopoulos, Charalampos Papamanthou, and Dawn Song. MIRAGE: succinct arguments for randomized algorithms with applications to universal zk-snarks. In Srdjan Capkun and Franziska Roesner, editors, *USENIX Security*, pages 2129–2146, 2020.

[LFKN90] Carsten Lund, Lance Fortnow, Howard J. Karloff, and Noam Nisan. Algebraic Methods for Interactive Proof Systems. In *FOCS*, 1990.

[LKKO24] Seunghwa Lee, Hankyung Ko, Jihye Kim, and Hyunok Oh. vCNN: Verifiable Convolutional Neural Network Based on zk-SNARKs. In *IEEE Transactions on Dependable and Secure Computing*, 2024.

[Loc24] Lockheed Martin. Lockheed martin, meta collaborate on large language models for national security applications, 2024.

[LWQ+24] Tao Lu, Haoyu Wang, Wenjie Qu, Zonghui Wang, Jinye He, Tianyang Tao, Wenzhi Chen, and Jiaheng Zhang. An Efficient and Extensible Zero-knowledge Proof Framework for Neural Networks. Cryptology ePrint Archive, 2024.

[LXW+24] Haokun Lin, Haobo Xu, Yichen Wu, Jingzhi Cui, Yingtao Zhang, Linzhan Mou, Linqi Song, Zhenan Sun, and Ying Wei. Duquant: Distributing outliers via dual transformation makes stronger quantized llms, 2024.

[LXZ21] Tianyi Liu, Xiang Xie, and Yupeng Zhang. zkCNN: Zero Knowledge Proofs for Convolutional Neural Network Predictions and Accuracy. In *ACM CCS*, 2021.

[LXZ+24] Tianyi Liu, Tiancheng Xie, Jiaheng Zhang, Dawn Song, and Yupeng Zhang. Pianist: Scalable zkRollups via Fully Distributed Zero-Knowledge Proofs. In *IEEE SP*, 2024.

[LYZ24] Jiarui Li, Ye Yuan, and Zehua Zhang. Enhancing llm factual accuracy with rag to counter hallucinations: A case study on domain-specific queries in private knowledge-bases, 2024.

[MAB+24] Abhika Mishra, Akari Asai, Vidhisha Balachandran, Yizhong Wang, Graham Neubig, Yulia Tsvetkov, and Hannaneh Hajishirzi. Fine-grained hallucination detection and editing for language models. In *First Conference on Language Modeling*, 2024.

[MLG23] Potsawee Manakul, Adian Liusie, and Mark Gales. Selfcheckgpt: Zero-resource black-box hallucination detection for generative large language models. In *Conference on empirical methods in natural language processing*, pages 9004–9017, 2023.

[MMN+24] Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, and Jianfeng Gao. Large Language Models: A Survey, 2024.

[MS25] Subhankar Maity and Manob Jyoti Saikia. Large language models in healthcare and medical applications: A review. *Bioengineering*, 2025.

[MY25] Hideaki Miyaji and Hiroshi Yamamoto. Proof of Learning Using Zero-Knowledge Proof. In *ICCE*, 2025.

[NWZ+24] Cheng Niu, Yuanhao Wu, Juno Zhu, Siliang Xu, Kashun Shum, Randy Zhong, Juntong Song, and Tong Zhang. Ragtruth: A hallucination corpus for developing trustworthy retrieval-augmented language models. In *Annual Meeting of the Association for Computational Linguistics*, pages 10862–10878, 2024.

[Ope25] OpenAI. Chatgpt, 2025. Large language model, GPT-4.1.

[OTG+25] Hadas Orgad, Michael Toker, Zorik Gekhman, Roi Reichart, Idan Szpektor, Hadas Kotek, and Yonatan Belinkov. LLMs know more than they show: On the intrinsic representation of LLM hallucinations. In *ICLR*, 2025.

[PGL+26] Qi Pang, Radhika Garg, Ziling Liu, Hanshen Xiao, Virginia Smith, Wenting Zheng, and Xiao Wang. Noisette: Certifying differential privacy mechanisms efficiently. *IACR Cryptol. ePrint Arch.*, 2026.

[PH23] Shahar Papini and Ulrich Haböck. Improving logarithmic derivative lookups using GKR. Cryptology ePrint Archive, 2023.

[PP24] Christodoulos Pappas and Dimitrios Papadopoulos. Sparrow: Space-Efficient zkSNARK for Data-Parallel Circuits and Applications to Zero-Knowledge Decision Trees. In *ACM CCS*, 2024.

[PP25] Christodoulos Pappas and Dimitrios Papadopoulos. Hobbit: Space-Efficient zkSNARK with Optimal Prover Time. In *Proceedings of the 34th USENIX Conference on Security Symposium*, USENIX Security '25, 2025.

[QSL+25] Wenjie Qu, Yijun Sun, Xuanming Liu, Tao Lu, Yanpei Guo, Kai Chen, and Jiaheng Zhang. zkgpt: An efficient non-interactive zero-knowledge proof framework for LLM inference. In Lujo Bauer and Giancarlo Pellegrino, editors, *USENIX Security*, pages 2045–2063, 2025.

[RSZ25] Rahul Raman, Khushi Sharma, and Sai Qian Zhang. Rethinking the Outlier Distribution in Large Language Models: An In-depth Study, 2025.

[RTB+23] Nazneen Rajani, Lewis Tunstall, Edward Beeching, Nathan Lambert, Alexander M. Rush, and Thomas Wolf. No robots. https://huggingface.co/datasets/HuggingFaceH4/no_robots, 2023.

[RWB+25] Arman Riasi, Haodi Wang, Rouzbeh Behnia, Viet Vo, and Thang Hoang. Zero-Knowledge AI Inference with High Precision. In *ACM CCS*, 2025.

[RWC+19] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 2019.

[Sal23] Alex Salt. Openai keeps dumbing down chatgpt, 2023.

[SBS+24] Gaurang Sriramanan, Siddhant Bharti, Vinu Sankar Sadasivan, Shoumik Saha, Priyatham Kattakinda, and Soheil Feizi. Llm-check: Investigating detection of hallucinations in large language models. *NeurIPS*, pages 34188–34216, 2024.

[Scr24] Scroll. CENO: Non-uniform, Segment and Parallel Zero-knowledge Virtual Machine, 2024. GitHub repository.

[SLP+21] Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. RoFormer: Enhanced Transformer with Rotary Position Embedding, 2021.

[SLZ24] Haochen Sun, Jason Li, and Hongyang Zhang. zkLLM: Zero Knowledge Proofs for Large Language Models. In *ACM CCS*, pages 4405–4419, 2024.

[SSPP25] Sriram Sridhar, Shravan Srinivasan, Dimitrios Papadopoulos, and Charalampos Papamanthou. Efficiently Provable Approximations for Non-Polynomial Functions. Cryptology ePrint Archive, 2025.

[SSZ21] Suraj Subramanian, Mark Saroufim, and Jerry Zhang. Quantization in practice. https://pytorch.org/blog/quantization-in-practice/, 2021. Accessed: January 14, 2026.

[STW23] Srinath Setty, Justin Thaler, and Riad Wahby. Unlocking the lookup singularity with Lasso. Cryptology ePrint Archive, 2023.

[Tea] Polygon Zero Team. Plonky 2: Improved Plonk with Fast Verification and Universal SRS.

[Tea25] Gemma Team. Gemma 3 Technical Report, 2025.

[TGM+25] Gefei Tan, Adria Gascon, Sarah Meiklejohn, Mariana Raykova, Xiao Wang, and Ning Luo. Founding Zero-Knowledge Proof of Training on Optimum Vicinity. In *ACM CCS*, 2025.

[Tha13] Justin Thaler. Time-Optimal Interactive Proofs for Circuit Evaluation. In *CRYPTO*, 2013.

[Tha25] Justin Thaler. Sum-check Is All You Need: An Opinionated Survey on Fast Provers in SNARK Design. Cryptology ePrint Archive, 2025.

[vBBWN25] Boris van Breugel, Yelysei Bondarenko, Paul Whatmough, and Markus Nagel. Fptquant: Function-preserving transforms for llm quantization, 2025.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is All you Need. In *NeurIPS*, 2017.

[Wik25a] Wikipedia contributors. Diella (ai system) — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Diella_(AI_system)&oldid=1328050476, 2025. [Online; accessed 12-January-2026].

[Wik25b] Wikipedia contributors. Transcendental function — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Transcendental_function&oldid=1327837638, 2025.

[Wik25c] Wikipedia contributors. Transformer (deep learning) — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Transformer_(deep_learning)&oldid=1330245756#KV_caching, 2025.

[WYX+21] Chenkai Weng, Kang Yang, Xiang Xie, Jonathan Katz, and Xiao Wang. Mystique: Efficient Conversions for Zero-Knowledge Proofs with Applications to Machine Learning. In *USENIX Security*, 2021.
[XLF+25] Tiancheng Xie, Tao Lu, Zhiyong Fang, Siqi Wang, Zhenfei Zhang, Yongzheng Jia, Dawn Song, and Jiaheng Zhang. zkPyTorch: A hierarchical optimized compiler for zero-knowledge machine learning. Cryptology ePrint Archive, 2025.
[XZC+22] Tiancheng Xie, Jiaheng Zhang, Zerui Cheng, Fan Zhang, Yupeng Zhang, Yongzheng Jia, Dan Boneh, and Dawn Song. zkBridge: Trustless Cross-chain Bridges Made Practical. In *ACM CCS*, 2022.
[YJK+22] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *OSDI*, 2022.
[Zca20] Zcash Foundation. The halo2 book. Technical report, 2020.
[ZCF24] Hadas Zeilberger, Binyi Chen, and Ben Fisch. BaseFold: Efficient Field-Agnostic Polynomial Commitment Schemes from Foldable Codes. In *CRYPTO*, 2024.
[ZFZS20] Jiaheng Zhang, Zhiyong Fang, Yupeng Zhang, and Dawn Song. Zero Knowledge Proofs for Decision Tree Predictions and Accuracy. In *ACM CCS*, 2020.
[zko24] zkonduit. EZKL: Easy zero knowledge library. https://github.com/zkonduit/ezkl, 2024.
[ZPM+24] Muru Zhang, Ofir Press, William Merrill, Alisa Liu, and Noah A. Smith. How language model hallucinations can snowball. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235, pages 59670–59684, 2024.
[ZSCZ25] Jiaxing Zhao, Srinath Setty, Weidong Cui, and Greg Zaverucha. MicroNova: Folding-Based Arguments with Efficient (On-Chain) Verification. In *IEEE SP*, pages 1964–1982, 2025.

A Additional Preliminaries

A.1 Sumcheck Protocol

Definition 1 (Generalized Sumcheck). *The relation $\mathcal{R}_{Sum}$ is the set of all tuples*

$$(\mathbb{x};\mathbb{w}) = ((K, \{[[f]]_i\}_{i\in[m]}); \{f_i\}_{i\in[m]} : K = \sum_{\boldsymbol{x}\in\{0,1\}^{\log n}} \prod_{i\in[m]} f_i(\boldsymbol{x})),$$

where $f_i \in \mathbb{F}_{<d}[X_1,\ldots,X_n]$ is a $\log n$-variate polynomial of degree strictly less than d in each variable.

The sumcheck protocol, introduced by Lund *et al.*. [LFKN90] proceeds as follows:

Protocol 1: Sumcheck protocol

(1) $\mathcal{V}$: Set $K_1 \leftarrow K$.
(2) For $i = 1,\ldots,\log n$ do:
  (a) $\mathcal{P}$: Sends:
  $$h_i(x) \leftarrow \sum_{\mathbf{z}\in\{0,1\}^{\log n-i}} \prod_{j\in[m]} f_i(r_1,\ldots,r_{i-1},x,\mathbf{z})$$
  to $\mathcal{V}$.
  (b) $\mathcal{V}$: checks if $h_i(0)+h_i(1)=K_i$. Picks a random point $r_i \in \mathbf{F}$, sets $K_{i+1} \leftarrow h_i(r_i)$, and sends r_i to $\mathcal{P}$.
(3) $\mathcal{P}$: Sends $\{y_j = f_j(r_1,\ldots,r_{\log n})\}_{j\in[m]}$ to $\mathcal{V}$.
(4) $\mathcal{V}$: Checks if $K_{\log n+1} = \prod_{j\in[m]} y_j$ and queries $[[f_j]]$ to validate the correctness of y_j for all $j \in [m]$.

Theorem 1. *The sumcheck protocol for as described above has perfect completeness and (knowledge) round-by-round soundness of $\kappa_{rbr} = \epsilon_{rbr} = \frac{m\cdot d}{|\mathbb{F}|}$. Furthermore, the total prover time is $O(m\log^2 m\cdot n)$ and proof size and verification time of $O(md\log n)$.*

A.2 Batched Sumcheck Protocol

In this work, we will heavily use the batched sumcheck protocol that enables a prover to prove multiple sumcheck instances "in parallel". Here we will use the batched sumcheck protocol proposed in [CFS17]. More formally, we first define the following relation:

Definition 2 (Generalized Sumcheck). *The relation $\mathcal{R}^{Batch}_{Sum}$ is the set of all tuples*

$$(\mathbb{x};\mathbb{w}) = \left(\begin{array}{c} (\{K^{(j)}\}, \{[[f]]_i^{(j)}\}_{i,j\in[m]\times[h]}); \{f_i^{(j)}\}_{i,j\in[m]\times[h]} : \\ K^{(j)} = \sum_{\boldsymbol{x}\in\{0,1\}^{\log n}} \prod_{i\in[m]} f_i^{(j)}(\boldsymbol{x}), \forall j \in [h] \end{array} \right)$$

where $f_i^{(j)} \in \mathbb{F}_{<d}[X_1,\ldots,X_n]$ is a $\log n$-variate polynomial of degree strictly less than d in each variable.

To prove the above relation we use the following protocol:

Protocol 2: Batched Sumcheck protocol

(1) $\mathcal{V}$: Sample a random point $r \in \mathbb{F}$ and send it to $\mathcal{P}$. Furthermore, compute $K_1 \leftarrow \sum_{i\in[h]} r^i K^{(i)}$.
(2) For $i = 1,\ldots,\log n$ do:
  (a) $\mathcal{P}$: Computes:
  $$h_i^{(j)}(x) \leftarrow \sum_{\mathbf{z}\in\{0,1\}^{\log n-i}} \prod_{j\in[m]} f_i^{(j)}(r_1,\ldots,r_{i-1},x,\mathbf{z})$$
  for all $j \in [h]$ and sends $h_i(x) \leftarrow \sum_{j\in[h]} r^j h_i^{(j)}(x)$ to $\mathcal{V}$.
  (b) $\mathcal{V}$: checks if $h_i(0)+h_i(1)=K_i$. Picks a random point $r_i \in \mathbf{F}$, sets $K_{i+1} \leftarrow h_i(r_i)$, and sends r_i to $\mathcal{P}$.
(3) $\mathcal{P}$: Sends $\{y_i^{(j)} = f_i^{(j)}(r_1,\ldots,r_{\log n})\}_{i\in[m]}$ to $\mathcal{V}$.
(4) $\mathcal{V}$: Checks if $K_{\log n+1} = \sum_{j\in[h]} r^j \prod_{i\in[m]} y_i^{(j)}$ and queries $[[f_i^{(j)}]]$ to validate the correctness of $y_i^{(j)}$ for all $i, j \in [m]\times[h]$.

Theorem 2. *The sumcheck protocol for as described above has perfect completeness and (knowledge) round-by-round soundness of $\kappa_{rbr} = \epsilon_{rbr} = \max(\frac{h}{|\mathbb{F}|}, \frac{md}{|\mathbb{F}|})$. Furthermore, the total prover time is $O(hm\log^2 m\cdot n)$ and proof size and verification time of $O(h + md\log n)$.*

Proof of Round-by-Round Soundness. Initially, observe that $\mathbb{x} \notin \mathcal{R}^{Batch}_{Sum}$ if there exists $i \in [h]$ s.t $K^{(i)} \neq \sum_{\mathbf{x}\in\{0,1\}^{\log n}} \prod_{j\in[m]} f_j^{(i)}(\mathbf{x})$. In this case, we say that the transcript $(\mathbb{x})$ belongs to a doomed state. Now, assume that $(\mathbb{x})$ is in doomed state, we will show that $\Pr[(\mathbb{x}) \in \mathcal{D} \wedge (\mathbb{x}, r, K_1) \notin \mathcal{D}] \leq \frac{h}{|\mathbb{F}|}$. Note that $(\mathbb{x}, r, K_1) \notin \mathcal{D}$ if and only if for a random r it holds that $K_1 = \sum_{i\in[h]} r^i K'_i$, where K'_i the actual sum. This, however, happens only with probability $\frac{h}{|\mathbb{F}|}$ due to the Schwartz-Zippel Lemma. Now, assume that in the i-th round of the protocol, the transcript $\mathbf{tr}_i \in \mathcal{D}$. We will show that conditioned that $\mathbf{tr}_i \in \mathcal{D}$, $\Pr[(\mathbf{tr}_i, h_{i+1}, r_{i+1}) \notin \mathcal{D}] \leq \frac{md}{|\mathbb{F}|}$. Note that since $\mathbf{tr}_i \in \mathcal{D}$, then it must hold that $K_i = h_i(r_i) \neq \sum_{j\in[h]} r^j \sum_{\mathbf{z}\in\{0,1\}^{\log n-i}} \prod_{k\in[m]} f_k^{(j)}(r_1,\ldots,r_i,\mathbf{z}) \neq h'_{i+1}(0)+h'_{i+1}(1)$. At this point we consider two cases: (1) $h_{i+1}(x) = h'_{i+1}(x)$ where in this case the verifier would always reject and thus the prover would stay in the doomed state with probability 1 and (2) $h_{i+1}(x) \neq h'_{i+1}(x)$. In the last case, the only way the prover can leave the

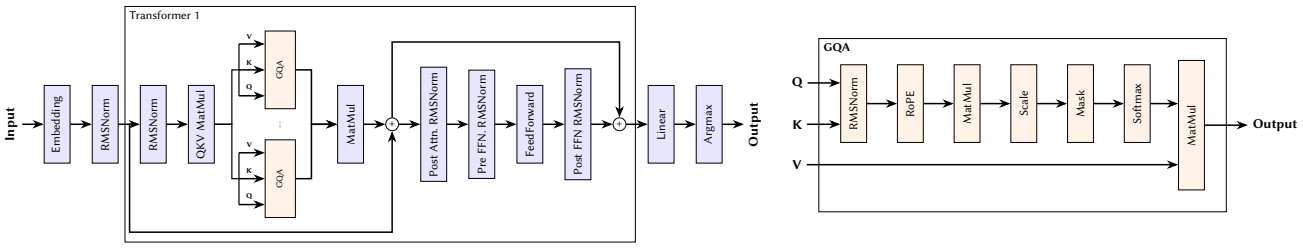

Figure 4: Gemma 3 architecture. For simplicity, we illustrate only a single transformer block. The subfigure details the operations of the grouped query attention mechanism.

doomed state is when $h_{i+1}(r_{i+1}) = h'_{i+1}(r_{i+1})$ which only happens with probability $\frac{md}{|\mathbb{F}|}$ due to the Schwartz-Zippel Lemma. Finally, assume that $\mathbf{tr}_{\log n+1} \in \mathcal{D}$. We will show that the probability that the final transcript $(\mathbf{tr}_{\log n+1}, \{y_i^{(j)}\}_{i,j\in[m]\times[h]})$ is doomed. To prove this, observe that since $\mathbf{tr}_{\log n+1} \in \mathcal{D}$, it must hold that $K_{\log n+1} \neq \sum_{j\in[h]} r^j \sum_{\mathbf{z}\in\{0,1\}^{\log n-i}} \prod_{k\in[m]} f_k^{(j)}(r_1, \ldots, r_{\log n})$. Hence, the prover either selects evaluation claims that agree with $K_{\log n+1}$ but disagree with at least one $f_i^{(j)}(r_1, \ldots, r_{\log n})$ or vice-versa. In any case, the verifier would reject with probability 1 and thus the prover would remain in a doomed state with probability 1. Consequently, the round-by-round soundness will be $\epsilon_{rbr} = \max(\frac{h}{|\mathbb{F}|}, \frac{md}{|\mathbb{F}|})$.

Proof of Round-by-Round Knowledge Soundness. Suppose that $\mathbf{tr}_0 = (\mathbb{x}) \in \mathcal{D}$ but $\Pr[(\mathbf{tr}_{\log n+1}, \{y_i^{(j)}\}_{i,j\in[m]\times[h]}) \notin \mathcal{D}] > \epsilon_{rbr}$. Then, by the above observation, with probability 1 it must hold $\Pr[\mathbf{tr}_{\log n+1} \notin \mathcal{D}$—since otherwise the verifier would reject which contradicts our initial assumption. Next, let $\Pr[(\mathbf{tr}_{\log n}, h_{\log n}(x), r_i) \notin \mathcal{D}] > \epsilon_{rbr}$ and assume that $\mathbf{tr}_{\log n} \in \mathcal{D}$. However, this is not possible since the last one can happen only with probability ϵ_{rbr}. Hence, $\mathbf{tr}_{\log n} \notin \mathcal{D}$. We follow the same argument up until the last step. Here, assume that $\Pr[(\mathbf{tr}_0, r, K_1) \notin \mathcal{D}] > \epsilon_{rbr}$. Then, it must hold that $(\mathbb{x}) = (\mathbf{tr}_0) \notin \mathcal{D}$. That is because if $(\mathbf{tr}_0) \in \mathcal{D}$ then $\Pr[(\mathbf{tr}_0, r, K_1) \notin \mathcal{D}] \leq h/|\mathbb{F}| < \epsilon_{rbr}$ thus reaching to a contradiction. Consequently, if $\Pr[(\mathbf{tr}_{\log n+1}, \{y_i^{(j)}\}_{i,j\in[m]\times[h]}) \notin \mathcal{D}] > \epsilon_{rbr}$, then $(\mathbb{x}) \in \mathcal{R}^{Batch}_{Sum}$.

A.3 Lookup Arguments

A lookup argument [STW23] enables a prover to efficiently access elements of a committed pre-defined table $\mathbf{T} \in \mathbb{F}^t$. For the needs of our protocol we use the following relation:

Definition 3 (Lookup argument). *The* lookup relation $\mathcal{R}_{lkp}$ *is defined as follows:*

$$(\mathbb{x}; \mathbb{w}) = \left(\begin{array}{c} (\llbracket f_{\boldsymbol{T}} \rrbracket, \llbracket f_{\boldsymbol{x}} \rrbracket, \llbracket f_{\boldsymbol{y}} \rrbracket); (\mathbf{T}, \mathbf{x}, \mathbf{y}) : \\ \boldsymbol{y}[i] = \boldsymbol{T}[\boldsymbol{x}[i]], \forall i \in [n] \end{array} \right)$$

where:

- $\boldsymbol{T} \in \mathbb{F}^t$ *the public table,* $\boldsymbol{x} \in \mathbb{F}^n$ *the committed indexes and* $\boldsymbol{y} \in \mathbb{F}^n$ *the restriction of* $\boldsymbol{T}$ *at* $\boldsymbol{x}$*.*
- $f_{\boldsymbol{T}}, f_{\boldsymbol{x}}$ *and* $f_{\boldsymbol{y}}$ *the MLEs of* $\boldsymbol{T}$*,* $\boldsymbol{x}$ *and* $\boldsymbol{y}$ *respectively.*

In practice, $\mathbf{T}$ might be exponentially large but decomposable, in the sense that we can segment it into smaller tables, perform lookups over these tables, and then recombine their output to compute the final result. This property was first explored by Jolt and Lasso [STW23, AST24] and we will also use it in order to prove the correct computation of activation functions. To encompass this property, we revise the above relation as follows:

Definition 4 (Lookup Argument with Decomposable Tables). *The* lookup relation $\mathcal{R}_{lkp}$ *over decomposable tables is defined as follows:*

$$(\mathbb{x}; \mathbb{w}) = \left(\begin{array}{c} (\{\llbracket f_{\boldsymbol{T}_i} \rrbracket\}_{i\in[m]}, \llbracket f_{\boldsymbol{x}} \rrbracket, \{\llbracket f_{\boldsymbol{x}_i} \rrbracket\}_{i\in[m]}, \llbracket f_{\boldsymbol{y}} \rrbracket, \{\llbracket f_{\boldsymbol{y}_i} \rrbracket\}_{i\in[m]}); \\ (\{\mathbf{T}_i\}_{i\in[m]}, \mathbf{x}, \{\mathbf{x}^{(i)}\}_{i\in[m]}, \mathbf{y}, \{\mathbf{y}^{(i)}\}_{i\in[m]}) : \\ \boldsymbol{y}^{(i)}[j] = \boldsymbol{T}_i[\boldsymbol{x}^{(i)}[j]], \forall j, i \in [n] \times [m] \wedge \\ \boldsymbol{y} = Comb_y(\boldsymbol{y}^{(1)}, \ldots, \boldsymbol{y}^{(m)}) \wedge \boldsymbol{x} = Comb_x(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}) \end{array} \right)$$

where:

- $\{\boldsymbol{T}_i\}_{i\in[m]}, \{\boldsymbol{x}^{(i)}\}_{i\in[m]}$ *and* $\{\boldsymbol{y}^{(i)}\}_{i\in[m]}$ *decompositions of the original table* $\boldsymbol{T}$*, index* $\boldsymbol{x}$ *and looked-up output* $\boldsymbol{y}$*.*
- $Comp_y$ *and* $Comp_x$ *are deterministic functions that take as input the decomposed vectors and output the original ones.*

In this work, we use the lookup PIOP proposed by [PH23] which first converts $\mathcal{R}_{lkp}$ to a LogUp equation [Hab22] and subsequently uses the GKR protocol [GKR08] to prove its correct computation. For the rest of this section, we will assume that the underlying PIOP ends up with evaluation claims of the MLEs of $\mathbf{y}$ and $\mathbf{x}$. Furthermore, we will assume that $m \in \mathbb{N}$ is a small constant which is true for our protocol. With that in mind, we have the following theorem:

Theorem 3. *There exists a PIOP for* $\mathcal{R}_{lkp}$ *with round-by-round (knowledge) soundness of* $\epsilon_{rbr} = \kappa_{rbr} = O(\frac{\log n}{|\mathbb{F}|})$*. Furthermore, it has prover complexity of* $O(n + t)$ *proof size and verification time of* $O(\log^2 n + \log^2 t)$*.*

A.3.1 Batched Lookup Argument. In our protocol, the prover has to simultaneously prove that for many instances $\mathbb{x}_1, \ldots, \mathbb{x}_h, \mathbb{x}_i \in \mathcal{R}_{lkp}$. More formally:

Theorem 4 (Batched Lookup Argument). *The relation of the batched lookup argument* $\mathcal{R}^{Batch}_{lkp}$ *is defined as follows:*

$$(\mathbb{x}; \mathbb{w}) = \left(\begin{array}{c} (\llbracket f_{\boldsymbol{T}} \rrbracket, \{\llbracket f_{\boldsymbol{x}_i} \rrbracket\}_{i\in[h]}, \{\llbracket f_{\boldsymbol{y}} \rrbracket\}_{i\in[h]}); (\mathbf{T}, \{\mathbf{x}_i\}_{i\in[h]}, \{\mathbf{y}_i\}_{i\in[h]}) : \\ \boldsymbol{y}_i[j] = \boldsymbol{T}[\boldsymbol{x}_i[j]], \forall i, j \in [h] \times [n] \end{array} \right)$$

To construct a PIOP for the above relation, we use a PIOP that is almost identical to the lookup PIOP of [PH23] but instead of invoking the standard sumcheck (used by the GKR protocol), we use the batched sumcheck protocol as explained in the previous section.

Theorem 5. *There exists a PIOP for $\mathcal{R}_{lkp}$ with round-by-round (knowledge) soundness of $\epsilon_{rbr} = \kappa_{rbr} = O(\frac{\max(h, \log n)}{|\mathbb{F}|})$. Furthermore, it has prover complexity of $O(h(n+t))$ proof size and verification time of $O(h \log n + \log^2 n + \log^2 t)$.*

A.4 Interactive Proofs & Polynomial IOPs

Definition 5 (Interactive Proofs). *An* interactive proof *for a NP-language $\mathcal{L}$ is a two-party protocol between a computationally unbounded* ***prover*** *$\mathcal{P}$ and a probabilistic polynomial-time (PPT)* ***verifier*** *$\mathcal{V}$ satisfying the following two properties:*

- ***Completeness:*** *For every $\mathbb{x} \in \mathcal{L}$, the honest prover convinces the verifier with high probability:*

$$\Pr\left[\langle \mathcal{P}, \mathcal{V} \rangle(\mathbb{x}) = 1\right] \geq 1 - \epsilon$$

  where the probability is over the randomness of the verifier, and ϵ is a negligible error bound.
- ***Soundness:*** *For every $\mathbb{x} \notin \mathcal{L}$ and for every (possibly malicious) prover $\mathcal{P}^*$,*

$$\Pr\left[\langle \mathcal{P}^*, V \rangle(\mathbb{x}) = 1\right] \leq \delta$$

  where the probability is over the randomness of the verifier, and δ is a negligible error bound.

Polynomial IOPs. A polynomial IOP (PIOP) is similar to an interactive proof with the only exception that the prover is also allowed to reply to the verifier with an oracle of a polynomial (which we denote with $[\![f]\!]$). Next, the verifier does not have read the entire polynomial but only query it at random evaluation points.

A.5 Arguments of Knowledge

Definition 6 ([CCC+25]). *A zero-knowledge succinct non-interactive argument of knowledge (zk-SNARK) for an indexed relation $\mathbb{i}$ is a tuple of* PPT *algorithms $\mathsf{S} = (\mathcal{G}, \mathcal{I}, \mathcal{P}, \mathcal{V})$ with the following algorithms:*

- $\mathcal{G}(1^\lambda) \to \mathsf{pp}$*: On input the security parameter λ, the randomized setup algorithm returns a structured reference* pp*.*
- $\mathcal{I}(\mathsf{pp}, \mathbb{i}) \to (\mathsf{pk}, \mathsf{vk})$*: On input the structured reference* pp *and an index $\mathbb{i}$, the deterministic indexing algorithm outputs a proving key* pk *and a verification key* vk*.*
- $\mathcal{P}(\mathsf{pk}, \mathbb{x}, \mathbb{w}) \to \pi$*: On input the proving key* pk *(for index $\mathbb{i}$), a statement-witness pair $(\mathbb{x}, \mathbb{w})$ such that $(\mathbb{i}, \mathbb{x}, \mathbb{w}) \in \mathcal{R}$, the randomized proving algorithm outputs a proof π.*
- $\mathcal{V}(\mathsf{vk}, \mathbb{x}, \pi) \to b$*: On input the verification key* vk *(for index $\mathbb{i}$), a statement $\mathbb{x}$, and proof π, the deterministic verification algorithm outputs accept or reject.*

Definition 7 (Completeness). *A zk-SNARK for an indexed relation $\mathcal{R}$ is correct if for any $(\mathbb{i}, \mathbb{x}, \mathbb{w}) \in \mathcal{R}$, we have:*

$$\Pr\left[\begin{array}{c} (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathcal{I}(\mathcal{G}(1^\lambda), \mathbb{i}) \\ \pi \leftarrow \mathcal{P}(\mathsf{pk}, \mathbb{x}, \mathbb{w}) \\ : \\ \mathcal{V}(\mathsf{vk}, \mathbb{x}, \pi) = 1 \end{array}\right] = 1.$$

Definition 8 (Succinctness). *A zk-SNARK is succinct if $\mathcal{P}$ outputs proofs of size $\mathsf{poly}(\lambda, \log|\mathbb{w}|)$, the running time of $\mathcal{P}$ is $\mathsf{poly}(\lambda, |\mathbb{x}|, |\mathbb{w}|)$, and the running time of $\mathcal{V}$ is $\mathsf{poly}(\lambda, |\mathbb{x}|, \log|\mathbb{w}|)$.*

Definition 9 (Knowledge Soundness). *A zk-SNARK for an indexed relation $\mathcal{R}$ satisfies knowledge soundness if for any PPT adversary $\mathcal{A}$, there exists a polynomial-time extractor $Ext_{\mathcal{A}}$ (running on the same input of $\mathcal{A}$, including its random coins) such that:*

$$\Pr\left[\begin{array}{c} \mathsf{pp} \leftarrow \mathcal{G}(1^\lambda), \\ ((\mathbb{i}, \mathbb{x}, \pi), \mathbb{w}) \leftarrow (\mathcal{A}(\mathsf{pp}), Ext_{\mathcal{A}}(\mathsf{pp})), \\ (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathcal{I}(\mathsf{pp}, \mathbb{i}) \\ : \\ \mathcal{V}(\mathsf{vk}, \mathbb{x}, \pi) = 1 \ \wedge \ (\mathbb{i}, \mathbb{x}, \mathbb{w}) \notin \mathcal{R} \end{array}\right] \leq \mathsf{negl}(\lambda).$$

Definition 10 (Zero knowledge). *A zk-SNARK is considered to be zero-knowledge if there is a* PPT *simulator $\mathcal{S}$ such that for every adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$, auxiliary input $z \in \{0,1\}^{poly(\lambda)}$, it holds that*

$$\left|\Pr\left[\begin{array}{c} \mathsf{pp} \leftarrow \mathcal{G}(1^\lambda) \\ (\mathbb{i}, \mathbb{x}, \mathbb{w}, st) \leftarrow \mathcal{A}_1(z, \mathsf{pp}) \\ (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathcal{I}(\mathsf{pp}, \mathbb{i}) \\ \pi \leftarrow \mathcal{P}(\mathsf{pk}, \mathbb{x}, \mathbb{w}) \\ b \leftarrow \mathcal{A}_2(\pi, st, \mathbb{i}, \mathbb{x}) \\ : \\ b = 1 \wedge (\mathbb{i}, \mathbb{x}, \mathbb{w}) \in \mathcal{R} \end{array}\right] - \Pr\left[\begin{array}{c} (\mathsf{pp}, \tau) \leftarrow \mathcal{S}(1^\lambda) \\ (\mathbb{i}, \mathbb{x}, \mathbb{w}, st) \leftarrow \mathcal{A}_1(z, \mathsf{pp}) \\ (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathcal{I}(\mathsf{pp}, \mathbb{i}) \\ \pi \leftarrow \mathcal{S}(\tau, z, \mathsf{pp}, \mathbb{x}) \\ b \leftarrow \mathcal{A}_2(\pi, st, \mathbb{i}, \mathbb{x}) \\ : \\ b = 1 \wedge (\mathbb{i}, \mathbb{x}, \mathbb{w}) \in \mathcal{R} \end{array}\right]\right| \leq \mathsf{negl}(\lambda).$$

This definition captures statistical zero-knowledge, since $\mathcal{A}$ is unbounded. However, if $\mathcal{A}$ is a probabilistic polynomial-time (PPT) adversary, then the definition corresponds to computational zero-knowledge. The notion of perfect zero-knowledge requires the two probabilities to be exactly equal.

Definition 11 (Polynomial Interactive Oracle Proof [CBBZ23, HLSS24]). *An interactive oracle proof for a relation $\mathcal{R}$ satisfies the following:*

- *Every message from the prover is a polynomial oracle $([\![f]\!], d_1, \ldots, d_\ell)$, where $f \in \mathbb{F}[x_1, \ldots, x_\ell]$, and d_i denotes the degree of f in variable x_i.*
- *Every message from the verifier consists of a random challenge in $\mathbb{F}^\ell$.*
- *The verifier receives oracle access to polynomial evaluations at any points.*

The complexity of PIOP is measured as follows:

- ***Prover complexity:*** *The total runtime of the PIOP prover.*
- ***Verifier complexity:*** *The total runtime of the PIOP verifier.*
- ***Query complexity:*** *The number of queries made by the verifier to the polynomial oracles.*
- ***The round complexity:*** *The number of rounds of the protocol.*
- ***Size of proof oracles:*** *The total length of the transmitted polynomials.*
- ***Size of the witness:*** *The total length of the witness polynomials.*

Chiesa *et al.* [CHM+20] formalize a method for compiling a PIOP into an argument of knowledge using a polynomial commitment scheme (PCS). In essence, the compilation process replaces all oracle polynomials in the PIOP with PCS commitments and adds evaluation proofs from the PCS for each polynomial query.

A.6 Polynomial Commitments

Definition 12 (Polynomial Commitment Scheme). *A multilinear polynomial commitment scheme* PC *over a field* $\mathbb{F}$ *consists of a tuple of algorithms* (Setup, Commit, Open, Eval)*:*

- $\mathsf{Setup}(1^\lambda, \ell) \to \mathsf{pp}$*: Takes a security parameter* λ *and* $\ell \in \mathbb{N}$ *(i.e., the number of variables in a polynomial) and outputs public parameters* pp.
- $\mathsf{Commit}(\mathsf{pp}, f) \to C$*: Takes a multilinear polynomial* $f \in \mathbb{F}[X_1, \ldots, X_\ell]$ *and outputs a commitment* C.
- $\mathsf{Open}(\mathsf{pp}, C, f) \to b$*: Takes a commitment* C *and a multilinear polynomial* $f \in \mathbb{F}[X_1, \ldots, X_\ell]$ *and outputs a bit* b.
- $\mathsf{Eval}(\mathsf{pp}, C, \mathbf{z}, y; f)$*: A protocol between the prover* $\mathcal{P}$ *and the verifier* $\mathcal{V}$ *with public input a commitment* C*, an evaluation point* $\mathbf{z} \in \mathbb{F}^\ell$*, and a value* $y \in \mathbb{F}$*. The prover* $\mathcal{P}$ *additionally knows a multilinear polynomial* $f \in \mathbb{F}[X_1, \ldots, X_\ell]$ *and aims to convince* $\mathcal{V}$ *that* f *is an opening to* C *and* $f(\mathbf{z}) = y$*. The verifier outputs a bit* b *at the end of the protocol.*

The scheme PC *satisfies the following properties:*

(1) ***Completeness****: For any bound* $\ell \in \mathbb{N}$*, for any multilinear polynomial* $f \in \mathbb{F}[X_1, \ldots, X_\ell]$*, and any point* $\mathbf{z} \in \mathbb{F}^\ell$*,*

$$\Pr\left[\begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \ell) \\ C \leftarrow \mathsf{Commit}(\mathsf{pp}, f) \\ \vdots \\ \mathsf{Eval}(\mathsf{pp}, C, \mathbf{z}, f(\mathbf{z}); f) = 1 \end{array}\right] = 1.$$

(2) ***Binding****: For any* $\ell \in \mathbb{N}$ *and any probabilistic polynomial-time (PPT) adversary* $\mathcal{A}$*,*

$$\Pr\left[\begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \ell) \\ (C, f_0, f_1) \leftarrow \mathcal{A}(\mathsf{pp}) \\ b_0 \leftarrow \mathsf{Open}(\mathsf{pp}, C, f_0) \\ b_1 \leftarrow \mathsf{Open}(\mathsf{pp}, C, f_1) \\ \vdots \\ b_0 = b_1 = 1 \wedge f_0 \neq f_1 \end{array}\right] \leq \mathsf{negl}(\lambda).$$

(3) ***Knowledge Soundness****:* Eval *is an argument of knowledge for the relation*

$$\mathcal{R}_{\mathsf{Eval}}(\mathsf{pp}) = \left\{ \begin{array}{c} [(C, \mathbf{z}, y); f] \ : \\ f(\mathbf{z}) = y \ \wedge \ \mathsf{Open}(\mathsf{pp}, C, f) = 1, \end{array} \right\}$$

where $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \ell)$*. That is, for any* $\ell \in \mathbb{N}$ *and any PPT algorithms* $\mathcal{A}$ *and* $\mathcal{P}^*$*, there exists an expected polynomial-time extractor Ext such that, for any randomness* r*,*

$$\Pr\left[\begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \ell) \\ (C, \mathbf{z}, y) \leftarrow \mathcal{A}(\mathsf{pp}, r) \\ \mathsf{Eval}(\mathsf{pp}, C, \mathbf{z}, y)\langle \mathcal{P}^*(r), \mathcal{V}\rangle = 1 \end{array}\right] \approx \Pr\left[\begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \ell) \\ (C, \mathbf{z}, y) \leftarrow \mathcal{A}(\mathsf{pp}, r) \\ f \leftarrow \mathit{Ext}(\mathsf{pp}, r) \\ ((C, \mathbf{z}, y); f) \in R_{\mathsf{Eval}}(\mathsf{pp}) \end{array}\right].$$

B Protocols for Large Language Model Operations

B.1 Generalized Matrix Multiplication

One of the most frequent computations performed by LLMs and Neural Networks in general is the matrix multiplication. In this work we use the PIOP first proposed by Thaler [Tha13] relies on the sumcheck protocol to prove the correctness of matrix multiplications. For our purposes, we define a specialized relation for matrix multiplication defined as follows:

Definition 13 (General Matrix Multiplication Relation). *The relation of the generalized matrix multiplication is defined as:*

$$(\mathbb{x}; \mathbb{w}) \in \left(\begin{array}{c} (a, b, [\![f_{\boldsymbol{A}}]\!], [\![f_{\boldsymbol{B}}]\!], [\![f_{\boldsymbol{C}}]\!], [\![f_{\boldsymbol{D}}]\!]); (\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D}) : \\ \boldsymbol{D} = a\boldsymbol{A} \cdot \boldsymbol{B} + b\boldsymbol{C} \end{array} \right)$$

Where:

- $\boldsymbol{A} \in \mathbb{F}^{m \times l}, \boldsymbol{B} \in \mathbb{F}^{l \times n}$ *and* $\boldsymbol{C} \in \mathbb{F}^{m \times n}$ *the input matrices and* $\boldsymbol{D} \in \mathbb{F}^{m \times n}$ *the output matrix.*
- $f_{\boldsymbol{A}}, f_{\boldsymbol{B}}, f_{\boldsymbol{C}}$ *and* $f_{\boldsymbol{D}}$ *the MLEs of* $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}$ *and* $\boldsymbol{D}$*.*

Protocol Description. We provide a detailed description of the generalized matrix multiplication PIOP in **Protocol 3**. Our protocol is similar to that of the "standard" PIOP for matrix multiplications using the sumcheck protocol [Tha13] but slightly modified to encompass the constant terms a, b and the addition matrix $\mathbf{C} \in \mathbb{F}^{n \times m}$.

Protocol 3: General Matrix Multiplication PIOP

(1) $\mathcal{V}$: It samples random points $\mathbf{r}_1 \in \mathbb{F}^{\log m}$ and $\mathbf{r}_2 \in \mathbb{F}^{\log n}$ and sends them to $\mathcal{P}$.
(2) $\mathcal{P}$: Computes $y_C = f_C(\mathbf{r}_1, \mathbf{r}_2)$ and $y_D = f_D(\mathbf{r}_1, \mathbf{r}_2)$ and sends them to the verifier.
(3) $\mathcal{P}$-$\mathcal{V}$: Both parties interact following the sumcheck protocol over the instance:
$$a^{-1}(y_D - by_C) = \sum_{\mathbf{x} \in \{0,1\}^{\log l}} f_A(\mathbf{r}_1, \mathbf{x}) f_B(\mathbf{x}, \mathbf{r}_2)$$
(4) $\mathcal{V}$: At the end of the protocol, the verifier ends up with evaluation claims y_A, y_B at the random points $(\mathbf{r}_1, \mathbf{r}')$ and $(\mathbf{r}', \mathbf{r}_2)$. It uses $[\![f]\!]_A$, $[\![f]\!]_B$, $[\![f]\!]_C$ and $[\![f]\!]_D$ to validate the correctness of y_A, y_B, y_C and y_D respectively.

B.1.1 Batched Generalized Matrix Multiplication. Looking ahead, some PIOPs necessitate the prover to prove "in parallel" the correctness of multiple matrix to matrix products. For this, we introduce the following relation:

Definition 14 (Batched General Matrix Multiplication Relation). *The relation* $\mathcal{R}^{Batched}_{MatMul}$ *of the batched generalized matrix multiplication is defined as:*

$$(\mathbb{x}; \mathbb{w}) \in \left(\begin{array}{c} (\{a_i, b_i, [\![f_{\boldsymbol{A}_i}]\!], [\![f_{\boldsymbol{B}_i}]\!], [\![f_{\boldsymbol{C}_i}]\!], [\![f_{\boldsymbol{D}_i}]\!]\}_{i \in [h]}); \\ (\{\boldsymbol{A}_i, \boldsymbol{B}_i, \boldsymbol{C}_i, \boldsymbol{D}_i\}_{i \in [h]}) : \\ \boldsymbol{D}_i = a_i \boldsymbol{A}_i \cdot \boldsymbol{B}_i + b_i \boldsymbol{C}_i \end{array} \right)$$

Where:

- $\boldsymbol{A}_i \in \mathbb{F}^{m_i \times l_i}, \boldsymbol{B}_i \in \mathbb{F}^{l_i \times n_i}$ *and* $\boldsymbol{C}_i \in \mathbb{F}^{m_i \times n_i}$ *the input matrices and* $\boldsymbol{D}_i \in \mathbb{F}^{m_i \times n_i}$ *the output matrix for all* $i \in [h]$*.*
- $f_{\boldsymbol{A}_i}, f_{\boldsymbol{B}_i}, f_{\boldsymbol{C}_i}$ *and* $f_{\boldsymbol{D}_i}$ *the MLEs of* $\boldsymbol{A}_i, \boldsymbol{B}_i, \boldsymbol{C}_i$ *and* $\boldsymbol{D}_i$ *for all* $i \in [h]$*.*

Protocol Description. The PIOP for proving proving $\mathcal{R}_{BatchedMatMul}$ is almost identical to that of a single matrix multiplication PIOP

with the only exception that the prover uses the sumcheck protocol for batch proving multiple instances as defined in [CBBZ23]. For completeness we give a description of our PIOP in **Protocol 4**.

Protocol 4: Batched General Matrix Multiplication PIOP

(1) $\mathcal{V}$: It samples random points $\mathbf{r}_1 \in \mathbb{F}^{\log \max(m_i)}$ and $\mathbf{r}_2 \in \mathbb{F}^{\log \max(n_i)}$ and sends them to $\mathcal{P}$.
(2) $\mathcal{P}$: Computes $y_{C_i} = f_{C_i}(\mathbf{r}_1^{(i)}, \mathbf{r}_2^{(i)})$ and $y_{D_i} = f_{D_i}(\mathbf{r}_1^{(i)}, \mathbf{r}_2^{(i)})$ and sends them to the verifier, where $\mathbf{r}_1^{(i)}, \mathbf{r}_2^{(i)}$ are truncated sub-vectors of $\mathbf{r}_1, \mathbf{r}_2$ so that they fit m_i and n_i.
(3) $\mathcal{P}$-$\mathcal{V}$: Both parties interact following the batched sumcheck protocol over the instances:

$$a_i^{-1}(y_{D_i} - b_i y_{C_i}) = \sum_{\mathbf{x} \in \{0,1\}^{\log l_i}} f_{A_i}(\mathbf{r}_1, \mathbf{x}) f_{B_i}(\mathbf{x}, \mathbf{r}_2)$$

(4) $\mathcal{V}$: At the end of the protocol, the verifier ends up with all evaluation claims y_{A_i}, y_{B_i}. It uses $[\![f]\!]_{A_i}$, $[\![f]\!]_{B_i}$, $[\![f]\!]_{C_i}$ and $[\![f]\!]_{D_i}$ to validate the correctness of y_{A_i}, y_{B_i}, y_{C_i} and y_{D_i} respectively.

Theorem 6. ***Protocol 4*** *is a PIOP for $\mathcal{R}_{MatMul}^{Batched}$ with $\kappa_{rbr} = \epsilon_{rbr} = \max(\frac{h(1+2\log\max(n_i)\max(m_i))}{|\mathbb{F}|}, \epsilon_{sum})$ (knowledge) round-by-round soundness error where ϵ_{sum} the round-by-round soundness error of the sumcheck protocol. In addition, it has prover time of $O(\sum_{i\in[h]}(m_i l_i + n_i l_i + m_i n_i))$ field operations and proof size and verification time of $O(h + \log(\max\{l_i\}_{i\in[h]}))$.*

Complexity Analysis. We will focus on the batched general matrix multiplication. Note that in order to obtain the asymptotics for the non-batched we just set $h = 1$. With that in mind, observe that in step 2 the prover has to compute y_{C_i} and y_{D_i} for all $i \in [h]$ which takes $O(\sum_{i\in[h]} m_i n_i)$ field operations. Next, in step (3) it first has to compute $f_{\mathbf{A}_i}$, $f_{\mathbf{B}_i}$ for all $i \in [h]$ which takes $O(\sum_{i\in[h]}(l_i m_i + l_i n_i))$ field operation and finally generate the batched sumcheck proof which requires $O(\sum_{i\in[h]} l_i)$ field operations. Consequently, the total prover time will be $O(\sum_{i\in[h]}(m_i l_i + n_i l_i + m_i n_i))$. Regarding the proof size and verification time, in step (2) the verifier has to receive $2h$ field elements and in step (3) $O(h + \log(\max\{l_i\}_{i\in[h]})$ field elements leading to a proof size and verification time of $O(h + \log(\max\{l_i\}_{i\in[h]}))$.

Round-By-Round Soundness. Assume that $\mathbf{tr}_0 = \mathbb{x}$ is in a doomed state, that is, there exists $i \in [h]$ such that $a_i\mathbf{A}_i\cdot\mathbf{B}_i + b_i\mathbf{C}_i \neq \mathbf{D}_i$. Then, $\Pr[\mathbf{tr}_1 = (\mathbf{tr}_0, \mathbf{r}_1, \mathbf{r}_2, \{y_{C_i}, y_{D_i}\}_{i\in[h]}) \notin \mathcal{D}] \leq \frac{h(1+2\log\max(n_i)\max(m_i))}{|\mathbb{F}|}$. That is because for each $i \in [h]$ either:

(1) $f_{\mathbf{D}_i}(\mathbf{r}_1, \mathbf{r}_2)$ agrees with $f_{\mathbf{D}'_i}(\mathbf{r}_1, \mathbf{r}_2)$ where $a_i\mathbf{A}_i\cdot\mathbf{B}_i + b_i\mathbf{C}_i = \mathbf{D}'_i$ which can happen with probability at most $\log n_i m_i/|\mathbb{F}|$.
(2) $f_{\mathbf{C}_i}(\mathbf{r}_1, \mathbf{r}_2)$ agrees with $f_{\mathbf{C}'_i}(\mathbf{r}_1, \mathbf{r}_2)$ where $a_i\mathbf{A}_i\cdot\mathbf{B}_i + b_i\mathbf{C}'_i = \mathbf{D}_i$ which can happen with probability at most $\log n_i m_i/|\mathbb{F}|$.
(3) It folds that $f_{\mathbf{D}_i}(\mathbf{r}_1, \mathbf{r}_2) - b_i f_{\mathbf{C}_i}(\mathbf{r}_1, \mathbf{r}_2) = a_i f_{\mathbf{A}_i\cdot\mathbf{B}_i}(\mathbf{r}_1, \mathbf{r}_2)$, which can happen only with $2/|\mathbb{F}|$ probability.

By taking the union bound of the above cases we reach the desirable bound.

Next, assume that $\mathbf{tr}_1$ is doomed, that is, there exists $i \in [h]$ such that $a_i^{-1}(y_{D_i} - b_i y_{C_i}) \neq \sum_{\mathbf{x}\in\{0,1\}^{\log l_i}} f_{\mathbf{A}_i}(\mathbf{r}_1, \mathbf{x}) f_{\mathbf{B}_i}(\mathbf{x}, \mathbf{r}_2)$. This means that $\mathbf{tr}_1$ is the instance of a batch sumcheck protocol in a doomed state. Since the rest of the protocol corresponds to a batched sumcheck, from Theorem 1 we know that the round-by-round soundness of all other rounds will be at most ϵ_{sum}. Hence, the round-by-round soundness of the above PIOP will be $\epsilon_{rbr} = \max(\frac{h(1+2\log\max(n_i)\max(m_i))}{|\mathbb{F}|}, \epsilon_{sum})$.

Round-By-Round Knowledge Soundness. Let $\mathbf{tr}_f$ be the final transcript and assume that $\Pr[\mathbf{tr}_f \notin \mathcal{D}] > \epsilon_{rbr}$, where $\mathcal{D}$ the doomed state. We will show that it must hold that $(\mathbb{x}) \notin \mathcal{D}$. From the round-by-round knowledge soundness of the sumcheck protocol, it must hold $\mathbf{tr}_1 = (\mathbb{x}, \mathbf{r}_1, \mathbf{r}_2, \{y_{C_i}, y_{D_i}\}_{i\in[h]} \notin \mathcal{D}$. Now assume that $\Pr[\mathbf{tr}_1 = (\mathbb{x}, \mathbf{r}_1, \mathbf{r}_2, \{y_{C_i}, y_{D_i}\}_{i\in[h]})] > \epsilon_{rbr}$. Then by our discussion in the round-by-round soundness it must hold that $\mathbf{tr}_0 \notin \mathcal{D}$, otherwise we would reach a contradiction. Hence $\kappa_{rbr} = \epsilon_{rbr}$

B.2 Requantization & GeLU

In this section we provide more details about the requantization layer. Formally, we want to construct a PIOP for the following relation:

Definition 15 (Requantization Layer). *The relation $\mathcal{R}_{requant}$ of the requantization layer is defined as:*

$$(\mathbb{x}; \mathbb{w}) \in \left(\begin{array}{c} ([\![f_{\boldsymbol{x}}]\!], [\![f_{\boldsymbol{y}}]\!]); (\boldsymbol{x}, \boldsymbol{y}) : \\ \boldsymbol{y} = \text{ReQuant}(\boldsymbol{x}) \end{array} \right)$$

Where $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{F}^n$ the input/output vectors and $f_{\boldsymbol{x}}, f_{\boldsymbol{y}} : \mathbb{F}^{\log n} \to \mathbb{F}$ their MLEs and ReQuant the requantization function as we defined in the main body.

Protocol Description. Protocol 5, gives a detailed description of the PIOP for proving $\mathcal{R}_{requant}$.

Protocol 5: Requantization

(1) $\mathcal{P}$: Compute $\mathbf{x}' \leftarrow \text{Quant}(2^f\epsilon)\cdot\mathbf{x} + 2^{f+\delta=1}$.
(2) $\mathcal{P}$-$\mathcal{V}$: Interact following the lookup PIOP for $\mathcal{R}_{lkp}^{Requant}$ over decomposable tables with recombination functions $\text{Comb}_x, \text{Comb}_y$ as described in the main body. Note that the prover does not provide $[\![f_{\mathbf{x}'}]\!]$ to the verifier but only its evaluation claim.
(3) $\mathcal{V}$: At the end of that protocol, the verifier ends up with an evaluation claim $y_{\mathbf{x}'}$ of the MLE of $\mathbf{x}'$. It then converts that claim to a claim of the MLE of $\mathbf{x}$ by setting $y_{\mathbf{x}} \leftarrow (y_{\mathbf{x}'} - 2^{f+\delta-1})/\text{Quant}(2^f\epsilon)$ and queries the oracle $[\![f_{\mathbf{x}}]\!]$ to validate its correctness.

Theorem 7. ***Protocol 5*** *is a PIOP for $\mathcal{R}_{requant}$ with $\kappa_{rbr} = \epsilon_{rbr} = \epsilon_{lkp}$ (knowledge) round-by-round soundness error. In addition, it has prover complexity of $O(n)$ field operations and proof size and verification time of $O(\log^2 n)$.*

Complexity Analysis. Regarding prover time, step (1) requires $O(n)$ field operations while step (2) requires $O(n)$ field operations (see Theorem 3). Consequently, the total prover time is $O(n)$. Finally, proof size and verification time are identical to that of the lookup PIOP, namely, $O(\log^2 n)$.

Round-By-Round (Knowledge) Soundness. Since our protocol corresponds to a lookup PIOP, the round-by-round (knowledge) soundness will be identical to those of the lookup PIOP.

Merging requantization with Non-Linear Layers. Non-linear layers like GeLU and Softmax also make use of lookup tables at their core to prove their operation. If a non-linear layer is followed after a linear one, then, instead of adding a separate and explicit requantization layer we can merge both the *requantization* and the *non linear layer operation* as a unified layer. This is done by following the requantization lookup argument as described above, but replacing tables that perform range checking with ones that compute the non-linear layer. This not only reduces the proving time and proof size but also increases the accuracy since the non-linear layer now takes the unclamped output of the linear layer as its input. Next, we will show this unrolls when we want to prove the correct computation of the GeLu activation function. In subsequent sections, when describing a non-linear layer, we omit the requantization step to simplify the protocol description.

B.2.1 Merging requantization with GeLu. First, note that the GeLu function takes as input a vector $\mathbf{x} \in \mathbb{F}^n$ and outputs $\mathbf{y} \leftarrow \text{GeLu}(\mathbf{x})$ where $\mathbf{y}[i] = \mathbf{x}[i] \cdot \text{P}[X \leq \mathbf{x}[i]]$ where $\text{P}[X \leq \mathbf{x}[i]]$ is the cumulative distribution function of the standard Gaussian distribution. To prove the above function we consider two cases: (1) all elements of $\mathbf{x} \in \mathbb{F}^n$ are q-bit numbers (where q the quantization level) and (2) elements of $\mathbf{x} \in \mathbb{F}^n$ are q'-bit numbers or even larger where $q' > q$. In the first case, we apply the standard lookup PIOP over a table $\mathbf{T}_{\text{GeLu}} \in \mathbb{F}^{2^q}$ such that $\mathbf{T}_{\text{GeLu}}[j] = \text{Quant}(\text{DeQuant}(j) \cdot P[X \leq \text{DeQuant}(j)])$, for all $j \in [2^q]$. In the second case, we use the lookup argument of the requantization but (1) replace $\mathbf{T}_{q'/q}$ with $\mathbf{T}_{\text{GeLu}}$ and (2) the edge values of Comb_y should be replaced with $\text{Quant}(\text{DeQuant}(2^{q-1}) \cdot P[X \leq \text{DeQuant}(2^{q-1})])$ and $\text{Quant}(\text{DeQuant}(-2^{q-1}+1) \cdot P[X \leq \text{DeQuant}(-2^{q-1}+1)])$.

B.3 Embedding

In this section we provide more details about the PIOP for the embedding layer. Not the relation we try to prove is:

Definition 16 (Embedding Layer). *The* embedding relation $\mathcal{R}_{embed}$ *is the set of all pairs such that:*

$$(\mathbb{x}; \mathbb{w}) = \left(\begin{array}{c} (\mathbf{x}, [\![f_{\mathbf{T}}]\!], [\![f_{\mathbf{Y}}]\!]); (\mathbf{T}, \mathbf{Y}) : \\ \boldsymbol{Y} = \text{Embed}(\boldsymbol{T}, \boldsymbol{x}) \end{array} \right)$$

where

- $\mathbf{x} = (x_0, x_1, \ldots, x_{s-1}) \in \mathcal{V}^s$, $\mathbf{E} \in \mathbb{F}^{v \times d}$, *the input tokens vector,* $\boldsymbol{Y} \in \mathbb{F}^{s \times d}$ *the output matrix and* $\boldsymbol{T} \in \mathbb{F}^{t \times d}$ *the embedding matrix.*
- $f_{\mathbf{T}}$, $f_{\mathbf{Y}}$ *the MLEs of* $\mathbf{T}, \mathbf{Y}$.

Protocol Description. As we mentioned in the main body, we can re-write the embedding layer as $\mathbf{Y} = \mathbf{H}_{\mathbf{x}} \cdot \mathbf{T}$ where $\mathbf{H}_{\mathbf{x}}$ the one-hot encoding matrix defined as follows:

Definition 17 (One-Hot Encoding Matrix). *Let* $\mathcal{V} = \{0, 1, \ldots, t-1\}$ *be the set containing the token identifiers. For a sequence* $\mathbf{x} = (x_0, x_1, \ldots, x_{s-1}) \in \mathcal{V}^s$, *the* one-hot encoding matrix $\mathbf{H}_{\mathbf{x}} \in \{0,1\}^{s \times v}$ *of* $\mathbf{x}$ *is defined entrywise by*

$$\mathbf{H}_{\mathbf{x}}[i, j] = \begin{cases} 1 & \text{if } j = x_i, \\ 0 & \text{otherwise,} \end{cases}$$

for all $i \in \{0, \ldots, s-1\}$ *and* $j \in \{0, \ldots, t-1\}$.

Protocol 6 gives a detailed description of our PIOP. Initially, both parties interact following the matrix multiplication PIOP (step 1). In the last round, they end up with evaluation claims $y_{\mathbf{Y}}, y_{\mathbf{T}}$ and $y_{\mathbf{H}_x}$ of the MLEs of $\mathbf{Y}, \mathbf{T}$ and $\mathbf{H}_x$. The verifier uses $[\![f_{\mathbf{Y}}]\!]$ and $[\![f_{\mathbf{T}}]\!]$ to validate the correctness of the first two claims and validate the last one locally by checking if $y_{\mathbf{H}_x} = \sum_{(i,j) \in [s] \times [t] \wedge \mathbf{H}_{\mathbf{x}}[i,j]=1} \beta((\mathbf{i}, \mathbf{j}), \mathbf{r})$ which can be computed efficiently due to the sparsity of $\mathbf{H}_{\mathbf{x}}$.

Protocol 6: Embedding

(1) $\mathcal{P}$-$\mathcal{V}$: Interact following the PIOP for matrix multiplication for proving that $\mathbf{Y} = \mathbf{H}_x \cdot \mathbf{T}$.
(2) $\mathcal{V}$: The verifier ends up with evaluation claims $y_{\mathbf{Y}}$, $y_{\mathbf{T}}$ and $y_{\mathbf{H}_{\mathbf{x}}}$ of the MLEs of $\mathbf{Y}$, $\mathbf{T}$ and $\mathbf{H}_{\mathbf{x}}$ respectively. It uses $[\![f_{\mathbf{Y}}]\!]$ and $[\![f_{\mathbf{T}}]\!]$ to validate the correctness of the first two and locally checks if:

$$y_{\mathbf{H}_x} = \sum_{(i,j) \in [s] \times [t] \wedge \mathbf{H}_{\mathbf{x}}[i,j]=1} \beta((\mathbf{i}, \mathbf{j}), \mathbf{r})$$

Theorem 8. ***Protocol 6*** *is a PIOP for* $\mathcal{R}_{embed}$ *with* $\kappa_{rbr} = \epsilon_{rbr} = \epsilon_{MatMul}$ *(knowledge) round-by-round soundness error where* ϵ_{MatMul} *the round-by-round soundness error of the matrix multiplication PIOP. In addition, it has prover time of* $O((t+s) \cdot d)$ *field operations and proof size and verification time of* $O(\log t)$.

Complexity Analysis. Complexity analysis is derived directly from the complexity of the PIOP for matrix multiplication. Note that one can concretely improve the prover's performance by evaluating $f_{\mathbf{H}_x}(\mathbf{r}', \mathbf{x})$ using $O(s)$ field operations and not $O(st)$. This is done by first computing the values $\beta(\mathbf{i}, \mathbf{r}')$ for all $\mathbf{i} \in \{0,1\}^{\log s}$ and then setting $f_{\mathbf{H}_x}(\mathbf{r}', \mathbf{j}) = \sum_{i \in [s] \wedge \mathbf{H}_{\mathbf{x}}[i,j]=1} \beta(\mathbf{i}, \mathbf{r})$. **Round-By-Round (Knowledge) Soundness.** It is not hard to see that the above protocol inherits the same round-by-round (knowledge) soundness of the matrix multiplication PIOP. Hence, we omit the proof.

B.4 Positional Encodings

Recall that positional encodings are key components in LLMs used to enable order in words. In this work, we will focus on proving the correctness of the two most used positional techniques: (1) Absolute Positional Encoding [VSP+17] (APE) and (2) Rotary Position Embedding [SLP+21] (RoPE).

B.4.1 Absolute Positional Encoding. In order to apply absolute positional encoding on an embedded input matrix $\mathbf{X} \in \mathbb{F}^{s \times d}$, we need to add it with a matrix s first rows of the matrix $\mathbf{W} \in \mathbb{F}^{s' \times d}$, computed during the training, where s' the largest possible sequence length. Hence, we want to construct a PIOP showing that $\mathbf{Y} = \mathbf{X} + \mathbf{W}[1:s]$, where the verifier holds the oracles of the MLEs of $\mathbf{Y}, \mathbf{X}, \mathbf{W}$. For the needs of our protocol we will always assume that the number of rows of $\mathbf{X}$ and $\mathbf{W}$ are powers of two (i.e., $s = 2^k$ and $s' = 2^{k'}$). More formally, we propose the following relation:

Definition 18 (Absolute Positional Encoding Relation). *The relation $\mathcal{R}_{APE}$ of the absolute positional encoding is defined as:*

$$(\mathbb{x};\mathbb{w}) \in \left(\begin{array}{c} (\{[\![f_Y]\!],[\![f_X]\!],[\![f_W]\!]\};(\{\boldsymbol{Y},\boldsymbol{X},\boldsymbol{W}\}: \\ \boldsymbol{Y}=\boldsymbol{X}+\boldsymbol{W}[1:s] \end{array} \right)$$

where:

- $\boldsymbol{Y},\boldsymbol{X} \in \mathbb{F}^{s\times d}$ *the input and output matrices and* $\boldsymbol{W} \in \mathbb{F}^{s'\times d}$ *the positional matrix.*
- $[\![f_Y]\!], [\![f_X]\!]$ *and* $[\![f_W]\!]$ *the MLEs of* $\boldsymbol{X}$, $\boldsymbol{Y}$ *and* $\boldsymbol{W}$ *respectively.*

Protocol Description. We give a detailed description of our protocol in **Protocol 7**. Our protocol is similar to the standard PIOP for matrix or vector addition, namely, the verifier selects a random point $\mathbf{r} \in \mathbb{F}^{\log(s\cdot d)}$, and checks if the sum of the evaluation claims of the inputs agrees with that of the output. As mentioned earlier, however, the matrix $\mathbf{W}$ has a larger number of rows compared to $\mathbf{X}$—hence the random point selected by the verifier does not "fit" the MLE of $\mathbf{W}$. To overcome that limitation, we observe that (1) we only need the first s rows of $\mathbf{W}$ and that (2) $f'(x) = \beta(\mathbf{0},\mathbf{y})f'(\mathbf{x}) = f_{\mathbf{W}}(\mathbf{0},\mathbf{x})$, where $f' : \mathbb{F}^{\log s\cdot d} \to \mathbb{F}$ the MLE of the first s rows of $\mathbf{W}$ and $\mathbf{0} \in \mathbb{F}^{\log s'/s}$. Based on these observations, the prover can evaluate the first s rows of $\mathbf{W}$ using $\mathbf{r}$ by computing $y_W = f'(\mathbf{r}) = f_{\mathbf{W}}((\mathbf{0}||\mathbf{r}))$.

Protocol 7: Absolute Positional Encoding PIOP

(1) $\mathcal{V}$: Samples a random point $\mathbf{r} \in \mathbb{F}^{\log s\cdot d}$ and sends it to the prover.
(2) $\mathcal{P}$: Computes and sends to $\mathcal{V}$ the points $y_Y = f_{\mathbf{Y}}(\mathbf{r})$, $y_X = f_X(\mathbf{r})$ and $y_W = f_{\mathbf{W}}((\mathbf{0}||\mathbf{r}))$, where $\mathbf{0} \in \mathbb{F}^{\log(s'/s)}$.
(3) $\mathcal{V}$: Checks if $y_Y = y_X + y_W$ and queries the corresponding oracles to check their correctness.

Theorem 9. ***Protocol 7*** *is a PIOP for $\mathcal{R}_{APE}$ with $\kappa_{rbr} = \epsilon_{rbr} = \frac{3\log s\cdot d}{|\mathbb{F}|}$ (knowledge) round-by-round soundness error. In addition, it has prover time of $O(s\cdot d)$ field operations and proof size and verification time of $O(1)$.*

Complexity Analysis. Regarding the prover time observe the only computation performed by the prover happens at step (2) and requires $O(s\cdot d)$ field operations—just to compute y_Y, y_Z and y_W. Finally, note that the verifier only gets three evaluation claims, hence the proof size and verification time is $O(1)$ field elements.

Round-By-Round Soundness and Knowledge. Regarding round-by-round soundness, observe that if $\mathbf{tr}_0 = \mathbb{x}$ belongs to a doomed state (e.g., $\mathbf{Y} \neq \mathbf{X} + \mathbf{W}[1:s]$) then $\Pr[(\mathbf{tr}_0,\mathbf{r},y_Y,y_X,y_W) \notin \mathcal{D}] \leq \frac{3\log s\cdot d}{|\mathbb{F}|}$. This is because $(\mathbf{tr}_0,\mathbf{r},y_Y,y_X,y_W) \in \mathcal{D}$ if and only if for the random $\mathbf{r}$ the following hold:

(1) $f_{\mathbf{Y}}(\mathbf{r}) = f_{\mathbf{Y}'}(\mathbf{r})$ where $\mathbf{Y}' = \mathbf{X} + \mathbf{W}_{<s}$.
(2) $f_{\mathbf{X}}(\mathbf{r}) = f_{\mathbf{X}'}(\mathbf{r})$ where $\mathbf{Y} = \mathbf{X}' + \mathbf{W}_{<s}$.
(3) $f_{\mathbf{W}}(\mathbf{0}||\mathbf{r}) = f_{\mathbf{W}'}(\mathbf{0}||\mathbf{r})$ where $\mathbf{Y} = \mathbf{X} + \mathbf{W}'_{<s}$.

By the Schwartz-Zippel Lemma, and by taking the union bound, we have that $\Pr[(\mathbf{tr}_0,\mathbf{r},y_Y,y_X,y_W) \notin \mathcal{D}] \leq \frac{3\log s\cdot d}{|\mathbb{F}|}$. Hence, $\epsilon_{rbr} = \frac{3\log s\cdot d}{|\mathbb{F}|}$. Finally, we follow a similar argument to prove that $\kappa_{rbr} = \epsilon_{rbr}$.

B.4.2 Rotary Positional Encoding. At a very high level, a rotary positional encoding takes the input matrix $\mathbf{X} \in \mathbb{F}^{s\times d}$ and outputs $\mathbf{Y} \in \mathbb{F}^{s\times d}$ such that $\mathbf{Y}_i = \mathbf{X}_i \circ \mathbf{W}_{1,i} + \mathbf{X}'_i \circ \mathbf{W}_{2,i}$, where $\mathbf{X}'_i = [-\mathbf{X}_i[2],\mathbf{X}_i[1],\ldots,-\mathbf{X}_i[d],\mathbf{X}_i[d-1]]$ and $\mathbf{W}_1,\mathbf{W}_2 \in \mathbb{F}^{s'\times d}$ weight matrices learned during training (recall that $s \leq s'$ is the maximum sequence length). In this section, we show how to construct a PIOP for the following relation:

Definition 19 (Rotary Positional Encoding Relation). *The relation $\mathcal{R}_{RoPE}$ of the absolute positional encoding is defined as:*

$$(\mathbb{x};\mathbb{w}) \in \left(\begin{array}{c} (\{[\![f_Y]\!],[\![f_X]\!],[\![f_{W_1}]\!],[\![f_{W_2}]\!]\};(\{\boldsymbol{Y},\boldsymbol{X},\boldsymbol{W}_1,\boldsymbol{W}_2\}: \\ \boldsymbol{Y}_i=\boldsymbol{X}_i\circ\boldsymbol{W}_1+\boldsymbol{X}'_i\circ\boldsymbol{W}_2,\forall i\in[s] \end{array} \right)$$

where:

- $\boldsymbol{Y},\boldsymbol{X} \in \mathbb{F}^{s\times d}$ *the input and output matrices and* $\boldsymbol{W}_1,\boldsymbol{W}_2 \in \mathbb{F}^{s'\times d}$ *the positional matrices. Furthermore,* $\boldsymbol{X}'_i = [-\boldsymbol{X}_i[2], \boldsymbol{X}_i[1],\ldots,-\boldsymbol{X}_i[d],\boldsymbol{X}_i[d-1]]$ *for all* $i \in [s]$.
- $[\![f_Y]\!], [\![f_X]\!], [\![f_{W_1}]\!]$ *and* $[\![f_{W_2}]\!]$ *the MLEs of* $\boldsymbol{X}$, $\boldsymbol{Y}$, $\boldsymbol{W}_1$ *and* $\boldsymbol{W}_2$ *respectively.*

Protocol Description. Since computing a rotary positional encoding is simply a sum of two Hadamard products, one can directly show the correct computation of $\mathbf{Y}$ by using the sumcheck protocol for proving the correctness of a Hadamard product. In particular, the prover shows that:

$$y_Y = \sum_{\mathbf{x}\in\{0,1\}^{\log(s\cdot d)}} \beta(\mathbf{x},\mathbf{r})\big(f_{\mathbf{X}}(\mathbf{x})f_{\mathbf{W}_{1,\leq s}}(\mathbf{x}) + f_{\mathbf{X}'}(\mathbf{x})f_{\mathbf{W}_{2,\leq s}}(\mathbf{x})\big)$$

Where y_Y is the evaluation claim of $f_{\mathbf{Y}}$ at a random point picked by the verifier. Observe that at the end of that protocol the verifier will hold evaluation claims for all involved MLEs. Note that it can directly validate the correctness of the claims for $f_{\mathbf{Y}}, f_{\mathbf{X}}$ as well as $f_{\mathbf{W}_{2,\leq s}}, f_{\mathbf{W}_{2,\leq s}}$ (using the same technique as in the absolute positional encoding), since it has the corresponding oracles. What remains is to find a "lightweight" method of validating the correctness of a claim of $f_{\mathbf{X}'}$ *without having the prover send the oracle of* $f_{\mathbf{X}'}$. To achieve this, we will exploit the special structure of $\mathbf{X}'$ in order to efficiently evaluate $f_{\mathbf{X}'}$ by accessing only using a small number of evaluations of $f_{\mathbf{X}}$. In particular, observe that $f_{\mathbf{X}}(x,\mathbf{z}) = (1-x)f_{\mathbf{X}}(0,\mathbf{z}) + xf_{\mathbf{X}}(1,\mathbf{z})$ and that $f_{\mathbf{X}'}(x,\mathbf{z}) = (1-x)\big(-f_{\mathbf{X}}(1,\mathbf{z})\big) + xf_{\mathbf{X}(0,\mathbf{z})}$. Hence, given two evaluations $f_{\mathbf{X}}(1,\mathbf{r}), f_{\mathbf{X}}(0,\mathbf{r})$ one can easily compute $f(r_x,\mathbf{r})$. So, at the end of the sumcheck protocol, the prover sends evaluation claims of $f_{\mathbf{X}}(1,\cdot)$ and $f_{\mathbf{X}}(0,,\cdot)$. The verifier first uses these claims to check the correctness of the claim of $f_{\mathbf{X}'}$ and subsequently checks if these claims are correct using the evaluation claim of $f_{\mathbf{X}}$.

Protocol 8: Rotary Positional Encoding PIOP

(1) $\mathcal{V}$: Samples a random point $\mathbf{r} \in \mathbb{F}^{\log s\cdot d}$ and sends it to the prover.
(2) $\mathcal{P}$: Computes and sends to $\mathcal{V}$ the claim $y_Y = f_{\mathbf{Y}}(\mathbf{r})$.
(3) $\mathcal{P}$-$\mathcal{V}$: Both parties interact following the sumcheck protocol over the instance:

$$y_Y = \sum_{\mathbf{x}\in\{0,1\}^{\log(s\cdot d)}} \beta(\mathbf{x},\mathbf{r})\big(f_{\mathbf{X}}(\mathbf{x})f_{\mathbf{W}_1}(\mathbf{x}) + f_{\mathbf{X}'}(\mathbf{x})f_{\mathbf{W}_2}(\mathbf{x})\big)$$

Let $f_{\mathbf{X}}(x_1,\ldots,x_{\log sd}) = (1-x_1)f_{\mathbf{X}}^{(1)}(x_2,\ldots,x_{\log sd}) + x_1 f_{\mathbf{X}}^{(2)}(x_2,\ldots,x_{\log sd})$, where x_1 the least significant variable. The prover additionally sends to the verifier the evaluation claims $y_{X,1}$ and $y_{X,2}$ of $f_{\mathbf{X}}^{(1)}$ and $f_{\mathbf{X}}^{(2)}$.

(4) $\mathcal{V}$: Let $\mathbf{r}' = (r_1, \mathbf{r}'') \in \mathbb{F}^{\log sd}$ be the random challenges during the sumcheck protocol. The verifier checks if $y_{X'} = r_1 y_{X,1} - (1-r_1)y_{X,2}$ and $y_X = r_1 y_{X,2} + (1-r_1)y_{X,1}$. Furthermore, validates the correctness of $y_X, y_Y, y_{W_1}, y_{W_2}$ using the corresponding polynomial oracles. Note that for the last two, the verifier has to query them at the point $(\mathbf{0}, \mathbf{r}') \in \mathbb{F}^{\log s'd}$.

Theorem 10. ***Protocol 8*** *is a PIOP for $\mathcal{R}_{RoPE}$ has $k_{rbr} = \epsilon_{rbr} = \max(\frac{\log sd}{|\mathbb{F}|}, \epsilon_{sum})$ (knowledge) round-by-round soundness error. In addition, it has prover time of $O(s\cdot d)$ field operations and proof size and verification time of $O(\log(s\cdot d))$.*

Complexity Analysis. Regarding the prover time, step (2) requires $O(s\cdot d)$ field operations to compute y_Y and step (3) requires $O(s\cdot d)$ field operations to generate the sumcheck proof and compute $y_{X,1}$ and $y_{X,2}$. So, the total prover complexity is $O(s\cdot d)$ field operations. Regarding the proof size, at step (2) the verifier receives one field element and the proof complexity of the sumcheck of step (3) is $O(\log s\cdot d)$. Hence, the total proof size is $O(\log s\cdot d)$. As for the verification time, we follow a similar argument for steps (2),(3). Furthermore, since step (4) requires only $O(1)$ field operations, the total verification time will be $O(\log s\cdot d)$.

Round-By-Round Soundness. Assume that $\mathbf{tr}_0 = (\mathbb{x})$ is in a doomed state, that is, there exists $i \in [s]$ such that $\mathbf{Y}_i \neq \mathbf{X}_i \circ \mathbf{W}_{1,<s} + \mathbf{X}'_i \circ \mathbf{W}_{2,<s}$. We will show that $\Pr[(\mathbf{tr}_0, \mathbf{r}, y_Y) \notin \mathcal{D}] \leq \frac{\log sd}{|\mathbb{F}|}$. First, observe that $(\mathbf{tr}_0, \mathbf{r}, y_Y) \notin \mathcal{D}$ if and only if for the random $\mathbf{r}$ it holds that $f(\mathbf{r}) = \sum_{\mathbf{x}\in\{0,1\}^{\log(sd)}} \beta(\mathbf{x},\mathbf{r})\big(f_{\mathbf{X}}(\mathbf{x})f_{\mathbf{W}_1}(\mathbf{x}) + f_{\mathbf{X}'}(\mathbf{x})f_{\mathbf{W}_2}(\mathbf{x})\big)$. Due to the Schwartz-Zippel lemma, this happens with probability at most $\log sd/|\mathbb{F}|$. Next, assume that $\mathbf{tr}_1 \in \mathcal{D}$. Then, due to the round-by-round soundness of the sumcheck protocol, all subsequent transcripts will have round-by-round soundness error of at most ϵ_{rbr}. Finally, let $\mathbf{tr}_{2+\log sd} = (\mathbf{tr}_{1+\log sd}, y_{X,1}, y_{X,2})$ be the final transcript and assume that $\mathbf{tr}_{1+\log sd} \in \mathcal{D}$, namely, $K_{\log sd+1} \neq \beta(\mathbf{r}',\mathbf{r})\big(f_{\mathbf{X}}(\mathbf{r}')f_{\mathbf{W}_1}(\mathbf{r}') + f_{\mathbf{X}'}(\mathbf{r}')f_{\mathbf{W}_2}(\mathbf{r}')\big)$. Then, either (1) $K_{\log sd+1} \neq \beta(\mathbf{r}',\mathbf{r})\big(y_X y_{W_1} + y_{X'} y_{W_2} = \beta(\mathbf{r}',\mathbf{r})\big((r_1 y_{X,2} + (1-r_1)y_{X,1})y_{W_1} + (r_1 y_{X,1} - (1-r_1)y_{X,2})y_{W_2}\big)$ or (2) the oracle queries do not match. In both cases, the verifier fails hence $\Pr[\mathbf{tr}_{2+\log sd} \notin \mathcal{D}] = 0$. So, $\epsilon_{rbr} = \max(\frac{\log sd}{|\mathbb{F}|}, \epsilon_{sum})$.

Round-By-Round Knowledge Soundness. Assume that $\Pr[\mathbf{tr}_{2+\log sd} \notin \mathcal{D}] > \epsilon_{rbr}$. Then, the round-by-round knowledge soundness of the sumcheck protocol implies that $\mathbf{tr}_1 = (\mathbf{tr}_0, \mathbf{r}, y_Y) \notin \mathcal{D}$. Finally, assume that $\Pr[\mathbf{tr}_1 \notin \mathcal{D}] > \epsilon_{rbr}$. Then, it must hold that $\mathbf{tr}_0 \notin \mathcal{D}$ otherwise we would reach a contradiction due to the Schwartz-Zippel Lemma. Consequently, $\kappa_{rbr} = \epsilon_{rbr}$.

B.5 Argmax

The argmax layer is an activation function that takes as input a matrix $\mathbf{X} \in \mathbb{F}^{s\times d}$ and outputs a vector $\mathbf{y} \in \mathbb{F}^s$ such that $\mathbf{y}[i] = \max \mathbf{X}[i,:]$, for all $i \in [s]$. In this section, we construct a PIOP for proving the correct computation of an argmax. More formally, we propose the following relation:

Definition 20 (Argmax). *The relation $\mathcal{R}_{argmax}$ consists of all pairs*

$$(\mathbb{x};\mathbb{w}) = \big((\![f_{\boldsymbol{X}}]\!], [\![f_{\boldsymbol{m}}]\!]); (\mathbf{X},\mathbf{m}) : \boldsymbol{m}[i] = \max(\boldsymbol{X}[i,:]), \forall i \in [s]\big),$$

where $f_{\boldsymbol{X}}$ and $f_{\boldsymbol{m}}$ the MLEs of $\boldsymbol{X} \in \mathbb{F}^{s\times d}$ and $\boldsymbol{m} \in \mathbb{F}^s$.

Protocol Description. To prove the above, we decompose the argmax operation in two "easier" to prove checks: (i) the first one, checks if $\mathbf{m}[i]$ is larger compared to all elements of the i-th row of $\mathbf{X}$ and (ii) the second one ensures that $\mathbf{m}[i]$ is an element of the i-th row of $\mathbf{X}$. For (i) we prove that $\mathbf{D}[i,j] = \mathbf{m}[i] - \mathbf{X}[i,j] \geq 0$ for all $i,j \in [s]\times[d]$ while for (ii) we show that for all $i \in [s]$ there exists at least one $j \in [d]$ where $\mathbf{D}[i,j] = 0$, or alternatively, $\prod_{j\in[d]} \mathbf{D}[i,j] = 0$.

Based on the above, the prover first computes the vector $\mathbf{D}$ (step (1)) and then invokes the lookup PIOP with $\mathbf{D}$ as input to prove that $\mathbf{D}[i,j] \geq 0$, for all i,j. At the end of that protocol the verifier receives $y_{\mathbf{D},1}$ the evaluation claim of the MLE of $\mathbf{D}$ at $(\mathbf{r}_1^{(1)}, \mathbf{r}_2^{(1)})$. To "link" $\mathbf{D}$ with $\mathbf{m}$ and $\mathbf{X}$ for which the verifier has oracle access, the prover also sends evaluation claims $y_{\mathbf{m},1}, y_{\mathbf{X},1}$ at $\mathbf{r}_2^{(1)}$ and $(\mathbf{r}_1^{(1)}, \mathbf{r}_2^{(1)})$ respectively (step (3)). Then, the verifier checks if $y_{\mathbf{D},1} = y_{\mathbf{m},1} - y_{\mathbf{X},1}$. Note that if $\mathbf{D}$ then the above equation would hold because:

$$f_{\mathbf{D}}(\mathbf{r}_1,\mathbf{r}_2) = f_{\mathbf{M}}(\mathbf{r}_1,\mathbf{r}_2) - f_{\mathbf{X}}(\mathbf{r}_1,\mathbf{r}_2) = \sum_{\mathbf{i}\in\{0,1\}^{\log d}} \beta(\mathbf{i},\mathbf{r}_1) f_{\mathbf{m}}(\mathbf{r}_2) - f_{\mathbf{X}}(\mathbf{r}_1,\mathbf{r}_2) = f_{\mathbf{m}}(\mathbf{r}_2) - f_{\mathbf{X}}(\mathbf{r}_1,\mathbf{r}_2)$$

where $\mathbf{M} = [\mathbf{m}^T,\ldots,\mathbf{m}^T] \in \mathbb{F}^{s\times d}$.

Next, the prover uses the GKR protocol to prove the correct computation of a circuit that takes as input $\mathbf{D}$ and outputs $\prod_{j\in[d]} \mathbf{D}[i,j] = 0$, for all $i \in [s]$. In practice, we prove the above using the specialized GKR protocol for that type of circuit as proposed in [Tha13] (step (5)). Finally, the verifier ends up with an evaluation claim of the MLE of $\mathbf{D}$ which is again linked with claims of the MLEs of $\mathbf{m}$ and $\mathbf{X}$ (step (6), (7)). Finally, in step (8), the verifier queries the oracles of $\mathbf{m}$ and $\mathbf{X}$ to validate the correctness of the claims.

Protocol 9: Argmax

(1) $\mathcal{P}$: Compute $\mathbf{D}[i,j] = \mathbf{m}[i] - \mathbf{X}[i,j]$, for all $i,j \in [s]\times[d]$.
(2) $\mathcal{P}$-$\mathcal{V}$: Run the lookup PIOP protocol for the relation $\mathcal{R}_{lkp}^{range}$ showing that $\mathbf{D} \geq 0$. Finally, the verifier ends up with an evaluation claim $y_{\mathbf{D},1}$ at $(\mathbf{r}_1^{(1)}, \mathbf{r}_2^{(1)})$.
(3) $\mathcal{P}$: Send the evaluation claims $y_{\mathbf{m},1}$ and $y_{\mathbf{X},1}$ of the MLEs of $\mathbf{m}$ and $\mathbf{X}$ at the points $\mathbf{r}_2^{(1)}$ and $(\mathbf{r}_2^{(1)}, \mathbf{r}_1^{(1)})$.
(4) $\mathcal{V}$: Check if $y_{\mathbf{D},1} = y_{\mathbf{m},1} - y_{\mathbf{X},1}$ and otherwise abort.
(5) $\mathcal{P}$-$\mathcal{V}$: Interact following the GKR protocol [GKR08] (and more specifically a variant of the grand product PIOP of [Tha13]) over a circuit that takes as input $\mathbf{D}$ and outputs:

$$\prod_{j\in[d]} \mathbf{D}[i,j] = 0, \forall i \in [s]$$

Finally, the verifier ends up with an evaluation claim $y_{\mathbf{D},2}$ at $(\mathbf{r}_1^{(2)}, \mathbf{r}_2^{(2)})$.

(6) $\mathcal{P}$: Send the evaluation claims $y_{\mathbf{m},2}$ and $y_{\mathbf{X},2}$ of the MLEs of $\mathbf{m}$ and $\mathbf{X}$ at the points $\mathbf{r}_2^{(2)}$ and $(\mathbf{r}_2^{(2)}, \mathbf{r}_1^{(2)})$.

(7) $\mathcal{V}$: Check if $y_{\mathbf{D},2} = y_{\mathbf{m},2} - y_{\mathbf{X},2}$ and otherwise abort.

(8) $\mathcal{V}$: Query the oracles $[\![f_{\mathbf{m}}]\!]$ and $[\![f_{\mathbf{D}}]\!]$ to validate the correctness of the claims $y_{\mathbf{m},1}, y_{\mathbf{m},2}$ and $y_{\mathbf{D},1}, y_{\mathbf{D},2}$ respectively.

Theorem 11. ***Protocol 9*** *is a PIOP for $\mathcal{R}_{argmax}$ with $\kappa_{rbr} = \epsilon_{rbr} = \max(\epsilon_{lkp}, \epsilon_{sum}, \frac{\log(sd)}{|\mathbb{F}|})$ (knowledge) round-by-round soundness error. In addition, it has prover time of $O(sd)$ field operations and proof size and verification time of $O(\log^2 sd)$.*

Complexity Analysis. Regarding the prover time observe that step (1) requires sd field additions, step (2) needs $O(sd)$ field operations (see Theorem 3), steps (3), (6) require $O(sd)$ field operations and step (5) needs $O(sd)$ field operations (see Section 5 of [Tha13]). Hence, the total prover time is $O(sd)$. Regarding the proof size and verification time, it is not to see that both are dominated by steps (2) and (5) where the prover has to run the GKR protocol for circuits with depth $\log(d)$. Hence, the total proof size and verification time is $O(\log^2(sd))$.

Round-By-Round (Knowledge) Soundness. Assume that the initial transcript $\mathbf{tr}_0 = \mathbb{x}$ is in a doomed state, that is, there exists $i \in [s]$ such that $\mathbf{m}[i] \neq \max(\mathbf{X}[i,:])$. Then, for the next transcript $\mathbf{tr}_1 = (\mathbf{tr}_0, (\mathbf{tr}_{lkp}, y_{\mathbf{m},1}, y_{\mathbf{X},1}))$ (where $\mathbf{tr}_{lkp}$ the transcript of the lookup protocol) it must hold that $\Pr[\mathbf{tr}_1 \notin \mathcal{D}] \leq \max(\frac{\log(sd)}{|\mathbb{F}|}, \epsilon_{lkp})$. To prove this, we consider the following cases:

(1) There exists i, j such that $\mathbf{D}[i,j] < 0$.
(2) There exists i, j such that $\mathbf{D}[i,j] \neq \mathbf{m}[i] - \mathbf{X}[i,j]$.

In case (1), the prover enters the lookup protocol in a doomed state which it can leave only with ϵ_{lkp} probability. In case (2), the prover can only leave the doomed state if it gets "lucky" and $y_{\mathbf{m},1} - y_{\mathbf{X},1} = f_{\mathbf{m}}(\mathbf{r}_2^{(1)}) - f_{\mathbf{X}}(\mathbf{r}_1^{(1)}, \mathbf{r}_2^{(1)})$ which happens only with $\frac{\log(sd)}{|\mathbb{F}|}$ due to the Schwartz-Zippel Lemma.

Next, assume that $\mathbf{tr}_1 \in \mathcal{D}$. Then, for the transcript $\mathbf{tr}_2 = (\mathbf{tr}_1, (\mathbf{tr}_{gkr}, y_{\mathbf{m},2}, y_{\mathbf{X},2}))$ it must hold that $\Pr[\mathbf{tr}_2 \notin \mathcal{D}] \leq \max(\frac{\log(sd)}{|\mathbb{F}|}, \epsilon_{gkr})$. To prove this, we consider two cases:

(1) There exists i, j such that $\mathbf{D}[i,j] \neq 0$.
(2) There exists i, j such that $\mathbf{D}[i,j] \neq \mathbf{m}[i] - \mathbf{X}[i,j]$.

In case (1), the prover enters the GKR protocol in a doomed state which it can leave only with probability at most ϵ_{gkr}. In case (2), the prover can only leave the doomed state if it gets "lucky" and $y_{\mathbf{m},2} - y_{\mathbf{X},2} = f_{\mathbf{m}}(\mathbf{r}_2^{(2)}) - f_{\mathbf{X}}(\mathbf{r}_1^{(2)}, \mathbf{r}_2^{(2)})$ which happens only with $\frac{\log(sd)}{|\mathbb{F}|}$ due to the Schwartz-Zippel Lemma.

Hence the round-by-round soundness of the above protocol is $\max(\epsilon_{gkr}, \epsilon_{lkp}, \frac{\log(sd)}{|\mathbb{F}|})$. Finally, to prove round-by-round (knowledge) soundness we follow a similar argument but in reverse order (see the example in Section 4.3 for more details).

B.6 Normalization Layers

In this section, we will show how to prove the correct computation of normalization layers. We will focus on the ones heavily used in LLMs, namely, RMSNorm and LayerNorm.

B.6.1 RMSNorm. The PIOP for the following relation is given in **Protocol 10**.

Definition 21 (RMSNorm). *The relation $\mathcal{R}_{RMSNorm}$ consists of all pairs*

$$(\mathbb{x}; \mathbb{w}) = \big(([\![f_{\mathbf{X}}]\!], [\![f_{\mathbf{Y}}]\!]); (\mathbf{X}, \mathbf{Y}) : \mathbf{Y} = RMSNorm(\mathbf{X})\big)$$

where $f_{\mathbf{X}}$ and $f_{\mathbf{Y}}$ the MLEs of the $\mathbf{X} \in \mathbb{F}^{s \times d}$ and $\mathbf{Y} \in \mathbb{F}^{s \times d}$ respectively.

Protocol 10: RMSNorm

(1) $\mathcal{P}$: Compute $\mathbf{d} \in \mathbb{F}^n$ such that:

$$\mathbf{d}[i] = \text{Quant}\Big(\frac{1}{\sqrt{\frac{1}{d}\sum_{j\in[d]} \mathbf{X}[i,j]^2}}\Big)$$

and send $[\![f_{\mathbf{d}}]\!]$ to $\mathcal{V}$. Furthermore, compute $\mathbf{X}' \in \mathbb{F}^{s\times d}$ such that $\mathbf{X}'[i,j] = \mathbf{d}[i] \cdot \mathbf{X}[i,j]$.

(2) $\mathcal{P}$-$\mathcal{V}$: Both parties interact following the PIOP for $\mathcal{R}_{requant}$ on inputs $(\mathbf{Y}, \mathbf{X}')$ but without having to send $[\![f_{\mathbf{X}'}]\!]$. Instead, the verifier receives an evaluation claim $y_{\mathbf{X}'}$ of $f_{\mathbf{X}'}$ at $\mathbf{r} = (\mathbf{r}_1, \mathbf{r}_2) \in \mathbb{F}^{\log s} \times \mathbb{F}^{\log d}$.

(3) $\mathcal{P}$-$\mathcal{V}$: Run the sumcheck protocol for the instance:

$$y_{\mathbf{X}'} = \sum_{\mathbf{x}_1 \in \{0,1\}^{\log s}} f_{\mathbf{d}}(\mathbf{x}_1) \sum_{\mathbf{x}_2 \in \{0,1\}^{\log d}} f_{\mathbf{X}}(\mathbf{x}_1, \mathbf{x}_2)$$

(4) $\mathcal{V}$: At the end of the protocol, the verifier receives the evaluation claims $y_{\mathbf{X}}$ and $y_{\mathbf{d}}$ and checks their correctness using $[\![f_{\mathbf{X}}]\!]$ and $[\![f_{\mathbf{d}}]\!]$ respectively.

(5) $\mathcal{P}$-$\mathcal{V}$: Let $\mathbf{y}_s \in \mathbb{F}^s$ be a vector such that $\mathbf{y}_s[i] = \sum_{j\in[s]} \mathbf{Y}[i,j]$. Both parties interact following the lookup protocol to prove that $\mathbf{y}_s \in \mathcal{R}_{lkp}^{range}$ (without having the prover send the oracle of the MLE of $\mathbf{y}_s$).

(6) $\mathcal{P}$-$\mathcal{V}$: Let y_s be the evaluation claim of the MLE of $\mathbf{y}_s$ at the point $\mathbf{r}_s$ produced at the end of the above protocol. Both parties interact following the sumcheck protocol for the instance:

$$y_s = \sum_{\mathbf{i}\in\{0,1\}^{\log s}} \beta(\mathbf{i}, \mathbf{r}_s) \sum_{\mathbf{j}\in\{0,1\}^{\log s}} f_{\mathbf{Y}}(\mathbf{i}, \mathbf{j})$$

(7) $\mathcal{V}$: Use $[\![f_{\mathbf{Y}}]\!]$ to validate the correctness of the evaluation claim of the MLE of $\mathbf{Y}$ produced at the final round of the sumcheck protocol and locally validate the correctness of the evaluation claim of $\beta(\mathbf{x}, \mathbf{r})$.

Theorem 12. ***Protocol 10*** *is a PIOP for $\mathcal{R}_{RMSNorm}$ with $\kappa_{rbr} = \epsilon_{rbr} = \max(\epsilon_{lkp}, \epsilon_{sum}, \frac{\log sd}{|\mathbb{F}|})$ (knowledge) round-by-round soundness error. In addition, it has prover time of $O(sd)$ field operations and proof size and verification time of $O(\log^2 sd)$.*

Complexity Analysis. Regarding the prover time, observe that step (1) requires $O(sd)$ field operations, step (2) requires $O(sd)$ field operations (see Theorem 7), step (3) requires $O(sd)$ field operations (see Theorem 1), step (5) requires $O(sd)$ operations for

computing $\mathbf{y}_s$ and $O(s)$ operations for proving that $\mathbf{y}_s \in \mathcal{R}_{lkp}^{range}$ and step (6) requires $O(sd)$ field operations (see Theorem 1). So, the total prover complexity is $O(sd)$ field operations. Regarding the proof size and verification time, they both are $O(\log^2 sd)$ as they are dominated by the requantization PIOP of step (2) (see Theorem 7).

Round-By-Round (Knowledge) Soundness. Assume that the initial transcript $\mathbf{tr}_0 = \mathbb{x}$ is in the doomed state. We will show that for the transcript $\mathbf{tr}_1 = (\mathbf{tr}_0, \mathbf{tr}_{lkp})$, it must hold that $\Pr[\mathbf{tr}_1 \notin \mathcal{D}] \leq \epsilon_{requant} = \epsilon_{lkp}$. First, because $\mathbf{tr}_0 \in \mathcal{D}$, we consider the following cases:

(1) $(\mathbf{Y}, \mathbf{X}') \notin \mathcal{R}_{requant}$.
(2) There exists $i, j \in [s] \times [d]$ such that $\mathbf{X}'[i, j] \neq \mathbf{d}[i] \cdot \mathbf{X}[i, j]$.
(3) There exists $i \in [s]$ such that $\sum_{j \in [d]} \mathbf{Y}[i, j]$ is not "close" to the quantized value of d/δ_4.

If case 1 holds, then, the PIOP prover of $\mathcal{R}_{requant}$ enters in a doomed state and can only leave that state with probability $\epsilon_{requant}$. If case 2 holds, then the prover can only leave the doomed state if it gets "lucky" and the evaluation claim $y_{\mathbf{X}'}$ equals to:

$$\sum_{\mathbf{x}_1 \in \{0,1\}^{\log s}} f_{\mathbf{d}}(\mathbf{x}_1) \sum_{\mathbf{x}_2 \in \{0,1\}^{\log d}} f_{\mathbf{X}}(\mathbf{x}_1, \mathbf{x}_2)$$

Due to the Schwartz-Zippel Lemma, this can happen only with probability $\frac{\log sd}{|\mathbb{F}|}$. Finally, if case 3 holds, then, the prover remains in a doomed state with probability 1. Hence, $\Pr[\mathbf{tr}_1 \notin \mathcal{D}] \leq \max(\frac{\log sd}{|\mathbb{F}|}, \epsilon_{requant})$.

Next, assume that $\mathbf{tr}_1 \in \mathcal{D}$. We will show that for the transcript $\mathbf{tr}_2 = (\mathbf{tr}_1, \mathbf{tr}_{sum})$ it must hold that $\Pr[\mathbf{tr}_2 \notin \mathcal{D}] \leq \epsilon_{sum}$. Since $\mathbf{tr}_1 \in \mathcal{D}$ we consider the following cases:

(1) It holds that $y_{\mathbf{X}'} \neq \sum_{\mathbf{x}_1 \in \{0,1\}^{\log s}} f_{\mathbf{d}}(\mathbf{x}_1) \sum_{\mathbf{x}_2 \in \{0,1\}^{\log d}} f_{\mathbf{X}}(\mathbf{x}_1, \mathbf{x}_2)$
(2) There exists $i \in [s]$ such that $\sum_{j \in [d]} \mathbf{Y}[i, j]$ is not "close" to the quantized value of d/δ_4.

If case 1 holds, then the prover enters the sumcheck protocol in a doomed state which leaves only with probability ϵ_{sum}. Alternatively, if case 2 holds, then the prover stays in a doomed state with probability 1. Consequently, $\Pr[\mathbf{tr}_2 \notin \mathcal{D}] \leq \epsilon_{sum}$.

Now, assume that $\mathbf{tr}_2 \in \mathcal{D}$. We will show that for the transcript $\mathbf{tr}_3 = (\mathbf{tr}_2, \mathbf{tr}_{lkp})$ it must hold that $\Pr[\mathbf{tr}_3 \notin \mathcal{D}] \leq \max(\epsilon_{lkp}, \frac{\log sd}{|\mathbb{F}|})$. To prove this, we consider the following cases given that $\mathbf{tr}_2 \in \mathcal{D}$:

(1) $\mathbf{y}_s \notin \mathcal{R}_{lkp}^{range}$.
(2) There exists $i \in [s]$ such that $\mathbf{y}_s[i] \neq \sum_{j \in [s]} \mathbf{Y}[i, j]$

If case 1 holds, then the PIOP prover for $\mathcal{R}_{lkp}^{range}$ enters the protocol in a doomed state which it leaves only with ϵ_{lkp} probability. If case 2 holds, then the prover leaves the doomed state if it gets "lucky" and for the evaluation claim y_s, it holds that:

$$y_s = \sum_{\mathbf{i} \in \{0,1\}^{\log s}} \beta(\mathbf{i}, \mathbf{r}_s) \sum_{\mathbf{j} \in \{0,1\}^{\log s}} f_{\mathbf{Y}}(\mathbf{i}, \mathbf{j})$$

This however, happens with probability $\frac{\log sd}{|\mathbb{F}|}$ due to the Schwartz-Zippel Lemma. Hence, $\Pr[\mathbf{tr}_3 \notin \mathcal{D}] \leq \max(\epsilon_{lkp}, \frac{\log sd}{|\mathbb{F}|})$.

Finally, assume that $\mathbf{tr}_3 \in \mathcal{D}$. We will show that for the transcript $\mathbf{tr}_4 = (\mathbf{tr}_3, \mathbf{tr}_{sum})$, it must hold that $\Pr[\mathbf{tr}_4 \notin \mathcal{D}] \leq \epsilon_{sum}$. Since $\mathbf{tr}_3 \in \mathcal{D}$ then it must hold that the sum in the sumcheck instance of step (6) is invalid. Thus, the prover would enter the sumcheck protocol in a doomed state and consequently the probability it leaves the doomed state is ϵ_{sum}.

Consequently, the round-by-round soundness of the above protocol is $\epsilon_{rbr} = \max(\epsilon_{lkp}, \epsilon_{sum}, \frac{\log sd}{|\mathbb{F}|})$. Finally, to prove round-by-round knowledge soundness we follow the same argument but in reverse order.

B.6.2 LayerNorm. We want to construct a PIOP for the following relation:

Definition 22 (LayerNorm). *The relation $\mathcal{R}_{layernorm}$ consists of all pairs*

$$(\mathbb{x}; \mathbb{w}) = \big((\llbracket f_{\mathbf{X}} \rrbracket, \llbracket f_{\mathbf{Y}} \rrbracket); (\mathbf{X}, \mathbf{Y}) : \boldsymbol{Y} = \mathit{LayerNorm}(\boldsymbol{X})\big)$$

where $f_{\mathbf{X}}$ and $f_{\mathbf{Y}}$ the MLEs of the $\boldsymbol{X} \in \mathbb{F}^{s \times d}$ and $\boldsymbol{Y} \in \mathbb{F}^{s \times d}$ respectively.

Protocol Description. In **Protocol 11** we present our PIOP for $\mathcal{R}_{layernorm}$. Initially, the prover computes $\mathbf{m}$, $\mathbf{X}'$ and $\mathbf{x}_s \in \mathbb{F}^s$ such that $\mathbf{x}_s[i] = \sum_{j \in [d]} \mathbf{X}[i, j]$ and sends the $\llbracket f_{\mathbf{m}} \rrbracket$ to the verifier (step (1)). Next, the prover interacts with the verifier following the PIOP for RMSNorm to show that $\mathbf{Y} = \text{RMSNorm}(\mathbf{X}') = \text{LayerNorm}(\mathbf{X} - \mathbf{m})$ (steps $(2) - (3)$). Finally, the prover interacts with the verifier to prove the well-formedness of $\mathbf{m}$ (steps $(4) - (7)$). In particular, the prover interacts with the verifier using the lookup PIOP to show that the vector $\text{Quant}(1/d)y_s - 2^q y_{\mathbf{m}} \approx 0$ (step 4). At, the end of this protocol the verifier holds evaluation claims of $y_{\mathbf{m}}$ and y_s of $f_{\mathbf{m}}, f_{\mathbf{x}_s}$ at a random point $\mathbf{r}_3 \in \mathbb{F}^{\log s}$. It uses $\llbracket f_{\mathbf{m}} \rrbracket$ to validate the correctness of $y_{\mathbf{m}}$ (step (5)) and reduces the claim y_s to a claim of $f_{\mathbf{X}}$ using the sumcheck protocol for the instance (step (6)):

$$y_s = \sum_{\mathbf{x}_1 \in \{0,1\}^{\log s}} \beta(\mathbf{x}_1, \mathbf{r}_3) \sum_{\mathbf{x}_2 \in \{0,1\}^{\log d}} f_{\mathbf{X}}(\mathbf{x}_1, \mathbf{x}_2)$$

Finally the correctness evaluation claim of $f_{\mathbf{X}}$ is checked using $\llbracket f_{\mathbf{X}} \rrbracket$.

Protocol 11: LayerNorm

(1) $\mathcal{P}$: Compute $\mathbf{m} = (\mu_1, \ldots, \mu_s)$, $\mathbf{X}'$ such that $\mathbf{X}'[i, j] = \mathbf{X}[i, j] - \mathbf{m}[i]$ for all $i, j \in [s] \times [d]$ and $\mathbf{x}_s \in \mathbb{F}^s$ such that $\mathbf{x}_s[i] = \sum_{j \in [d]} \mathbf{X}[i, j]$. Send $\llbracket f_{\mathbf{m}} \rrbracket$ to the verifier.
(2) $\mathcal{P}$-$\mathcal{V}$: Invoke the PIOP for $\mathcal{R}_{RMSNorm}$ using as input $(\mathbf{X}', \mathbf{Y})$. Note that the verifier does not hold $\llbracket f'_{\mathbf{X}} \rrbracket$ but instead, it only receives an evaluation claim $y_{\mathbf{X}'}$ of $f_{\mathbf{X}'}$ at $(\mathbf{r}_1, \mathbf{r}_2) \in \mathbb{F}^{\log s} \times \mathbb{F}^{\log d}$. Furthermore, the prover sends the evaluation claims $y_{\mathbf{X}}$ and $y_{\mathbf{m}}$ of $f_{\mathbf{X}}$ and $f_{\mathbf{m}}$ at $(\mathbf{r}_1, \mathbf{r}_2)$ and $\mathbf{r}_1$ respectively.
(3) $\mathcal{V}$: Check if $y_{\mathbf{X}'} = y_{\mathbf{X}} - y_{\mathbf{m}}$ and query $\llbracket f_{\mathbf{X}} \rrbracket$ and $\llbracket f_{\mathbf{m}} \rrbracket$ to validate the correctness of the claims.
(4) $\mathcal{P}$-$\mathcal{V}$: Invoke the PIOP for $\mathcal{R}_{lkp}^{range}$ with input $\mathbf{d}$ equal to $\text{Quant}(1/d)\mathbf{x}_s$ - $2^q\mathbf{m}$. At the end of the protocol, the verifier ends up with an evaluation claim $y_{\mathbf{d}}$ of $f_{\mathbf{d}}$ at a point $\mathbf{r}_3 \in \mathbb{F}^{\log s}$. In addition, it receives evaluation claims $y_s, y_{\mathbf{m}}$ of $f_{\mathbf{x}_s}$ and $f_{\mathbf{m}}$ at $\mathbf{r}_3$.
(5) $\mathcal{V}$: The verifier checks if $y_{\mathbf{d}} = \text{Quant}(1/d)y_s - 2^q y_{\mathbf{m}}$ and queries $\llbracket f_{\mathbf{m}} \rrbracket$ to check the correctness of $y_{\mathbf{m}}$.

(6) $\mathcal{P}$-$\mathcal{V}$: Both parties interact using the sumcheck protocol for the instance:
$$y_s = \sum_{\mathbf{x}_1 \in \{0,1\}^{\log s}} \beta(\mathbf{x}_1, \mathbf{r}_3) \sum_{\mathbf{x}_2 \in \{0,1\}^{\log d}} f_{\mathbf{X}}(\mathbf{x}_1, \mathbf{x}_2)$$
(7) $\mathcal{V}$: At the end of the protocol, the verifier obtains an evaluation claim of $f_{\mathbf{X}}$ which it validates by querying $[\![f_{\mathbf{X}}]\!]$.

Theorem 13. ***Protocol 11*** *is a PIOP for $\mathcal{R}_{layernorm}$ with $\kappa_{rbr} = \epsilon_{rbr} = \max(\epsilon_{lkp}, \epsilon_{sum}, \frac{\log sd}{|\mathbb{F}|})$ (knowledge) round-by-round soundness error. In addition, it has prover time of $O(sd)$ field operations and proof size and verification time of $O(\log^2 sd)$.*

Complexity Analysis. Regarding the prover time, observe that step (1) requires $O(sd)$ field operations, step (2) requires $O(sd)$ field operations (Theorem 12) step (4) requires $O(s)$ field operations and, finally, step (6) requires $O(sd)$ field operations. Hence, the total prover time is $O(sd)$ field operations. Finally, both the proof size and verification time have $O(\log^2 sd)$ complexity due to the step (2).

Round-By-Round (Knowledge) Soundness. Assume that the initial transcript $\mathbf{tr}_0 = \mathbb{x}$ is in a doomed state. We will show that for the transcript $\mathbf{tr}_1 = (\mathbf{tr}_0, \mathbf{tr}_{RMSNorm})$ it must hold that $\Pr[\mathbf{tr}_1 \notin \mathcal{D}] \leq \max(\epsilon_{RMSNorm}, \frac{\log sd}{|\mathbb{F}|})$. Since $\mathbf{tr}_0 \in \mathcal{D}$, we consider three cases:

(1) $(\mathbf{X}', \mathbf{Y}) \notin \mathcal{R}_{RMSNorm}$.
(2) There exists $i, j \in [s]] \times [d]$ such that $\mathbf{X}'[i,j] \neq \mathbf{X}[i,j] - \mathbf{d}[i]$.
(3) $\mathbf{m}$ is not well-formed.

If case 1 holds, then the PIOP prover for $\mathcal{R}_{RMSNorm}$ begins in a doomed state, so the probability it leaves this state is $\epsilon_{RMSNorm}$. If case 2 holds, then the prover leaves from a doomed state if it gets "lucky" and $y_{\mathbf{X}} - y_{\mathbf{m}} = f_{\mathbf{X}}(\mathbf{r}_1, \mathbf{r}_2) - f_{\mathbf{m}}(\mathbf{r}_1)$. Due to the Schwartz-Zippel Lemma, this can only happen with probability at most $\frac{\log sd}{|\mathbb{F}|}$. Finally, if case 3 holds, then the prover remains in a doomed state with probability 1. Hence, $\Pr[\mathbf{tr}_1 \notin \mathcal{D}] \leq \max(\epsilon_{RMSNorm}, \frac{\log sd}{|\mathbb{F}|})$.

Next, assume that $\mathbf{tr}_1 \in \mathcal{D}$. We will show that for the transcript $\mathbf{tr}_2 = (\mathbf{tr}_1, \mathbf{tr}_{lkp})$, it must hold that $\Pr[\mathbf{tr}_2 \notin \mathcal{D}] \leq \max(\epsilon_{lkp}, \frac{\log s}{|\mathbb{F}|})$. Because $\mathbf{tr}_1 \in \mathcal{D}$, we consider the following cases:

(1) $\mathbf{d} \notin \mathcal{R}^{range}_{lkp}$.
(2) $\mathbf{d} \neq \text{Quant}(1/d) y_s - 2^q y_{\mathbf{m}}$.

If case 1 holds, then the PIOP prover for $\mathcal{R}^{range}_{lkp}$ begins the protocol in a doomed state which can leave with probability at most ϵ_{lkp}. If case 2 holds, then the prover can leave a doomed state if $y_{\mathbf{d}} = \text{Quant}(1/d) f_{\mathbf{x}_s}(\mathbf{r}_3) - 2^q f_{\mathbf{m}}(\mathbf{r}_3)$ which happens with probability at most $\frac{\log s}{|\mathbb{F}|}$. Hence $\Pr[\mathbf{tr}_2 \notin \mathcal{D}] \leq \max(\epsilon_{lkp}, \frac{\log s}{|\mathbb{F}|})$.

Finally, assume that $\mathbf{tr}_2 \in \mathcal{D}$. We will show that for the final state $\mathbf{tr}_3 = (\mathbf{tr}_2, \mathbf{tr}_{sum})$, it must hold that $\Pr[\mathbf{tr}_3 \notin \mathcal{D}] \leq \epsilon_{sum}$. Since $\mathbf{tr}_2 \in \mathcal{D}$, it must hold that the sum y_s of the sumcheck instance of step (6) is invalid. Hence the sumcheck prover begins in a doomed state which it leaves with probability at most ϵ_{sum}.

B.7 Softmax

In this Section, we present our PIOP for proving the correct computation of a softmax layer. In more detail, given as input a matrix $\mathbf{X} \in \mathbb{F}^{s \times s}$, we want to prove that $\mathbf{Y} = \text{Softmax}(\mathbf{X})$, where $\mathbf{Y}[i,j] = \frac{e^{-\mathbf{X}[i,j]}}{\sum_{k \in [s]} e^{-\mathbf{X}[i,k]}}$. To achieve this, we rely on the techniques proposed by zkLLM [SLZ24]. That is, instead of proving the softmax function as is, which would require performing divisions within the zkSNARK, the prover works as follows: It uses as an addition witness information the matrix $\mathbf{Z} \in \mathbb{F}^{s \times s}$ claimed to be $\mathbf{Z}[i,:] = \ln(\sum_{k \in [s]} e^{-\mathbf{X}[i,k]})$ and utilizing the shift-invariance property of the softmax function, shows that $\mathbf{Y}[i,j] = \text{Softmax}(\mathbf{X})[i,j] = \text{Softmax}(\mathbf{X} - \mathbf{Z}) = e^{-\mathbf{X}[i,j] + \mathbf{Z}[i,j]}$.

In this way, proving the correctness of softmax reduces to proving the correctness of an exponentiation which can be proven directly using the lookup PIOP. Note, however, that this is not enough. The prover also need to show that $\mathbf{Z}$ has been selected honestly which is done by showing that $\sum_{j \in [s]} \mathbf{Y}[i][j]$ is very "close" to the quantized value of 1.0.

More formally, we want to construct a PIOP for the following relation:

Definition 23 (Softmax Layer). *The relation $\mathcal{R}_{softmax}$ as defined as:*
$$(\mathbb{x}; \mathbb{w}) \in \left(\begin{array}{c} ([\![f_Y]\!], [\![f_X]\!], [\![f_Z]\!]); (\boldsymbol{Y}, \boldsymbol{X}, \boldsymbol{Z}) : \\ \boldsymbol{Y} = Softmax(\boldsymbol{X} - \boldsymbol{Z}) \wedge \\ \sum_{j \in [s]} \boldsymbol{Y}[i,j] \approx Quant(1.0), \forall i \in [s] \end{array} \right)$$
where:
- $\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{Z} \in \mathbb{F}^{s \times s}$ *the input, output and auxiliary matrices.*
- f_X, f_Y, f_Z *the MLEs of $\boldsymbol{X}$, $\boldsymbol{Y}$ and $\boldsymbol{Z}$ respectively.*
- $\boldsymbol{Y}[i,j] = \frac{e^{-\boldsymbol{X}[i,j]}}{\sum_{k \in [s]} e^{-\boldsymbol{X}[i,k]}} = e^{-\boldsymbol{X}[i,j] + \boldsymbol{Z}[i,j]}$.

Protocol Description. We give a detailed description of our PIOP protocol. As mentioned earlier, the prover algorithm is almost identical to that of zkLLM [SLZ24]. The main difference between our protocol and that of zkLLM [SLZ24] is that their protocol only uses scaling factors of the form 2^k with $k \in \mathbb{Z}$ where as we allow for $k \in \mathbb{R}$ to allow for smaller step sizes between table entries for the same table size. Note that our protocol heavily relies on two relations $\mathcal{R}^{range}_{lkp}$ and $\mathcal{R}^{exp}_{lkp}$, both of them are proven using the lookup protocol [PH23]. Note that $\mathcal{R}^{range}_{lkp}$ holds if and only if all elements of a vector are within a proper range while $\mathcal{R}^{exp}_{lkp}$ shows that for every $(\mathbf{x}, \mathbf{y}) \in \mathcal{R}^{exp}_{lkp}$, $\mathbf{y} = e^{-\mathbf{x}}$. When proving both relations, we decompose each element of the input in 16-bit segments. For $\mathcal{R}^{exp}_{lkp}$ we also follow the optimization of [SLZ24] where we only compute the exponent of the most significant 16bit segment while simply range check the rest of the segments.

Protocol 12: Softmax

(1) $\mathcal{P}$: Prepare all witness data required to prove $\mathcal{R}^{range}_{lkp}$ and $\mathcal{R}^{exp}_{lkp}$ and send all required oracles to $\mathcal{V}$.
(2) $\mathcal{P}$-$\mathcal{V}$: Let $\mathbf{y}_s \in \mathbb{F}^s$ be a vector such that $\mathbf{y}_s[i] = \sum_{j \in [s]} \mathbf{Y}[i][j]$. Both parties interact following the

lookup protocol to prove that $\mathbf{y}_s \in \mathcal{R}_{lkp}^{range}$ (without having the prover send the oracle of the MLE of $\mathbf{y}_s$).

(3) $\mathcal{P}$-$\mathcal{V}$: Let y_s be the evaluation claim of the MLE of $\mathbf{y}_s$ at the point $\mathbf{r}_s$ produced at the end of the above protocol. Both parties interact following the sumcheck protocol for the instance:

$$y_s = \sum_{\mathbf{i}\in\{0,1\}^{\log s}} \beta(\mathbf{i}, \mathbf{r}_s) \sum_{\mathbf{j}\in\{0,1\}^{\log s}} f_{\mathbf{Y}}(\mathbf{i}, \mathbf{j})$$

(4) $\mathcal{V}$: Use $[[f_{\mathbf{Y}}]]$ to validate the correctness of the evaluation claim of the MLE of $\mathbf{Y}$ produced at the final round of the sumcheck protocol and locally validate the correctness of the evaluation claim of $\beta(\mathbf{x}, \mathbf{r})$.

(5) $\mathcal{P}$-$\mathcal{V}$: Interact using the lookup protocol to prove that $(\mathbf{D}, \mathbf{Y}) \in \mathcal{R}_{lkp}^{exp}$ where $\mathbf{D} = \mathbf{X} - \mathbf{Z}$.

(6) $\mathcal{V}$: At the end of the above protocol the verifier ends up with evaluation claims $y_{\mathbf{D}}, y_{\mathbf{Y}}$ of the MLEs of $\mathbf{D}$ and $\mathbf{Y}$ respectively. Additionally, it receives evaluation claims (at the same point with $y_{\mathbf{D}}$) $y_{\mathbf{X}}$ and $y_{\mathbf{Z}}$ of the MLEs of $\mathbf{X}$ and $\mathbf{Z}$. Finally, (1) check the validity of $y_{\mathbf{X}}, y_{\mathbf{Z}}, y_{\mathbf{Y}}$ using $[[f_{\mathbf{X}}]]$, $[[f_{\mathbf{Z}}]]$ and $[[f_{\mathbf{Y}}]]$ and (2) check if $y_{\mathbf{X}} - y_{\mathbf{Z}} = y_{\mathbf{D}}$.

B.7.1 Batch Proving Softmax Layers. As we will see next, the prover must show the correctness of softmax over multiple matrices simultaneously. For this, we introduce a new relation $\mathcal{R}_{Softmax}^{Batched}$ defined as follows:

Definition 24 (Batched Softmax Layer). *The relation $\mathcal{R}_{Softmax}^{Batched}$ of batched softmax layer is defined as:*

$$(\mathbb{x}; \mathbb{w}) \in \left(\begin{array}{c} \{([[f_{Y_k}]], [[f_{X_k}]], [[f_{Z_k}]]); (\boldsymbol{Y}_k, \boldsymbol{X}_k, \boldsymbol{Z}_k)\}_{k\in[h]} : \\ \boldsymbol{Y}_k = Softmax(\boldsymbol{X}_k - \boldsymbol{Z}_k) \wedge \\ \sum_{j\in[s]} \boldsymbol{Y}_k[i,j] \approx Quant(1.0), \forall i \in [s], k \in [h] \end{array} \right)$$

where:

- $\boldsymbol{X}_k, \boldsymbol{Y}_k, \boldsymbol{Z}_k \in \mathbb{F}^{s\times s}$ *the input, output and auxiliary matrices of the k-th instance where* $k \in [h]$.
- $f_{X_k}, f_{Y_k}, f_{Z_k}$ *the MLEs of* $\boldsymbol{X}_k$, $\boldsymbol{Y}_k$ *and* $\boldsymbol{Z}_k$ *respectively.*
- $\boldsymbol{Y}_k[i,j] = \frac{e^{-\boldsymbol{X}_k[i,j]}}{\sum_{k\in[s]} e^{-\boldsymbol{X}_k[i,k]}} = e^{-\boldsymbol{X}_k[i,j]+\boldsymbol{Z}_k[i,j]}$.

Protocol Description. Our PIOP is almost identical to that of proving a single softmax layer with the only exception that we replace the sumcheck and lookup PIOP with their batched variants. For completeness we give a detailed description of our PIOP on **Protocol 13**.

Protocol 13: Batched Softmax

(1) $\mathcal{P}$: Prepare all witness data required to prove $\mathcal{R}_{lkp}^{Batched,range}$ and $\mathcal{R}_{lkp}^{Batched,exp}$ and send all required oracles to $\mathcal{V}$.

(2) $\mathcal{P}$-$\mathcal{V}$: Let $\mathbf{y}_{k,s} \in \mathbb{F}^s$ be a vector such that $\mathbf{y}_{k,s}[i] = \sum_{j\in[s]} \mathbf{Y}_k[i][j]$ for all $k \in [h]$. Both parties interact following the batched lookup protocol to prove that $(\mathbf{y}_{1,s}, \ldots, \mathbf{y}_{h,s}) \in \mathcal{R}_{lkp}^{Batched,range}$ (without having the prover send the oracles of the MLE of each $\mathbf{y}_{k,s}$).

(3) $\mathcal{P}$-$\mathcal{V}$: Let $y_{k,s}$ be the evaluation claim of the MLE of $\mathbf{y}_{k,s}$ at the point $\mathbf{r}_s$ produced at the end of the above protocol for all $k \in [h]$. Both parties interact following the batched sumcheck protocol for the instances:

$$y_{k,s} = \sum_{\mathbf{i}\in\{0,1\}^{\log s}} \beta(\mathbf{i}, \mathbf{r}_s) \sum_{\mathbf{j}\in\{0,1\}^{\log s}} f_{\mathbf{Y}_k}(\mathbf{i}, \mathbf{j}), \forall k \in [h]$$

(4) $\mathcal{V}$: Use $[[f_{\mathbf{Y}_k}]]$ to validate the correctness of the evaluation claim of the MLE of $\mathbf{Y}_k$ produced at the final round of the sumcheck protocol for all $k \in [h]$ and locally validate the correctness of the evaluation claim of $\beta(\mathbf{x}, \mathbf{r})$.

(5) $\mathcal{P}$-$\mathcal{V}$: Interact using the lookup protocol to prove that $((\mathbf{D}_1, \mathbf{Y}_1), \ldots, (\mathbf{D}_h, \mathbf{Y}_h)) \in \mathcal{R}_{lkp}^{Batched,exp}$ where $\mathbf{D}_k = \mathbf{X}_k - \mathbf{Z}_k$ for all $k \in [h]$.

(6) $\mathcal{V}$: At the end of the above protocol the verifier ends up with evaluation claims $y_{\mathbf{D}_k}, y_{\mathbf{Y}_k}$ of the MLEs of $\mathbf{D}_k$ and $\mathbf{Y}_k$ respectively. Additionally, it receives evaluation claims (at the same point with $y_{\mathbf{D}_k}$) $y_{\mathbf{X}_k}$ and $y_{\mathbf{Z}_k}$ of the MLEs of $\mathbf{X}_k$ and $\mathbf{Z}_k$ for all $k \in [h]$. Finally, (1) check the validity of $y_{\mathbf{X}_k}, y_{\mathbf{Z}_k}, y_{\mathbf{Y}_k}$ using $[[f_{\mathbf{X}_k}]]$, $[[f_{\mathbf{Z}_k}]]$ and $[[f_{\mathbf{Y}_k}]]$ and (2) check if $y_{\mathbf{X}_k} - y_{\mathbf{Z}_k} = y_{\mathbf{D}_k}$ for all $k \in [h]$.

Theorem 14. ***Protocol 13*** *is a PIOP for $\mathcal{R}_{Softmax}^{Batch}$ with $\kappa_{rbr} = \epsilon_{rbr} = \max(\epsilon_{lkp}^{Batch}, \epsilon_{sum}, \frac{h\log s}{|\mathbb{F}|})$ (knowledge) round-by-round soundness error. In addition, it has a prover complexity of $O(h \cdot s^2)$ field operations and a proof size and verification time of $O(h\log s + \log^2 s))$.*

Complexity Analysis. Here, we will focus on the complexity analysis of the PIOP for the batched softmax since we can derive the complexity of the softmax one is derived by setting $h = 1$. Starting from the prover complexity, observe that step (2) requires $O(h \cdot s)$ field operations due to the complexity of the batch lookup PIOP, step (3) requires $O(h \cdot s^2)$ field operations due to the complexity of the sumcheck protocol and, finally, step (5) requires $O(h \cdot s^2)$ field operations due to the complexity of the batch lookup PIOP. Consequently, the total prover time is $O(h \cdot s^2)$. As for the prover time and verification time, steps (2),(3) and (5) have $O(h\log s + \log^2 s)$, $O(h+\log s)$ and $O(h\log s+\log^2 s)$ proof size and verification time respectively. Furthermore, step (2) has $O(\log s)$ verification time while step (6) has $O(1)$ verification time. Hence, the total proof size and verification time is $O(h\log s + \log^2 s)$.

Round-By-Round (Knowledge) Soundness. Assume that $\mathbf{tr}_0 = \mathbb{x}$ is in doomed state, that is, there exists $i \in [h]$ such that $\mathbf{Y}_i \neq \text{Softmax}(\mathbf{X}_i - \mathbf{Z}_i)$ or there exists a row $l \in [s]$, s.t $\sum_{j\in[s]} \mathbf{Y}_i[l,j]$ is "far" from $Quant(1.0)$. Then $\mathbf{tr}_1 = (\mathbf{tr}_0, \{\mathbf{y}_{k,s}\}_{k\in[s]})$ will also be in a doomed state with probability 1. Next, assuming that $\mathbf{tr}_1 \in \mathcal{D}$ we consider the following cases:

(1) $(\mathbf{y}_{1,s}, \ldots, \mathbf{y}_{h,s}) \in \mathcal{R}_{lkp}^{Batch,range}$ but $\exists i \in [s]$ s.t. $\mathbf{y}_{i,s} \neq \sum_{j\in[s]} \mathbf{Y}_i[j]$.

(2) $(\mathbf{y}_{1,s}, \ldots, \mathbf{y}_{h,s}) \notin \mathcal{R}_{lkp}^{Batch,range}$.

(3) $\exists k \in [s]$ s.t $(\mathbf{X}_k - \mathbf{Z}_k, \mathbf{Y}_k) \notin \mathcal{R}_{lkp}^{Batch,exp}$

Based on the above, the probability that $\mathbf{tr}_2 = (\mathbf{tr}_1, \mathbf{tr}_{lkp,1})$ leaves the doomed state, where $\mathbf{tr}_{lkp,1}$ is the transcript after the end of

the lookup protocol at step 3, is at most ϵ_{lkp}^{Batch}. More formally, $\Pr[\mathbf{tr}_2 \notin \mathcal{D}] \leq \max(\epsilon_{lkp}^{Batch}, \frac{h \log s}{|\mathbb{F}|})$. That is because if case (1) holds, then, the prover will leave the doomed state if it gets "lucky" and each $y_{k,s}$ equals to the evaluation of the MLE of $\sum_{j\in[s]} \mathbf{Y}_i[j]$ at $\mathbf{r}_s$. Due to the Schwartz-Zippel Lemma this can happen only with $\log s/|\mathbb{F}|$ probability. By taking the union bound we get the desired bound. Next, if case (2) holds the lookup PIOP prover enters the protocol in a doomed state and thus the probability it exits the non-doomed state will be ϵ_{lkp}^{Batch}. As for the case (3), the prover remains in a doomed state with probability 1.

Subsequently, assume that $\mathbf{tr}_2 \in \mathcal{D}$, then, we have the following cases:

(1) $(\mathbf{y}_{1,s}, \ldots, \mathbf{y}_{h,s}) \in \mathcal{R}_{lkp}^{Batch,range}$ but $\exists i \in [s]$ s.t. $\mathbf{y}_{i,s} \neq \sum_{j\in[s]} \mathbf{Y}_i[j]$.

(2) $\exists k \in [s]$ s.t $(\mathbf{X}_k - \mathbf{Z}_k, \mathbf{Y}_k) \notin \mathcal{R}_{lkp}^{Batch,exp}$

Then, for the transcript $\mathbf{tr}_3 = (\mathbf{tr}_2, \mathbf{tr}_{sum})$ it will hold that $\Pr[\mathbf{tr}_3 \notin \mathcal{D}] \leq \epsilon_{sum}$. That is because if case (1) holds then the prover enters the sumcheck protocol in a doomed state, and thus it can leave that state only with probability ϵ_{sum}. As for case (2), the prover stays in a doomed state with probability 1.

Now, assume that $\mathbf{tr}_3 \in \mathcal{D}$, that is, $\exists k \in [s]$ s.t $(\mathbf{X}_k - \mathbf{Z}_k, \mathbf{Y}_k) \notin \mathcal{R}_{lkp}^{Batch,exp}$. Then for the transcript $\mathbf{tr}_4 = (\mathbf{tr}_3, \mathbf{tr}_{lkp,2})$, it must hold that $\Pr[\mathbf{tr}_4 \notin \mathcal{D}] \leq \epsilon_{lkp}$ since the lookup PIOP prover enters in a doomed state and thus it can only leave that state with probability ϵ_{lkp}.

Finally, assume that $\mathbf{tr}_4 \in \mathcal{D}$ which means that there exists $y_{\mathbf{D}_k}$ that does not agree with the evaluation of the MLE of $\mathbf{D}_k$. Then, for the transcript $\mathbf{tr}_5 = (\mathbf{tr}_4, \{y_{\mathbf{X}_k}, y_{\mathbf{Z}_k}\}_{k\in[h]})$, it must hold that $\Pr[\mathbf{tr}_5 \notin \mathcal{D}] = 0$. This is because either the prover selects $y_{\mathbf{X}_k}, y_{\mathbf{Z}_k}$ where $y_{\mathbf{X}_k} - y_{\mathbf{Z}_k} = y_{\mathbf{D}_k}$ but do not agree with the oracles $[\![f_{\mathbf{X}_k}]\!]$, $[\![f_{\mathbf{Z}_k}]\!]$ or vice-versa. In both cases, the verifier will reject.

Until this point, we have shown that **Protocol 13** is round-by-round sound. To prove that it is round-by-round knowledge sound we follow a similar argument as we did in the previous proofs.

B.8 Multi-Head Attention Layer

In this section, we will show how to generate a zkSNARK proof for the main building block of LLMs which is the Multi-Head Attention Layer (MHE). To simplify discussion, we first focus on introducing a PIOP for a single-head attention layer, and later we show how to extend it in order to efficiently prove a MHE layer.

B.8.1 A PIOP for Single-Head Attention Layer. Recall that a single head attention layer is given an input matrix $\mathbf{X} \in \mathbb{F}^{s\times d}$ and three weight matrices $W_Q, W_K, W_V \in \mathbb{F}^{d\times d}$ and computes the output $Y \in \mathbb{F}^{s\times d}$ as $Y = \text{Softmax}(Q \cdot K^T/\sqrt{d}) \cdot V$, where $Q = X \cdot W_Q$, $K = X \cdot W_K$ and $V = X \cdot W_V$. More formally, we wish to construct a PIOP for the relation defined as follows:

Definition 25 (Single-Head Attention). *The relation $\mathcal{R}_{SHA}$ consists of the pairs:*

$$(\mathbb{x}; \mathbb{w}) = \left(\begin{array}{c} ([\![f_Y]\!], [\![f_X]\!]), ([\![f_{W_Q}]\!], [\![f_{W_K}]\!], [\![f_{W_V}]\!]; \\ (Y, X), (W_Q, W_K, W_V) \end{array} \right)$$

where

- $X, Y \in \mathbb{F}^{s\times d}$ *the input and output matrices.*
- $W_Q, W_K, W_V \in \mathbb{F}^{s\times d}$ *the weight matrices.*
- $f_Y, f_X, f_{W_Q}, f_{W_K}, f_{W_V} : \mathbb{F}^{\log sk} \to \mathbb{F}$ *the MLEs of* Y, X, W_Q, W_K *and* W_V *respectively.*
- $Y = Softmax(Q \cdot K^T/\sqrt{d}) \cdot V$, *where* $Q = X \cdot W_Q$, $K = X \cdot W_K$ *and* $V = X \cdot W_V$.

Protocol Description. At a very high level, the prover decomposes a single-head attention layer into "simpler" steps which can be proved using the PIOPs described above. In particular, the prover sequentially generates a proof over the following claims: (1) $Y = T_s \cdot V$, (2) $T_s = \text{Softmax}(T_p/\sqrt{d})$, (3) $T_p = Q \cdot K^T$ and that (4) $Q = X \cdot W_Q$, $K = X \cdot W_K$ and $V = X \cdot W_V$.

Protocol 14 gives a detailed description of our PIOP for the relation $\mathcal{R}_{SHA}$. First, the prover computes the intermediate matrices T_s and T_p and sends the oracle of the MLE of T_s, defined as $[\![f_{T_s}]\!]$ to the verifier (step 1). Next, the prover invokes the PIOP for matrix multiplication to prove (1) (steps 2-3) and the PIOP for the softmax layer to prove (2) (steps 4-5). At this point, observe that the verifier holds evaluation claims for the MLEs f_{T_s}, f_Y, f_V and $f_{(T_s/\sqrt{d})}$. It uses $[\![f_{T_s}]\!]$ and $[\![f_Y]\!]$ to validate the first two and keeps the rest to validate subsequent steps.

Next, in order to prove (3), both parties have to compute an evaluation claim for T_p–and not $T_p/\sqrt{d}$ (which was provided in the previous step). Thankfully, one can compute y_{T_p} from $y_{T_p/\sqrt{d}}$ by simply setting $y_{T_p} \leftarrow y_{T_p/\sqrt{d}} \cdot \text{Quant}(1/\sqrt{d})^{-1}$ where $\text{Quant}(1/\sqrt{d})^{-1}$ the quantized value of $1/\sqrt{d}$. Using y_{T_p}, the prover invokes the matrix multiplication PIOP to prove (3) (step 6). After that step, the verifier ends up with evaluation claims for the MLEs of the matrices Q, K^T and V which will be used to validate the correctness of (4). However, before doing so, note that the verifier has an evaluation claim for MLE of K^T and not of K. Nevertheless, we observe that given the evaluation claim pair (y_{K^T}, r_{K^T}), for a matrix $K \in \mathbb{F}^{s\times d}$ one can trivially compute (y_K, r_K) as follows: Parse r_{K^T} as $(r_1, r_2) \in \mathbb{F}^{\log d} \times \mathbb{F}^{\log s}$ and set $r_K \leftarrow (r_2, r_1)$ and $y_K \leftarrow y_{K^T}$. That is because by definition $K^T[x_1, x_2] = f_{K^T}(x_1, x_2) = f_K(x_2, x_1) = K[x_2, x_1]$, for all $x_1 \in \{0,1\}^{\log d}$ and $x_2 \in \{0,1\}^{\log s}$.

Provided the evaluation claims y_Q, y_K, y_V the prover invokes the PIOP for batch matrix multiplication to prove (4). Finally, the verifier ends up with evaluation claims of the MLEs of X, W_Q, W_K, W_V which are validated through the corresponding oracles.

Protocol 14: Single-Head Attention

(1) $\mathcal{P}$: While computing Y, store the matrices $T_s \in \mathbb{F}^{s\times s}$ and $T_p \in \mathbb{F}^{s\times d}$ where $T_s = \text{Softmax}(Q \cdot K^T/\sqrt{d})$ and $T_p = Q \cdot K^T$. Finally, send $[\![f_{T_s}]\!]$ to $\mathcal{V}$.

(2) $\mathcal{P}$-$\mathcal{V}$: Invoke the matrix multiplication PIOP to prove that $Y = T_s \cdot V$.

(3) $\mathcal{V}$ ends up with evaluation claims of f_Y, f_{T_s} and f_V at random points $r_Y, r_V \in \mathbb{F}^{\log(sd)}$ and $r_{T_s} \in \mathbb{F}^{2\log s}$, denoted with y_Y, y_{T_s} and y_V respectively. It checks if $y_Y = f_Y(r_Y)$ using $[\![f_Y]\!]$ and uses y_{T_s} and y_V to validate the next steps.

(4) $\mathcal{P}$-$\mathcal{V}$: Invoke the PIOP for softmax with input and output the matrices $T_s/\sqrt{d}$ and T_p respectively.

(5) $\mathcal{V}$: Ends up with evaluation claims y'_{T_s}, y_{T_p} of f_s and f_p at the points $r'_{T_s}, r_{T_p} \in \mathbb{F}^{2\log s}$. It uses $[\![f_s]\!]$ to check if $y_{T_s} = f_s(r_{T_s})$ and $y'_{T_s} = f_s(r'_{T_s})$.
(6) $\mathcal{P}$-$\mathcal{V}$: Set $y_{T_p} \leftarrow y_{T_p} \cdot \text{Quant}(1/\sqrt{d})^{-1}$ and interact over the PIOP for matrix multiplication using y_{T_p} as evaluation claim of the output matrix. $\mathcal{V}$ ends up with the evaluation claims y_Q, y_{K^T} of f_Q and f_{K^T} at the points r_Q and r_{K^T} respectively.
(7) $\mathcal{P}$-$\mathcal{V}$: Let $r_{K^T} = (r_1, r_2) \in \mathbb{F}^{\log d} \times \mathbb{F}^{\log s}$. Both parties set $y_K = y_{K^T}$ and $r_K = (r_2, r_1)$. Then, they invoke the PIOP for batched matrix multiplication to prove that $Q = X \cdot W_Q$, $K = X \cdot W_K$ and $V = X \cdot W_V$ using the evaluation claims (y_Q, r_Q), (y_K, r_K) and (y_V, r_V).
(8) $\mathcal{V}$: Ends up with three evaluation claims of f_X and one evaluation claim for each f_{W_Q}, f_{W_K} and f_{W_V} and uses $[\![f_X]\!]$, $[\![f_{W_Q}]\!]$, $[\![f_{W_K}]\!]$ and $[\![f_{W_V}]\!]$ to check their validity.

B.8.2 A PIOP for Single-Head Attention Layer with Masking. In the vast majority of state-of-the-art LLMs, an attention layer requires masking. That is, before using T_s as input to the softmax layer, we first mask it to T'_s such that for all i, j, where $i \geq j$, $T'_s[i, j] = T_s[i, j]$ and $T'_s[i, j] = -\infty$ otherwise. Subsequently, T'_s is used as input in the softmax layer. In this section, we will show how to construct a PIOP for proving the correctness of a SHA layer with masking. For this we first construct a specialized PIOP for masking and explain how to embed it in the **Protocol 14**.

A PIOP for Proving a Masking Operation. We want to generate a proof for the relation $\mathcal{R}_{Mask} = \{([\![f_{T_s}]\!], [\![f_{T'_s}]\!]); (T_s, T'_s) : T'_s = Mask(T_s)\}$. To achieve this, we first convert the masking operation into an "easy-to-prove" equation. In more detail, observe that $Mask(T_s) = T_s \circ D_1 + D_2$, where $D_1 \in \mathbb{F}^{s\times s}$ is a lower diagonal matrix filled with ones, $D_2 \in \mathbb{F}^{s\times s}$ an upper diagonal matrix filled with a symbol $-\infty$ and "$\circ$" the hadamard product between matrices. With that in mind, we propose a specialized PIOP which we describe in more detail on **Protocol 15**. Our protocol is almost identical with the specialized sumcheck protocol for proving a hadamard product with the only exception that the verifier needs to negate the impact of D_2 by computing $y' \leftarrow y_{T'_s} - f_{D_2}(r)$.

At this point, observe that naively computing evaluations of the MLEs of D_1 and D_2 would require $O(s^2)$ time. Fortunately, due to the "well-formedness" of both matrices this can be reduced in $O(s)$ time. To simplify discussion, we will show the verifier's algorithm for evaluating f_{D_1} since the algorithm for evaluating f_{D_2} is almost identical. For this, we first re-write $f_{D_1}(r_1, r_2)$ as $\sum_{\mathbf{i}\in\{0,1\}^{\log s}} \beta(\mathbf{i}, r_1) f^{(i)}_{D_1}(r_2)$ and evaluate each $f^{(i)}_{D_1}(r_2)$ individually. For this, observe that $f^{(i)}_{D_1}(r_2) = \sum_{j\in[i]} \beta(\mathbf{j}, r_2) = f^{(i-1)}_{D_1}(r_2) + \beta(\mathbf{i}, r_2)$. Based on the above, it is not hard to see that by computing the values $\beta(\mathbf{i}, r_2)$ for all $\mathbf{i} \in \{0,1\}^{\log s}$ one can evaluate all $f^{(i)}_{D_2}(r_2)$ in time $O(s)$. Finally, once these values have been evaluated, we can evaluate $f_{D_1}(r_1, r_2)$ in time $O(s)$—thus leading to a total evaluation time of $O(s)$.

Protocol 15: PIOP for Masking

(1) $\mathcal{V}$: Sample a random point $r \in \mathbb{F}^{2\log s}$ and send it to $\mathcal{P}$.
(2) $\mathcal{P}$: Compute $y_{T'_s} = f_{T'_s}(r)$ and send it to $\mathcal{V}$.
(3) $\mathcal{P}$-$\mathcal{V}$: First, both parties set $y' \leftarrow y_{T'_s} - f_{D_2}(r)$. Then, interact using the sumcheck protocol over the following instance:
$$y = \sum_{\mathbf{x}\in\{0,1\}^{2\log s}} \beta(r, \mathbf{x}) f_{T_s}(\mathbf{x}) f_{D_1}(\mathbf{x})$$
(4) $\mathcal{V}$: Ends up with evaluation claims y_{T_s} and y_{D_1} at a random point r'. It locally checks whether $y_{D_1} = f_{D_1}(r')$ and uses the oracles $[\![f_{T_s}]\!]$ and $[\![f_{T'_s}]\!]$ to validate the correctness of y_{T_s} and $y_{T'_s}$.

Embedding Masking in Protocol 14. Now, to prove the correct computation of a SHA layer using masking, the prover follows **Protocol 14** as is with the exception that after step 5 and before step 6 it invokes **Protocol 15**.

B.8.3 A PIOP for Multi-Head Attention Layer. We give a detailed description of the multi-head attention layer (MHA) in the main body. Here, we present our final PIOP in **Protocol 16**.

Protocol 16: Multi-Head Attention

(1) $\mathcal{P}$: Compute and store the matrices $T_{s,i} = \text{Softmax}(\hat{Q}_i \cdot \hat{K}_i^T/\sqrt{d_k})$ and $T_{p,i} = \hat{Q}_i \cdot \hat{K}_i^T$ for all $i \in [h]$. Finally, send all $[\![f_{T_{s,i}}]\!]$ to $\mathcal{V}$.
(2) $\mathcal{P}$-$\mathcal{V}$: Invoke the matrix multiplication PIOP to prove that $Y = \hat{Y} \cdot P_1^T$.
(3) $\mathcal{V}$: The verifier ends up with evaluation claims of the MLEs of Y, P_1^T and $\hat{Y}$. It uses $[\![f_Y]\!]$ to validate the first one and locally evaluates the second one. For the last one, it receives from $\mathcal{P}$ the evaluation claims of $y_{\hat{Y}_i}$ and checks if $y_{\hat{Y}} = \sum_{i\in[h]} \beta(\mathbf{i}, r_Y^{(1)}) y_{\hat{Y}_i}$.
(4) $\mathcal{P}$-$\mathcal{V}$: Using $\{y_{\hat{Y}_i}\}_{i\in[h]}$, invoke the PIOP for batch matrix multiplication to prove that $\hat{Y}_i = T_{s,i} \cdot \hat{V}_i$ for all $i \in [h]$.
(5) $\mathcal{V}$: Ends up with evaluation claims of $f_{T_{s,i}}$ and $f_{\hat{V}_i}$ which are in the next steps.
(6) $\mathcal{P}$-$\mathcal{V}$: Invoke the PIOP for batch proving a softmax layer with input and output the matrices $\{T_{p,i}/\sqrt{d}, T_{s,i}\}_{i\in[h]}$.
(7) $\mathcal{V}$: Ends up with evaluation claims $y'_{T_{s,i}}, y_{T_{p,i}}$ of $f_{T_{s,i}}$ and $f_{T_{p,i}}$. It uses $[\![f_{T_{s,i}}]\!]$ to check the correctness of $y_{T_{s,i}}$ and $y'_{T_{s,i}}$.
(8) $\mathcal{P}$-$\mathcal{V}$: Set $y_{T_{p,i}} \leftarrow y_{T_{p,i}} \cdot \text{Quant}(1/\sqrt{d_k})^{-1}$ and interact over the PIOP for batch matrix multiplication using $y_{T_{p,i}}$ as evaluation claims of the output matrix. $\mathcal{V}$ ends up with the evaluation claims $y_{\hat{Q}_i}, y_{\hat{K}_i^T}$, for $i \in [h]$.
(9) $\mathcal{V}$: Let $r_{K^T} = (r_1, r_2) \in \mathbb{F}^{\log \tilde{d}_k} \times \mathbb{F}^{\log s}$. $\mathcal{V}$ sets $y_K = y_{K^T}$ and $r_K = (r_2, r_1)$, samples a random point $r_h \in \mathbb{F}^{\log h}$ and computes $y_{\hat{Q}} = \sum_{i\in[h]} \beta(\mathbf{i}, r_h) y_{\hat{Q}_i}$ (and likewise it does for $y_{\hat{K}}$ and $y_{\hat{V}}$).

(10) $\mathcal{P}$-$\mathcal{V}$: Both parties interact over the PIOP for batched matrix multiplication using the evaluation claims $y_{\hat{Q}}, y_{\hat{K}}, y_{\hat{V}}$ to prove that $\hat{Q} = Q \cdot P$, $\hat{K} = K \cdot P$ and $\hat{V} = V \cdot P$.
(11) $\mathcal{V}$: At the end of the protocol, the verifier ends up with evaluation claims of the MLEs of K, Q, V and P. It locally validates the claims of P and uses the rest of the claims to verify the next step.
(12) $\mathcal{P}$-$\mathcal{V}$: Invoke the PIOP for batched matrix multiplication to prove that $Q = X \cdot W_Q$, $K = X \cdot W_K$ and $V = X \cdot W_V$ using the evaluation claims y_Q, y_K and y_V.
(13) $\mathcal{V}$: Ends up with three evaluation claims of f_X and one evaluation claim for each f_{W_Q}, f_{W_K} and f_{W_V} and uses $[[f_X]]$, $[[f_{W_Q}]]$, $[[f_{W_K}]]$ and $[[f_{W_V}]]$ to check their validity.

Theorem 15. ***Protocol 16*** *is a PIOP for $\mathcal{R}_{MHA}$ with $k_{rbr} = \epsilon_{rbr} = \max(\frac{h \log sd}{|\mathbb{F}|}, \epsilon_{sum}^{Batch}, \epsilon_{Softmax})$ (knowledge) round-by-round soundness error. In addition, it has a prover complexity of $O(h \cdot s^2 + s \cdot d)$ field operations and a proof size and verification time of $O(h \log s + \log^2 s))$.*

Complexity Analysis. Regarding prover time, observe that step (2) requires $O(s \cdot d)$ field operations (Theorem 6), step (4) requires $O(\sum_{i \in [h]}(2s \cdot \tilde{d}_k)) \approx O(s \cdot d)$ field operations (Theorem 6), step (6) requires $O(hs^2)$ field operations (Theorem A.3.1), step (8) requires $O(\sum_{i \in [h]}(2s \cdot \tilde{d}_k)) \approx O(\sum_{i \in [h]}(s \cdot d))$ (Theorem 6) and, finally, steps (10),(12) require $O(s \cdot d)$ field operations (Theorem 6). So, the total prover complexity is $O(h \cdot s^2 + s \cdot d)$. Likewise, we argue about proof size and verification time which are dominated by the PIOP for batch softmax with $O(h \log s + \log^2 s)$ proof size and verification time.

Round-By-Round (Knowledge) Soundness. Assume that $\mathbf{tr}_0 = (\mathbb{x}, \{[[f_{\mathbf{T}_{s,i}}]]\}_{i \in [h]})$ is in a doomed state, that is, there exists $i \in [h]$ such that $\mathbf{Y}_i \neq \text{Softmax}(Q_i K_i^T / \sqrt{d_k})$. Then, for the transcript $\mathbf{tr}_1 = (\mathbf{tr}_0, \mathbf{tr}_{sum})$ it must hold that $\Pr[\mathbf{tr}_1 \notin \mathcal{D}] \leq \epsilon_{sum}$. To prove this, we consider two cases (since $\mathbf{tr}_0$ is doomed):

(1) $\mathbf{Y} \neq \hat{\mathbf{Y}} \cdot \mathbf{P}_1^T$.
(2) $\exists i \in [h]$ such that $\hat{\mathbf{Y}}_i$ has been computed incorrectly.

In case (1) the prover enters the sumcheck protocol in a doomed state and thus can leave it only with probability ϵ_{sum}. In case (2), the prover stays in a doomed state with probability 1.

Next, assume that $\mathbf{tr}_1 \in \mathcal{D}$. Then, for the transcript $\mathbf{tr}_2 = (\mathbf{tr}_1, \{y_{\hat{Y}_i}\}_{i \in [h]})$ it should hold that $\Pr[\mathbf{tr}_2 \notin \mathcal{D}] \leq \frac{h \log sd}{|\mathbb{F}|}$. To prove this consider the two cases (since $\mathbf{tr}_1$ is doomed):

(1) There exists $i \in [h]$ such that $\hat{\mathbf{Y}}_i \neq \mathbf{T}_{s,i} \cdot \mathbf{V}_i$.
(2) $\mathbf{T}_{s,i}$ or $\mathbf{V}_i$ have been computed incorrectly.

In case (1) the prover enters a non-doomed state if it gets lucky and $y_{\mathbf{Y}_i}$ equals to the evaluation of the MLE of $\mathbf{T}_{s,i} \cdot \mathbf{V}_i$. Due to the Schwartz-Zippel Lemma this happens only with probability $\frac{\log sd}{|\mathbb{F}|}$. By taking the union bound of all heads we get the desired bound. As for case (2), the prover stays in a doomed state with probability 1.

Next, assume that $\mathbf{tr}_2 \in \mathcal{D}$. Then, for the transcript $\mathbf{tr}_3 = (\mathbf{tr}_2, \mathbf{tr}_{sum}^{Batch})$ it should hold that $\Pr[\mathbf{tr}_3 \notin \mathcal{D}] \leq \epsilon_{sum}$. To prove this, we consider the two cases as in the previous transcript $\mathbf{tr}_2$. In case (1) the prover enters the sumcheck protocol in a doomed state and thus it can only leave that state with probability ϵ_{sum}. In case (2) the prover stays in a doomed state with probability 1.

Subsequently, assume that $\mathbf{tr}_3 \in \mathcal{D}$. Then, for the transcript $\mathbf{tr}_4 = (\mathbf{tr}_3, \mathbf{tr}_{Softmax})$ it holds that $\Pr[\mathbf{tr}_4 \notin \mathcal{D}] \leq \epsilon_{Softmax}$. To prove this, we consider the following cases (since $\mathbf{tr}_3$ is doomed):

(1) There exists $i \in [h]$ such that $\mathbf{T}_{s_i} \neq \text{Softmax}(\mathbf{T}_{p,i})$.
(2) There exists $i \in [h]$ such that $\mathbf{T}_{p,i}$ or $\hat{\mathbf{V}}_i$ have been computed incorrectly.

In case (1) the prover enters the PIOP for softmax in a doomed state, which can leave only with probability $\epsilon_{Softmax}$. In case (2) the prover stays in the doomed state with probability 1.

Now, assume that $\mathbf{tr}_4 \in \mathcal{D}$. Then, for the transcript $\mathbf{tr}_5 = (\mathbf{tr}_4, \mathbf{tr}_{sum}^{Batch})$ it must hold $\Pr[\mathbf{tr}_5 \notin \mathcal{D}] \leq \epsilon_{sum}^{Batch}$. To prove this, we consider the following cases (since $\mathbf{tr}_4$ is doomed):

(1) There exists $i \in [h]$ such that $\mathbf{T}_{p,i} \neq \hat{\mathbf{Q}}_i \hat{\mathbf{K}}_i^T$ (or more precisely the sum $y_{\mathbf{T}_{p,i}}$ in the underlying sumcheck instance is incorrect)
(2) There exists $i \in [h]$ such that $\hat{\mathbf{V}}_i$ has been computed incorrectly.

In case (1), the prover would enter the batch sumcheck in a doomed state which it can leave only with probability ϵ_{sum}^{Batch}. In case (2), the prover stays in a doomed state with probability 1.

Next, assume that $\mathbf{tr}_5 \in \mathcal{D}$. Then, for the transcript $\mathbf{tr}_6 = (\mathbf{tr}_5, \mathbf{tr}_{sum}^{Batch})$. To prove this, we consider the following cases:

(1) At least one of the matrices $\hat{\mathbf{V}}$, $\hat{\mathbf{K}}$ or $\hat{\mathbf{Q}}$ have been computed incorrectly (or more precisely one of the sums $y_{\mathbf{Q}}$, $y_{\mathbf{K}}$ or $y_{\mathbf{V}}$ are incorrect).
(2) $\mathbf{Q}$, $\mathbf{K}$ or $\mathbf{V}$ have been computed incorrectly.

In the first case, the prover enters the batch sumcheck protocol in a doomed state, and thus, it can leave that state only with probability ϵ_{sum}^{Batch}. In the second case, the prover stays in the doomed state with probability 1.

Finally, assume that $\mathbf{tr}_6 \in \mathcal{D}$ meaning that $\mathbf{Q}$, $\mathbf{K}$ or $\mathbf{V}$ have been computed incorrectly or there exists an evaluation claim that is inconsistent with the given oracle. Then, the final transcript $\mathbf{tr}_7 = (\mathbf{tr}_6, \mathbf{tr}_{sum}^{Batch})$ will remain in the doomed state with probability ϵ_{sum}^{Batch} since the prover would enter the batch sumcheck protocol in a doomed state.

B.9 Proof of Non-Hallucination Gadget

Definition 26 (Attention Score Relation). *We define the relation $\mathcal{R}_{AS}$ to compute the attention score as:*

$$(\mathbb{x}; \mathbb{w}) \in \left(\begin{array}{c} (\{[[f_{\boldsymbol{K}}]], [[f_{\boldsymbol{X}}]], [[f_{\boldsymbol{Y}}]], [[f_{\boldsymbol{T}^{\log_2}}]], \text{score}\}; \\ \{\boldsymbol{K}, \boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{T}^{\log_2}\}) : \\ \boldsymbol{X}_{j,i} = \boldsymbol{K}_{j,j,i}, \ \forall j \in [s], i \in [h], \\ ([[f_{\boldsymbol{T}^{\log_2}}]], [[f_{\boldsymbol{X}}]], [[f_{\boldsymbol{Y}}]]; \boldsymbol{T}^{\log_2}, \boldsymbol{X}, \boldsymbol{Y}) \in \mathcal{R}_{lkp}^{\log_2}, \\ \text{score} = \sum_{j \in [s]} \sum_{i \in [h]} (\mathsf{P} \times \log_2(\mathsf{sf}_{\boldsymbol{X}}) + \boldsymbol{Y}[j, i]) \end{array} \right),$$

where:

- $\boldsymbol{K} \in \mathbb{F}^{s \times s \times h}$, score*: the kernel similarity matrix, and its attention score.*
- $\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{T}^{\log_2}$*: The diagonal of each head in the tensor $\boldsymbol{K}$, the logarithmic result of $\boldsymbol{X}$, and the lookup table. Let* sf' *be the minimum possible scaling factor, and* P *denotes the precision*

scale that preserves the accuracy of the logarithmic operation. $\boldsymbol{T}^{\log_2}$ *is defined as follows:*

$$T[x] = \begin{cases} 0 & x = 0\,, \\ \text{round}(\log_2(x) \cdot \text{P}) & \forall x \in (0, 1/\text{sf}')\,, \end{cases},$$

- score *is the attention score,* $\text{sf}_{\boldsymbol{X}}$ *denotes the quantization scaling factor of* $\boldsymbol{X}$.
- $[[f_{\boldsymbol{K}}]]$, $[[f_{\boldsymbol{X}}]]$, $[[f_{\boldsymbol{Y}}]]$, $[[f_{\boldsymbol{T}^{\log_2}}]]$*: the MLE oracles of* $\boldsymbol{K}$, $\boldsymbol{X}$, $\boldsymbol{Y}$, $\boldsymbol{T}^{\log_2}$ *respectively.*

Protocol Description. We describe the relation $\mathcal{R}_{\text{AS}}$ with three PIOPs.

(1) In the first PIOP, we extract the diagonal of the kernel similarity by the following equation

$$f_{\boldsymbol{X}}(\mathbf{j}^*, \mathbf{i}^*) = \sum_{\substack{\mathbf{j}'_1, \mathbf{j}'_2 \in \{0,1\}^{\log s} \\ \mathbf{i}' \in \{0,1\}^{\log h}}} f_{\text{sel}}(\mathbf{j}^*, \mathbf{i}^*; \mathbf{j}'_1, \mathbf{j}'_2, \mathbf{i}') f_{\boldsymbol{K}}(\mathbf{j}'_1, \mathbf{j}'_2, \mathbf{i}') \tag{1}$$

where the selector $\text{sel}(\mathbf{j}^*, \mathbf{i}^*; \mathbf{j}'_1, \mathbf{j}'_2, \mathbf{i}')$ is defined as follows:

$$f_{\text{sel}}(\mathbf{j}^*, \mathbf{i}^*; \mathbf{j}'_1, \mathbf{j}'_2, \mathbf{i}') = \begin{cases} \beta(\mathbf{j}^*, \mathbf{i}^*; \mathbf{j}'_1, \mathbf{i}'), & \text{for } \mathbf{j}'_1 = \mathbf{j}'_2 \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

The evaluation can be computed as follows:

$$\begin{aligned} &f_{\text{sel}}(\mathbf{r}^*_j, \mathbf{r}^*_i; \mathbf{r}'_{j_1}, \mathbf{r}'_{j_2}, \mathbf{r}'_i) \\ &= \prod_{k=1}^{\log s} ((1 - r^*_{j,k})(1 - r'_{j_1,k})(1 - r'_{j_2,k}) + r^*_{j,k} r'_{j_1,k} r'_{j_2,k}) \\ &\cdot \prod_{\ell=1}^{\log h} ((1 - r^*_{i,\ell})(1 - r_{i',\ell}) + r^*_{i,\ell} r_{i,\ell}) \end{aligned}$$

(2) In the second PIOP, we compute the attention score. Given the input sequence of the LLM model, the shape of a certain LLM layer is fixed, so we use a summation instead of the mean for simplicity. With the logarithmic result function $f_{\boldsymbol{Y}}$ there is:

$$\begin{aligned} \text{score} &= \sum_{j \in [s], i \in [h]} \text{P} \cdot \log_2(\text{sf}_{\mathbf{X}} \cdot \mathbf{X}[j, i]) \\ &= \sum_{\substack{\mathbf{j} \in \{0,1\}^{\log s} \\ \mathbf{i} \in \{0,1\}^{\log h}}} f_{<s}(\mathbf{j}) f_{<h}(\mathbf{i}) (\text{P} \cdot \log_2(\text{sf}_{\mathbf{X}}) + f_{\boldsymbol{Y}}(j, i)) \end{aligned} \tag{3}$$

(3) Finally, we leverage a lookup PIOP to prove the logarithmic operations.

Multi-Head Attention Score

(1) $\mathcal{P}$: Prepare all witness data required to prove $\mathcal{R}^{\log_2}_{\text{lkp}}$ and $\mathcal{R}_{\text{is-zero}}$ and send all required oracles to $\mathcal{V}$.
(2) $\mathcal{P}$-$\mathcal{V}$: Let $\mathbf{Y} \in \mathbb{F}^{s \times h}$ be a vector such that $\mathbf{Y}[j, i] = \log_2(\mathbf{X}[j, i])$. Both parties interact following the lookup protocol to prove that $\{(\mathbf{X}[j, i], \mathbf{Y}[j, i])\}_{j \in [s], i \in [h]}$ is a subset of $\{(x, \mathbf{T}[x])\}_{x \in [0, 1/\text{sf}')}$, where sf' is the minimum possible scaling factor.
(3) $\mathcal{P}$-$\mathcal{V}$: Let y^* be the evaluation claim of $f_{\boldsymbol{Y}}$ produced at the final round of the lookup protocol and score be the attention score multiplied by P. Both parties interact following the batched sumcheck protocol for the following equations:

$$\begin{aligned} \text{score} &= \sum_{\substack{\mathbf{j} \in \{0,1\}^{\log s}, \\ \mathbf{i} \in \{0,1\}^{\log h}}} f_{<s}(\mathbf{j}) f_{<h}(\mathbf{i}) \\ &\qquad (\text{P} \cdot \log_2(\text{sf}_X) + f_{\boldsymbol{Y}}(\mathbf{j}, \mathbf{i}))\,, \\ y^* &= \sum_{\substack{\mathbf{j} \in \{0,1\}^{\log s}, \\ \mathbf{i} \in \{0,1\}^{\log h}}} \beta(\mathbf{r}^*_j, \mathbf{r}^*_i; \mathbf{j}, \mathbf{i}) f_{\boldsymbol{Y}}(\mathbf{j}, \mathbf{i}) \end{aligned}$$

(4) $\mathcal{V}$: Let $y = f_{\boldsymbol{Y}}(\mathbf{r}_j, \mathbf{r}_i)$ be the evaluation produced at the previous sumcheck protocol. Use $[[f_{\boldsymbol{Y}}]]$ to validate the correctness y. Compute $f_{<s}(\mathbf{r}_j)$ and $f_{<h}(\mathbf{r}_i)$ locally.
(5) $\mathcal{P}$-$\mathcal{V}$: Let $x^* = f_{\boldsymbol{X}}(\mathbf{r}^*_j, \mathbf{r}^*_i)$ be the evaluation claim produced at the final round of the lookup protocol. Both the prover and the verifier run the sumcheck protocol for the following equation:

$$x^* = \sum_{\substack{\mathbf{j}'_1, \mathbf{j}'_2 \in \{0,1\}^{\log s}, \\ \mathbf{i}' \in \{0,1\}^{\log h}}} \text{sel}(\mathbf{r}^*_j, \mathbf{r}^*_i; \mathbf{j}'_1, \mathbf{j}'_2, \mathbf{i}') f_{\boldsymbol{K}}(\mathbf{j}'_1, \mathbf{j}'_2, \mathbf{i}') \tag{4}$$

(6) $\mathcal{V}$: Let $k' = f_{\boldsymbol{K}}(\mathbf{r}'_{j_1}, \mathbf{r}'_{j_2}, \mathbf{r}'_i)$ be the evaluation claim produced at the final round of the previous sumcheck protocol. Use $[[f_{\boldsymbol{K}}]]$ to validate the correctness k', and compute $\text{sel}(\mathbf{r}^*_j, \mathbf{r}^*_i; \mathbf{r}'_{j_1}, \mathbf{r}'_{j_2}, \mathbf{r}'_i)$ locally.

C Details Related to Accuracy and Performance

Proving Requantization: To keep lookup table size to a minimum, DeepProve first multiplies by a fixed point multiplier $\epsilon \cdot 2^f$, adds a rounding constant and then decomposes the input into chunks to be passed as the input to lookup tables. These chunks correspond to either the output value or the parts that are right-shifted away. Without loss of generality we can assume that we decompose in such a way that the values in each chunk of the decomposition are k-bit integers. As before the size of the right shift is $f + n$, thus we have $t := \lceil (f + n)/k \rceil$ chunks that are shifted away. If k does not divide $f + n$ exactly the last of these chunks should be of size r, the remainder when $f + n$ is divided by k. If we write $\text{In}(\bar{x})$ for the MLE of the input, $\text{Out}(\bar{x})$ for the MLE of the output and $\text{Shifted}_i(\bar{x})$ for each of the chunks that are right shifted away the prover and verifier check the equation

$$\epsilon \cdot 2^f \cdot \text{In}(\bar{x}) + 2^{f+n-1} = 2^{f+n} \cdot \text{Out}(\bar{x}) + \sum_{i=1}^{t} 2^{k(i-1)} \cdot \text{Shifted}_i(\bar{x}).$$

with $\tilde{In}(\bar{x})$ the MLE representing the input tensor, and $\tilde{Out}(\bar{x})$ the MLE representing the output tensor.

In addition, for $i \in [1, t-1]$ the lookup argument is used to perform a range check enforcing that $\text{Shifted}(\bar{X}) \in [0, 2^k - 1]$ and when $i = t$ it enforces $2^{k-r} \cdot \text{Shifted}_t(\bar{x}) \in [0, 2^k - 1]$ for all $x \in \mathcal{B}$ (here $\mathcal{B}$ is the boolean hypercube). The $Shifted_i(\bar{x})$ represents the MLE of the *chunked* and *shifted* part after applying the requantization operation. $f + n = t * q$ is the number of chunks, so each chunk is the same size as the quantization bit range and thus can be range checked using the same lookup table across the whole proving system.

Proving clamping. The previous section describes how to prove the rescaling and the right shift, but the values of $\text{Out}(\bar{x})$ are

not guaranteed to be k-bit integers and so they must be provably clamped. The simplest solution would be to have another lookup table that takes as input $\text{Out}(\bar{x})$ and outputs $\text{Clamp}(\bar{x})$. In practice though this clamping lookup table is too large and so `DeepProve` uses two different approaches based on whether the value is expected to be signed or not.

Unsigned clamping. When values are known to all be of the same unsigned, then the rescaled input is further decomposed as

$$\epsilon \cdot 2^f \cdot \text{In}(\bar{x}) + 2^{f+n-1} = \sum_{j=1}^{s} 2^{f+n+k\cdot j} \cdot \text{ZeroIn}_j(\bar{x}) + 2^{f+n} \cdot \text{Value}(\bar{x}) + \sum_{i=1}^{t} 2^{k(i-1)} \cdot \text{Shifted}_i(\bar{x}).$$

Here the $\text{ZeroIn}_j(\bar{x})$ are paired with multilinear polynomials $\text{ZeroOut}_j(\bar{x})$ where

$$\text{ZeroOut}_j(y) := \begin{cases} 1, & \text{if } \text{ZeroIn}_j(y) = 0, \\ 0, & \text{if } \text{ZeroIn}_j(y) \neq 0. \end{cases}$$

Then the prover and verifier check via Sumcheck that

$$\text{Out}(\bar{r}) = \sum_{b \in \mathcal{B}} \text{eq}(\bar{r}, b) \cdot \left(\prod_{j=1}^{s} \text{ZeroOut}_j(b) \cdot \text{Value}(b) + \left(1 - \prod_{j=1}^{s} \text{ZeroOut}_j(b)\right) \cdot \text{ClampingMax} \right).$$

Signed Clamping: When the sign of the outputs varies the rescaled input is further decomposed as

$$\epsilon \cdot 2^f \cdot \text{In}(\bar{x}) + 2^{f+n-1} = 2^{f+n+k\cdot s} \cdot \text{SgnIn}(\bar{x}) + \sum_{j=1}^{s-1} 2^{f+n+k\cdot j} \cdot \text{ZeroIn}_j(\bar{x}) + 2^{f+n} \cdot \text{Value}(\bar{x}) + \sum_{i=1}^{t} 2^{k(i-1)} \cdot \text{Shifted}_i(\bar{x}).$$

Here the $\text{ZeroIn}_j(\bar{x})$ are as before, but $\text{SgnIn}(\bar{x})$ is paired with a multilinear polynomials $\text{SgnOut}(\bar{x})$ where

$$\text{SgnOut}(y) := \begin{cases} 0, & \text{if } \text{SgnIn}(y) = 0, \\ 1, & \text{if } \text{SgnIn}(y) > 0, \\ -1, & \text{if } \text{SgnIn}(y) < 0. \end{cases}$$

$$\text{Out}(\bar{r}) = \sum_{b \in \mathcal{B}} \text{eq}(\bar{r}, b) \cdot \Bigg((1 - \text{SgnOut}^2(b)) \cdot \prod_{j=1}^{s-1} \text{ZeroOut}_j(b) \cdot \text{Value}(b) + \left(1 - (1 - \text{SgnOut}^2(b)) \cdot \prod_{j=1}^{s} \text{ZeroOut}_j(b)\right) \cdot \left(\frac{\text{CMax} - \text{CMin}}{2} \cdot \text{SgnOut}(b) + \frac{\text{CMax} + \text{CMin}}{2} \right) \Bigg).$$

Norm layers relying on zkLLM technique. Proving norms is challenging because small errors accumulate during the sum on the variance and expected value. This, in turn, drastically reduces the accuracy. To overcome that issue, we rely on similar techniques with the ones utilized by zkLLM [SLZ24] for softmax to prove compute and then prove norm layers.

For example during RMSNorm, for some input tensor X we normalise along the final dimension. For the sake of example assume that $X \in \mathbb{R}^{m \times n}$ and write X_q for the quantized tensor with scaling factor s_1. For each $1 \leq j \leq m$ the RMSNorm operation rescales X_j by the value

$$d_j := \sqrt{\frac{\sum_{i=1}^{n} X_{j,i}^2}{n}}.$$

If we were to prove this normally we would compute the element wise square of X_q, pass X_q^2 to a lookup argument to prove the division by n, square root and inversion, and then multiply X_q element wise by this inverse square root tensor. This has two problems, it incurs accuracy loss during the inverse square root lookup and the output of the layer has to undergo the requantization procedure immediately afterwards. Instead to retain as much accuracy as possible we utilise an adapted version of our requantization procedure and then enforce that the output of the RMSNorm satisfies a certain error bound. Observe that by starting with X_q we can obtain X again by multiplying by the scalar s_1. Then for each j the prover computes the value d_j as above. If the output of the RMSNorm should be quantized with the scaling factor s_2 then the prover computes the list of rescaling factors.

$$\frac{s_1}{d_j \cdot s_2} = 2^{-l_j} \cdot \epsilon_j, \text{ with } l_j \in \mathbb{N}, \epsilon_j \in [2^{-1}, 1].$$

Again some there is some predetermined precision factor $f \in \mathbb{N}$ and the prover determines what the maximum right shift will be by setting it as $r := \max_{j \in m} l_j + f$. The prover commits to the shift value r and the list of values $\{\text{Round}(\epsilon_j \cdot 2^r)\}_{j=1}^{m} \in \mathbb{F}_p^m$. Then just as in requantization the output is computed as

$$\text{RMSNorm}(X_{q,j}) = \left\lfloor \frac{X_{q,j} \cdot \epsilon_j \cdot 2^r + 2^{r-1}}{2^r} \right\rfloor.$$

Lemma 1. ***RMSNorm bound 1****: Let X_q, s_1 and s_2 be as above, write N_q for the output of the quantized RMSNorm evaluation procedure, if*

$$N_q = \text{Round}\left(\frac{\text{RMSNorm}(s_1 \cdot X_q)}{s_2} \right)$$

then

$$\left| \frac{n}{s_2^2} - \sum_{i=1}^{n} N_{q,j,i}^2 \right| \leq \frac{n}{s_2} + \frac{n}{4}$$

for all $1 \leq j \leq m$.

Proof. Assume N_q is actually the quantized output of RMSNorm then we have

$$s_2 \cdot N_{q,j,i} = \left(\frac{s_1 \cdot X_{q,j,i}}{d_j} \pm \frac{s_2}{2} \right),$$

with

$$d_j := \sqrt{\frac{\sum_{i=1}^{n} (s_1 \cdot X_{q,j,i})^2}{n}}.$$

Then we can establish the bound as follows,

$$\left|n - \sum_{i=1}^{n}(s_2 \cdot N_{q,j,i})^2\right| = \left|n - \sum_{i=1}^{n}\left(\frac{(s_1 \cdot X_{q,j,i})}{d_j} \pm \frac{s_2}{2}\right)^2\right|$$
$$= \left|n - \sum_{i=1}^{n}\left(\frac{(s_1 \cdot X_{q,j,i})^2}{d_j^2} \pm \frac{s_2 \cdot s_1 \cdot X_{q,i,j}}{d_j} + \frac{s_2^2}{4}\right)\right|$$
$$= \left|n - \sum_{i=1}^{n}\frac{(s_1 \cdot X_{q,j,i})^2}{d_j^2} \pm \sum_{i=1}^{n}\frac{s_2 \cdot s_1 \cdot X_{q,i,j}}{d_j} + \frac{n \cdot s_2^2}{4}\right|,$$

but by construction

$$\sum_{i=1}^{n}\frac{(s_1 \cdot X_{q,j,i})^2}{d_j^2} = n,$$

so this simplifies further to

$$\left|n - \sum_{i=1}^{n}(s_2 \cdot N_{q,j,i})^2\right| = \left|\pm\sum_{i=1}^{n}\frac{s_2 \cdot s_1 \cdot X_{q,i,j}}{d_j} + \frac{n \cdot s_2^2}{4}\right|$$
$$= \left|\pm\frac{s_2}{d_j}\sum_{i=1}^{n}(s_1 \cdot X_{q,j,i}) + \frac{n \cdot s_2^2}{4}\right|$$
$$\leq \frac{s_2}{d_j}\sum_{i=1}^{n}|(s_1 \cdot X_{q,j,i})| + \frac{n \cdot s_2^2}{4}.$$

We notice that the sum remaining on the right hand side is just the L_1-norm of the jth row, so we can use the property that if $y \in \mathbb{R}^k$ then $\|y\|_1 \leq \sqrt{k} \cdot \|y\|_2$ yielding

$$\left|n - \sum_{i=1}^{n}(s_2 \cdot N_{q,j,i})^2\right| \leq \frac{s_2 \cdot \sqrt{n}}{d_j}\sqrt{\sum_{i=1}^{n}(s_1 \cdot X_{q,j,i})^2} + \frac{n \cdot s_2^2}{4}$$
$$\leq s_2 \cdot n + \frac{n \cdot s_2^2}{4} \text{ (by the definition of } d_j).$$

Dividing both sides through by s_2^2 we obtain the desired result. □

Lemma 2. ***RMSNorm bound 2****: Let $N_{q,j}$ be computed as* $\text{Round}(\alpha_j \cdot X_{q,j})$ *for some $\alpha_j \in \mathbb{R}$. Then if N_q satisfies*

$$\left|\frac{n}{s_2^2} - \sum_{i=1}^{n}N_{q,j,i}^2\right| \leq \frac{n}{s_2} + \frac{n}{4}$$

for all $1 \leq j \leq m$ we must have that

$$\alpha_j = \frac{s_1 \cdot \sqrt{n}}{s_2 \cdot \sqrt{\sum_{i=1}^{n}(s_1 \cdot X_{q,j,i})^2}} \cdot \left(1 \pm \left(s_2 + \frac{s_2^2}{2}\right)\right).$$

Hence α_j is the normalising factor up to some error determined by the output scaling factor s_2.

Proof. Since $N_{q,j,i} = \text{Round}(\alpha_j \cdot X_{q,j,i})$, we can write:

$$N_{q,j,i} = \alpha_j \cdot X_{q,j,i} + e_{j,i}$$

where $e_{j,i} \in [-1/2, 1/2]$ is the rounding error.

Squaring both sides:

$$N_{q,j,i}^2 = \alpha_j^2 X_{q,j,i}^2 + 2\alpha_j X_{q,j,i} e_{j,i} + e_{j,i}^2$$

Summing over all i:

$$\sum_{i=1}^{n}N_{q,j,i}^2 = \alpha_j^2\sum_{i=1}^{n}X_{q,j,i}^2 + 2\alpha_j\sum_{i=1}^{n}X_{q,j,i} \cdot e_{j,i} + \sum_{i=1}^{n}e_{j,i}^2$$

Let $A = \sum_{i=1}^{n}X_{q,j,i}^2$, $B = \sum_{i=1}^{n}X_{q,j,i}e_{j,i}$, and $C = \sum_{i=1}^{n}e_{j,i}^2$.

Then:

$$\sum_{i=1}^{n}N_{q,j,i}^2 = \alpha_j^2 A + 2\alpha_j \cdot B + C \tag{5}$$

Bounding the error terms:

Since $|e_{j,i}| \leq 1/2$:

- $|B| = \left|\sum_{i=1}^{n}X_{q,j,i}e_{j,i}\right| \leq \sum_{i=1}^{n}|X_{q,j,i}| \cdot |e_{j,i}| \leq \frac{1}{2}\sum_{i=1}^{n}|X_{q,j,i}|$
- $C = \sum_{i=1}^{n}e_{j,i}^2 \leq \sum_{i=1}^{n}\frac{1}{4} = \frac{n}{4}$

Using the constraint:

From $\left|\frac{n}{s_2^2} - \sum_{i=1}^{n}N_{q,j,i}^2\right| \leq \frac{n}{s_2} + \frac{n}{4}$, we have:

$$\frac{n}{s_2^2} - \frac{n}{s_2} - \frac{n}{4} \leq \sum_{i=1}^{n}N_{q,j,i}^2 \leq \frac{n}{s_2^2} + \frac{n}{s_2} + \frac{n}{4}$$

Combining with equation (5):

$$\frac{n}{s_2^2} - \frac{n}{s_2} - \frac{n}{4} \leq \alpha_j^2 A + 2\alpha B + C \leq \frac{n}{s_2^2} + \frac{n}{s_2} + \frac{n}{4}$$

Since $C \leq \frac{n}{4}$, we can write:

$$\frac{n}{s_2^2} - \frac{n}{s_2} - \frac{n}{2} \leq \alpha_j^2 A + 2\alpha B \leq \frac{n}{s_2^2} + \frac{n}{s_2}$$

This gives us:

$$\left|\alpha_j^2 A - \frac{n}{s_2^2}\right| \leq \frac{n}{s_2} + \frac{n}{2} + 2|\alpha_j||B|$$

Error bound:

To quantify the error, note that from $\left|\alpha_j^2 A - \frac{n}{s_2^2}\right| \leq \frac{n}{s} + \frac{n}{2} + 2|\alpha_j||B|$:

$$\left|\left(\alpha_j \cdot \sqrt{A} - \frac{\sqrt{n}}{s_2}\right)\left(\alpha_j \cdot \sqrt{A} + \frac{\sqrt{n}}{s_2}\right)\right| \leq \left(\frac{n}{s_2} + \frac{n}{2} + 2|\alpha_j||B|\right).$$

Again using the fact that $|B| \leq \frac{1}{2}\sum_{i=1}^{n}|X_{q,i,j}|$ and the properties of the norms we get that

$$\left|\left(\alpha_j \cdot \sqrt{A} - \frac{\sqrt{n}}{s_2}\right)\left(\alpha_j \cdot \sqrt{A} + \frac{\sqrt{n}}{s_2}\right)\right| \leq \left(\frac{n}{s_2} + \frac{n}{2} + |\alpha_j|\sqrt{n} \cdot \sqrt{A}\right)$$
$$\leq \sqrt{n}\left(\frac{\sqrt{n}}{s_2} + |\alpha_j| \cdot \sqrt{A}\right) + \frac{n}{2}.$$

Since $\sqrt{A}$ and $\sqrt{n}/s_2$ are both positive $|\alpha_j \cdot \sqrt{A} + \sqrt{n}/s_2| = |\alpha| \cdot \sqrt{A} + \sqrt{n}/2$. Dividing both sides of the above by this term gives

$$\left|\left(\alpha_j \cdot \sqrt{A} - \frac{\sqrt{n}}{s_2}\right)\right| \leq \sqrt{n} + \frac{n}{2\left(|\alpha_j| \cdot \sqrt{A} + \frac{\sqrt{n}}{s_2}\right)}$$
$$\leq \sqrt{n}\left(1 + \frac{\sqrt{n}}{2\left(|\alpha_j| \cdot \sqrt{A} + \frac{\sqrt{n}}{s_2}\right)}\right)$$
$$\left|\left(\alpha_j - \frac{\sqrt{n}}{s_2 \cdot \sqrt{A}}\right)\right| \leq \frac{\sqrt{n}}{\sqrt{A}}\left(1 + \frac{\sqrt{n}}{2\left(|\alpha_j| \cdot \sqrt{A} + \frac{\sqrt{n}}{s_2}\right)}\right)$$
$$\leq \frac{\sqrt{n}}{s_2 \cdot \sqrt{A}}\left(s_2 + \frac{s_2^2 \cdot \sqrt{n}}{2\left(s_2 \cdot |\alpha_j| \cdot \sqrt{A} + \sqrt{n}\right)}\right)$$
$$\leq \frac{\sqrt{n}}{s_2 \cdot \sqrt{A}}\left(s_2 + \frac{s_2^2}{2}\right).$$

Removing the absolute value gives

$$\alpha_j = \frac{\sqrt{n}}{s_2 \cdot \sqrt{A}} \left(1 \pm \left(s_2 + \frac{s_2^2}{2}\right)\right).$$

Finally we recall that $A = \sum_{i=1}^{n} X_{q,j,i}^2$ hence the term

$$\frac{\sqrt{n}}{s_2 \cdot \sqrt{A}} = \frac{\sqrt{n}}{s_2 \cdot \sqrt{\sum_{i=1}^{n} X_{q,j,i}^2}} = \frac{s_1\sqrt{n}}{s_2 \cdot \sqrt{\sum_{i=1}^{n} (s_1 \cdot X_{q,j,i})^2}}$$

completing the proof. □

In fact the LayerNorm operation can be viewed as an RMSNorm where the mean of each row is subtracted first, that is

$$\text{LayerNorm}(X) = \text{RMSNorm}(X - \mu)$$

where μ is a tensor that is the row wise mean of X.

To prove LayerNorm we follow a similar procedure but also enforce a bound on the sum of each row. To maintain as much accuracy as possible just like in RMSNorm the prover provides a vector of ϵ_j and commits to a maximum right shift r. In addition the prover commits to a vector $\mu \in \mathbb{F}_p$ with

$$\mu_j = \text{Round}\left(2^r \cdot \frac{\sum_{i=1}^{n} \frac{(s_1 \cdot X_{q,j,i})}{n}}{\text{Var}\left(s_1 \cdot X_{q,j} - \sum_{i=1}^{n} \frac{(s_1 \cdot X_{q,j,i})}{n}\right)}\right).$$

Lemma 3. ***LayerNorm bound**: Let $X \in \mathbb{R}^{m \times n}$ be a matrix, write X_q for the quantized version of X with scaling factor s_1. Let $\mu \in \mathbb{Z}$ and write N_q for the quantized output of the operation*

$$N_{q,j,i} := \text{LayerNorm}(s_1 \cdot X_q)_{j,i} = \left\lfloor \frac{2^r \cdot \epsilon_j \cdot X_{q,j,i} + 2^{r-1} - \mu_j}{2^r} \right\rfloor,$$

with $\mu_j = 2^r \cdot \epsilon_j \sum_{i=1}^{n} \frac{x_{q,j,i}}{n}$ then

$$\left|\sum_{i=1}^{n} N_{q,j,i}\right| \leq \frac{n}{2}.$$

Proof. Without loss of generality we can assume $\epsilon_j = 1$, hence we have

$$|N_{q,j,i}| \leq \left| \frac{2^r \cdot \left(X_{q,j,i} - \sum_{i=1}^{n} \frac{X_{q,j,i}}{n}\right)}{2^r} + \frac{1}{2} \right|.$$

So taking the sum gives

$$\begin{aligned}\left|\sum_{i=1}^{n} N_{q,j,i}\right| &\leq \sum_{i=1}^{n} \left| \cdot \left(X_{q,j,i} - \sum_{i=1}^{n} \frac{X_{q,j,i}}{n}\right) + \frac{1}{2} \right| \\ &\leq \left| \sum_{i=1}^{n} X_{q,j,i} - n \cdot \sum_{i=1}^{n} \frac{X_{q,j,i}}{n} + \frac{n}{2} \right| \\ &\leq \frac{n}{2}.\end{aligned}$$

This completes the proof. □

Lemma 4. *If $N_{q,j,i}$ is calculated as*

$$N_{q,j,i} = X_{q,j,i} - a_j$$

for some constant a_j and the bound

$$\left|\sum_{i=1}^{n} N_{q,j,i}\right| \leq \frac{n}{2}$$

holds, then $|\mu_j - a_j| \leq 1/2$.

Proof. Starting with the bound $\left|\sum_{i=1}^{n} N_{q,j,i}\right| \leq \frac{n}{2}$ we can substitute for $N_{q,j,i}$ yielding

$$\begin{aligned}\sum_{i=1}^{n} \left|X_{q,j,i} - a_j\right| &\leq \frac{n}{2}, \\ \left|\left(\sum_{i=1}^{n} X_{q,j,i}\right) - n \cdot a_j\right| &\leq \frac{n}{2}.\end{aligned}$$

Noticing that $\sum_{i=1}^{n} X_{q,j,i} = n \cdot \mu_j$ the final equation above becomes

$$|n \cdot \mu_j - n \cdot a_j| \leq \frac{n}{2},$$

dividing through by n completes the proof. □

We note that in the proof above we did not make use of the scaling factor for X_q, this is because rescaling then taking the mean is equivalent to taking the mean and then rescaling. Since μ_j is the *quantized* mean value, so $|\mu_j - a_j| \leq \frac{1}{2}$ means that a is the quantized value of the mean.

Avoid requantization when possible. Even if the output of a multiplication between two 8-bit quantized integer gives a 16-bit integer, this may be acceptable in some scenario. A matrix $N \times M$ matrix multiplied by a $M \times P$ matrix can yield at maximum an integer of range $B = 2 * q + log_2(M) + 1$ bits. If $B < |\mathbb{F}|$ or is sufficiently small to still be used during fixed point multiplication for non linear layers, then there is no need to requantize right after. Requantizing incurs accuracy loss and thus avoiding it increases the accuracy. This strategy is even more effective when using a large field $\mathbb{F}$ during proving.