zkPyTorch: A Hierarchical Optimized Compiler for Zero-Knowledge Machine Learning

Tiancheng Xie[1], Tao Lu[1], Zhiyong Fang[1], Siqi Wang[1], Zhenfei Zhang[1], Yongzheng Jia[1], Dawn Song[2], Jiaheng Zhang[3]
[1]Polyhedra Network
[2]UC Berkeley
[3]National University of Singapore

Abstract

As artificial intelligence (AI) becomes increasingly embedded in high-stakes applications such as healthcare, finance, and autonomous systems, ensuring the verifiability of AI computations without compromising sensitive data or proprietary models is crucial. Zero-knowledge machine learning (ZKML) leverages zero-knowledge proofs (ZKPs) to enable the verification of AI model outputs while preserving confidentiality. However, existing ZKML approaches require specialized cryptographic expertise, making them inaccessible to traditional AI developers.

In this paper, we introduce ZKPyTorch, a compiler that seamlessly integrates ML frameworks like PyTorch with ZKP engines like Expander, simplifying the development of ZKML. ZKPyTorch automates the translation of ML operations into optimized ZKP circuits through three key components. First, a ZKP preprocessor converts models into structured computational graphs and injects necessary auxiliary information to facilitate proof generation. Second, a ZKP-friendly quantization module introduces an optimized quantization strategy that reduces computation bit-widths, enabling efficient ZKP execution within smaller finite fields such as M61. Third, a hierarchical ZKP circuit optimizer employs a multi-level optimization framework at model, operation, and circuit levels to improve proof generation efficiency.

We demonstrate ZKPyTorch effectiveness through end-to-end case studies, successfully converting VGG-16 and Llama-3 models from PyTorch, a leading ML framework, into ZKP-compatible circuits recognizable by Expander, a state-of-the-art ZKP engine. Using Expander, we generate zero-knowledge proofs for these models, achieving proof generation for the VGG-16 model in 2.2 seconds per CIFAR-10 image for VGG-16 and 150 seconds per token for Llama-3 inference, improving the practical adoption of ZKML.

1 Introduction

As AI systems increasingly make high-stakes decisions in domains like autonomous vehicles [5], healthcare [1], and finance [14], there is a growing need to verify these computations without compromising sensitive data or proprietary models. The challenge is particularly acute in regulated industries where model transparency is mandated but intellectual property protection is essential. Traditional transparency approaches [3] often necessitate revealing sensitive information, such as proprietary model details, creating a fundamental tension between transparency and confidentiality. Consider the scenario of a hospital employing AI for cancer diagnosis. While the hospital requires assurance of the AI's accuracy and consistency, the AI provider cannot disclose proprietary model details. Similar dilemmas arise in other domains, such as determining liability in autonomous vehicle accidents.

Zero-knowledge machine learning (ZKML) [4, 8, 9, 13, 15, 16, 18, 23, 26] offers a promising solution to this challenge. By leveraging advanced cryptographic techniques known as zero-knowledge proofs (ZKPs) [10], ZKML enables the verification of AI computations without exposing sensitive data, such as proprietary model parameters. ZKPs allow a "prover" to demonstrate the correctness of a computation without revealing the underlying model, enabling "verifiers" to confirm the accuracy of the AI's decisions. However, the complex mechanism of current cryptography poses significant barriers to the widespread adoption of ZKPs by traditional AI developers, who typically work with machine learning frameworks like PyTorch [20]. Achieving rigorous cryptographic security in ZKML requires that each machine-learning operation be meticulously designed using ZKP patterns. This process demands specialized cryptographic expertise, creating a challenge for traditional AI developers and hindering the seamless integration of ZKML into existing AI systems.

In this paper, we present ZKPyTorch, a compiler designed to bridge the gap between machine learning frameworks like PyTorch [20] and ZKP engines like Expander [7], streamlining the ZKML development process. Our compiler enables developers to write standard ML code without the need to learn new ZKP-specific programming patterns. It automatically translates ML operations, such as convolution, matrix multiplication, ReLU, softmax, and attention, into ZKP circuits and applies built-in optimizations for common ZKML patterns, ensuring efficient computational performance. ZKPyTorch mainly consists of three modules to seamlessly integrate

with the widely adopted PyTorch framework: a ZKP preprocessor for ML tasks, a ZKP-friendly ML quantization module, and a hierarchical ZKP circuit optimizer.

ZKP preprocessor for ML tasks. Integrating ZKP into ML frameworks poses significant challenges due to the complexity of computation pipelines. To standardize and formalize intricate ML patterns, ZKPyTorch utilizes the Open Neural Network Exchange (ONNX) format [19] as an intermediate representation, where we enable the structured conversion of ML models into computational graphs. Additionally, ZKPyTorch enhances these graphs by adding edges and nodes to generate auxiliary information required for ZKP circuits. For instance, proving the correctness of a division operation requires not only the quotient as the result but also the remainder as auxiliary data, along with proof that the remainder is smaller than the divisor. Similarly, nonlinear functions like ReLU, softmax, and normalization layers require additional lookup constraints to be ZKP-compatible. ZKPyTorch preprocesses ML models by structuring computations and generating necessary proof-related data.

ZKP-friendly ML quantization. ZKPyTorch contains a ZKP-friendly ML quantization technique to bridge the gap between traditional ML computations, which rely on floating-point operations, and ZKP computations, which operate over finite fields. Previous approaches [17, 23, 26] have used fixed-point representations to emulate floating-point arithmetic. To achieve accuracy comparable to the original ML model, these methods require large bit-width fixed-point numbers, necessitating operations over large finite fields, such as the scalar field of the BN254 curve. To address this challenge, we design novel quantization strategies tailored to the constraints of ZKP systems while maintaining predictive performance. Our optimal quantization method reduces computation bit-widths to fit smaller finite fields like M61, balancing ML accuracy with efficiency.

Hierarchical ZKP circuit optimizer. We introduce a hierarchical optimization framework for translating ML computation graphs into efficient proving circuits, incorporating three levels of optimization to enhance computational efficiency and proof scalability. Model-level optimizations preserve high-level semantics to streamline proof generation. For instance, generating multiple tokens in traditional large language models (LLMs) requires token-by-token computations, but proving their correctness can be optimized into a single ZKP circuit. Primitive operation level optimizations, targeting foundational ML operations like convolution and softmax, embed well-established techniques such as specialized arithmetic circuits, and table lookup circuits to enhance the proving process. Circuit-level optimizations focus on parallelizing ZKP circuits, enabling multi-core execution to significantly reduce proof generation time and efficiently prove large-scale ML models.

End-to-end user cases. With ZKPyTorch, we have seamlessly integrated PyTorch [20], a leading ML framework, with Expander [7], a state-of-the-art ZKP framework, enabling verifiable and privacy-preserving machine learning. We successfully convert VGG-16 [22] and Llama-3 [12] neural networks from PyTorch into ZKP-compatible circuits recognizable by Expander. Using Expander, we generate zero-knowledge proofs for these models. As shown in Table 1, our solution achieves proof generation for the VGG-16 model in just 2.2 seconds per CIFAR-10 [6] image using a single CPU core, and 150 seconds per token for Llama-3 inference. Our results demonstrate that this approach empowers AI developers to build cryptographically secure and verifiable neural networks while significantly reducing development costs. This streamlining accelerates the adoption of ZKML across various domains. Specifically, ZKPyTorch enables the easy deployment of verifiable Machine-Learning-as-a-Service (MLaaS) by integrating PyTorch with Expander, generating ZKPs to ensure inference correctness while preserving model confidentiality. It also facilitates verifiable model valuation, allowing AI stakeholders to assess model accuracy and robustness using ZKPs, ensuring transparent evaluation without exposing proprietary model details.

Table 1. Performance of proof generation using Expander with circuits generated by ZKPyTorch.

Model	# Parameters	Single-Core Performance
VGG-16	15.2 Million	2.2 sec / image
Llama-3	8 Billion	150 sec / token

1.1 Related Work

Zero-knowledge machine learning has evolved significantly since early work on decision trees [26]. Based on this fundamental concept, many efforts [4, 8, 9, 13, 15, 16, 18, 23] have been directed toward developing zero-knowledge proofs (ZKPs) for neural networks, making verifiable deep learning increasingly feasible. The academic foundations of the field were established through several groundbreaking projects that introduced novel constraint systems and proof techniques. ZKCNN [16] pioneered the application of ZKPs to convolutional neural networks (CNNs), demonstrating the practicality of verifiable inference on structured deep learning models. More recent work on scaling deep neural network (DNN) inference [4, 15] has pushed the boundaries further by developing advanced constraint systems capable of handling increasingly large and complex models while maintaining efficiency. Recent research [13, 18, 23] has made proving large language models (LLMs) nearly practical by employing range proofs, lookup proofs, and other advanced techniques to optimize proving efficiency. These efforts primarily focus on enhancing the efficiency of zero-knowledge proofs and addressing the computational bottlenecks inherent in verifying large-scale inference.

Compiler design is critical for expanding the applicability of ZKML, enabling more AI developers to participate in the ecosystem and lowering the barrier to adoption. ZEN [8] introduced compiler optimizations tailored for neural network circuits, focusing on efficient representation and constraint reduction to minimize proof generation costs. Building on this foundation, ZENO [9] further enhances zero-knowledge proof systems by optimizing constraint representation and proof generation, significantly reducing the overhead associated with verifying deep learning models. However, neither of these works directly integrates with modern ML frameworks, limiting accessibility for AI developers who rely on established ecosystems like PyTorch [20] and TensorFlow [24]. Additionally, their approaches are specifically designed for CNN models, making them unsuitable for more complex architectures, such as large language models (LLMs) that involve extensive matrix multiplications and attention mechanisms. Our work introduces a more general compiler that enables direct compilation from PyTorch code to ZKP circuits, providing seamless integration with ML frameworks and supporting a broader range of models beyond CNNs.

2 Background

2.1 Zero-Knowledge Proofs

Zero-knowledge proofs (ZKPs) are a cryptographic primitive that allows a prover to demonstrate the correctness of a computation to a verifier without revealing any secret information [10]. Specifically, given a function F and a target output y, the prover shows that it knows a public value x and a secret value y such that $y = F(x, w)$ while not revealing the secret value w. Here, the function F can describe an arbitrary computation. This property makes ZKPs particularly useful in privacy-preserving applications, such as authentication, blockchain scalability, and confidential transactions.

The workflow of zero-knowledge proof is shown in Figure 1, where a crucial step before the proof generation is to convert the function F to ZKP circuit. In this process, each addition and multiplication in the function F is compiled into a addition gate and a multiplication gate in the circuit, respectively. Besides addition and multiplication, ZKP circuit also supports non-linear operations such as maximum value and square root operations, which are achieved by introducing table lookup gates to constrain calculation results to be in a specific table. For large functions involving millions of operations, the circuit may contain millions of gates. Since proof generation latency is proportional to the number of gates, circuits of this scale can lead to significant computational overhead. Therefore, leveraging efficient circuit compile techniques becomes critical to reducing computational overhead.

The integration of ZKPs into machine learning, known as zero-knowledge machine learning (ZKML), enables the verification of ML model computations while preserving model

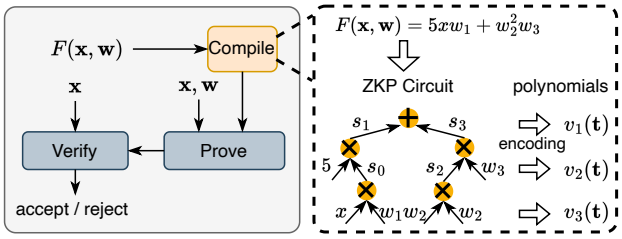

Figure 1. The workflow of zero-knowledge proof.

privacy. In ZKML, the inference process is treated as a function F, where either the model input or the model weights serve as the secret input. Similar to general ZKPs, proof generation first requires compiling the model function F into a ZKP circuit. However, the complexity of ML models makes this compilation workflow challenging. Existing ZKML solutions often demand extensive modifications to ML models built with classical frameworks like PyTorch, posing a barrier for AI developers. To overcome this, new approaches focus on compilation frameworks that automatically convert ML models into ZKP circuits, bridging the gap between AI development and cryptographic proof generation.

2.2 Directed Acyclic Graph

Directed Acyclic Graph (DAG) is a fundamental data structure in graph theory, characterized by a finite set of vertices and directed edges that do not form any cycles. Formally, a DAG is a directed graph $G = (V, E)$, where V is the set of vertices, and $E \subseteq V \times V$ is the set of directed edges, such that there exists no sequence of edges forming a cycle, i.e., there is no path $(v_1, v_2, \ldots, v_k)$ where $v_1 = v_k$.

Traditional ZKML compilers, such as ZENO [9], represent the machine learning (ML) function as a one-dimensional list, where the output of each layer directly serves as the input to the next layer. While this approach works for simple deep learning models with sequential architectures, it is insufficient for general ML models, especially those with complex structures such as residual layers, which connect outputs from multiple layers to the next layer.

In our compiler, we utilize a DAG to represent the computation process of ML models, where nodes V correspond to the primitive operations used in ML, such as matrix multiplications and element-wise activations, while directed edges E represent the data flow during model inference, indicating dependencies between operations. This DAG-based representation captures dependencies between different layers more accurately, enabling support for complex network architectures that involve non-sequential connections.

2.3 Machine Learning Quantization

Machine learning quantization is a crucial technique for reducing the storage and computational complexity of deep

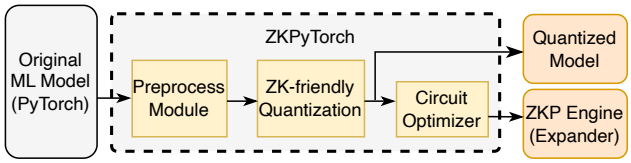

Figure 2. A high-level overview of ZKPyTorch.

learning models, making them more efficient for deployment on resource-constrained devices. Quantization approximates high-precision floating-point representations with lower-precision numerical formats, enabling faster inference while maintaining acceptable model accuracy. In ZKML setting, quantization plays a critical role in bridging the gap between ML models, which typically operate using floating-point arithmetic, and ZKP schemes, which perform computations in a finite field. Therefore, directly handling floating-point numbers is infeasible due to the high computational overhead required for native floating-point support in ZK circuits.

To address this, previous ZKML schemes [17, 23, 26] employ fixed-point numbers to approximate floating-point values. However, due to the wide dynamic range of floating-point numbers, fixed-point representations require a significantly larger bit width for accurate representation. For instance, the standard float32 format can represent positive numbers ranging from approximately 2^{-126} to 2^{127}, whereas 32-bit fixed-point numbers are limited to a range of 2^{-16} to 2^{16}. This disparity forces fixed-point representations to use additional bits to emulate floating-point numbers, thereby necessitating ZKML to operate over large finite fields, such as the scalar field of the BN254 curve, which considerably reduces efficiency in proof generation. Therefore, developing a ZKP-friendly quantization scheme is crucial to improving efficiency while maintaining model accuracy.

3 ZKPyTorch: A Hierarchical Optimized Compiler for ZKML

In this section, we introduce ZKPyTorch, a hierarchically optimized compiler that seamlessly integrates with the widely used PyTorch framework, removing traditional barriers and streamlining the ZKML development process. With ZKPyTorch, developers can write standard PyTorch code [20] without needing to learn ZKP-specific programming patterns. ZKPyTorch automatically translates PyTorch operations into ZKP circuits while applying built-in optimizations for common ML patterns, ensuring efficient memory usage and computational performance.

3.1 Architecture Overview

To bridge the gap between ML frameworks like PyTorch [20] and ZKP engines like Expander [7], our ZKML compiler comprises three key components. First, the preprocessing module formalizes complex ML computations as a directed acyclic graph (DAG) and augments it with additional nodes and edges to generate auxiliary information for zero-knowledge proofs, ensuring correctness and completeness. Second, we introduce a ZKP-friendly ML quantization module, which reconciles the differences between traditional ML frameworks operating on floating-point numbers and ZKP engines, which function over finite fields with modular arithmetic constraints. This module optimizes numerical precision while preserving model accuracy. Third, we present a hierarchical optimization framework that translates ML computation graphs into efficient ZKP circuits. Figure 2 provides a high-level overview of our ZKML compiler, with detailed technical explanations discussed in the following sections.

3.2 Preprocessing Module

This section present our preprocessing module for ZKML compiler. Given the complexity of ML frameworks like PyTorch, managing the entire computation pipeline for ZKPs integration presents substantial challenges. First, traditional ML frameworks support a wide range of operations, allowing users to define custom operations based on their needs. However, designing ZKP circuits for thousands of potential operations individually is impractical. Second, existing ZKML compilers like ZENO [9] are designed for simple deep learning models with sequential architectures. Their approaches are insufficient for more complex ML models, particularly those with intricate structures like Transformers. Third, there are fundamental discrepancies between traditional ML computations and ZKP circuits. For instance, proving the correctness of a division operation requires not only the quotient as the computational result but also the remainder as auxiliary data. These challenges lead us to introduce our preprocessing module. Its overall workflow are shown in Figure 3.

To accommodate the diverse operations in traditional ML frameworks, we standardize ML workflows using primitive operations. To achieve this, we leverage the Open Neural Network Exchange (ONNX) format [19], which formalizes ML primitive operations into a unified representation. ONNX acts as an intermediate representation, capturing various ML operations in a consistent manner. Thereby, we ensure that various ML operations, such as convolutions, activations, matrix multiplications, and pooling layers, are represented uniformly. In addition, ONNX supports various ML platforms, including PyTorch [20], TensorFlow [24], Scikit-Learn [21], and Caffe2 [2]. This opens up the potential to extend our ZKPyTorch compiler beyond PyTorch to other ML platforms.

Next, we employ directed acyclic graphs (DAGs) to represent computation processes in ML models. Figure 3 illustrates

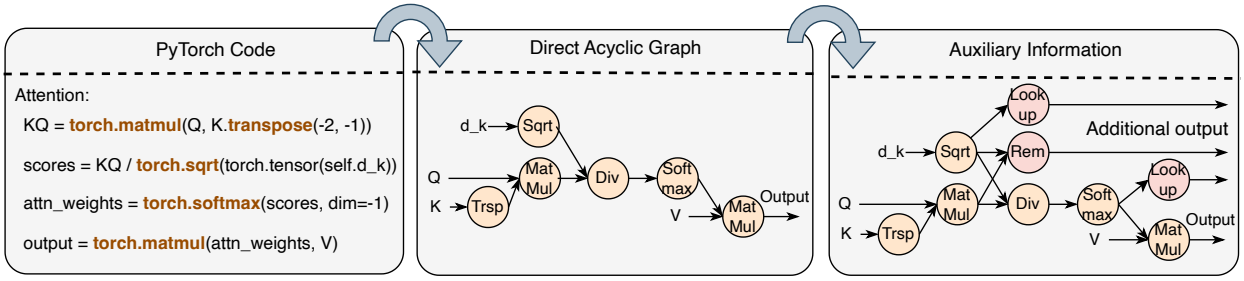

Figure 3. The preprocess procedure in ZKPyTorch.

an example of converting an ML model into a computational graph, where DAGs represent ML computations as a series of nodes connected by directed edges. Each node corresponds to a primitive operation and each edge represents the flow of data between these operations. The acyclic structure ensures the absence of circular dependencies, facilitating efficient computation and clear dependency tracking. This property is crucial for preserving the integrity of the model execution flow. In addition, the connections between nodes retain structural information. By grouping related nodes, we can recognize the original model architecture, enabling effective optimizations in subsequent modules.

Another key challenge lies in addressing the gap between standard ML computations and the requirements for ZKP circuits. Unlike traditional ML computations, which focus solely on obtaining results, ZKP circuits require additional auxiliary information to ensure cryptographic correctness. To address this, we insert additional nodes into our generated DAGs to add auxiliary information required for ZKP generation. Most operations requiring auxiliary information are non-linear and rely on range proofs or table lookups. A classic example is division, where we must prove that the remainder is smaller than the divisor. Another example is the softmax operation, where instead of using Taylor expansion to approximate the softmax operation through addition and multiplication, we employ a table lookup to retrieve the result directly from a precomputed table. Therefore, additional nodes are required in DAGs to perform table lookups.

In summary, our preprocessing module of ZKML compiler formalizes ML operations and standardizes computation processes. By using ONNX, we standardize ML operations across frameworks. Directed Acyclic Graphs (DAGs) represent computation processes, ensuring clear data flow and operation sequencing. To satisfy ZKP requirements, additional nodes are inserted to capture auxiliary information, like remainders in division operations. This approach simplifies the integration of ML models with cryptographic proofs, enhancing the modularity and scalability of our ZKML compiler.

Table 2. Accuracy of quantized convolutional neural networks on the CIFAR-10 dataset.

Model	Original Accuracy	Quantized Accuracy
VGG-16	94.13%	94.11%
ResNet-50	93.96%	93.94%
ResNet-101	93.83%	93.80%

3.3 ZKP-friendly ML Quantization

Machine learning quantization is a crucial technique for reducing the storage and computational complexity of deep learning models, making them more efficient for deployment on resource-constrained devices. In this context, quantization serves as a bridge between traditional ML computations, which use floating-point numbers for parameter storage and computation, and ZKP workflows, which operate within finite fields. To address this issue, most previous ZKML schemes [9, 13, 23, 25] directly employ fixed-point numbers to approximate floating-point values. However, due to the wide dynamic range of floating-point numbers, fixed-point representations require a significantly larger bit width to maintain accuracy, as discussed in Section 2.3. This disparity forces fixed-point representations to allocate additional bits to emulate floating-point numbers, thereby necessitating ZKML to operate over large finite fields, such as the scalar field of the BN254 curve, which significantly reduces efficiency in proof generation.

In this paper, we propose a ZKP-friendly ML quantization that uses integer-based representations, which are better suited for computations within finite fields. As part of ZKPyTorch's construction, this process involves transforming a given ML model into a ZKP-friendly form while preserving inference accuracy as much as possible. Adopting an integer-based representation necessitates the quantization of parameters originally represented as floating-point numbers. Unlike quantization techniques aimed at model compression, ZKP-friendly quantization must address stricter constraints imposed by ZKP backends. Specifically, it must ensure that all intermediate results remain integers, avoid computational overflow, and minimize reliance on non-linear

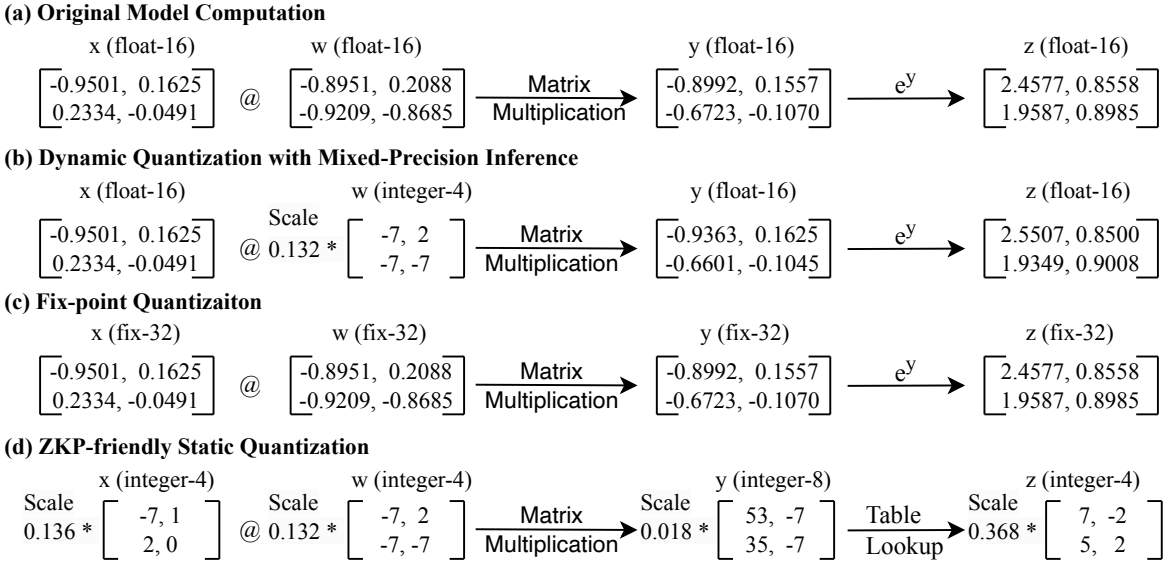

Figure 4. An example of ZKP-friendly quantization compared to other quantization methods.

operations, such as exponent in the softmax layer that requires costly emulation in ZKP circuits. Consequently, common quantization techniques such as dynamic quantization and mixed-precision inference, which are widely used in traditional transformer models, are not directly applicable. Instead, we design static quantization methods tailored to ZKP constraints, replacing floating-point operations with integer-based alternatives.

Figure 4 illustrates the difference between the three quantization methods. In practice, ZKPyTorch employs symmetric per-tensor static quantization for models evaluated in our experiments. During the calibration phase, we estimate the value range required to determine the quantization scale and verify that the selected bit width adequately represents intermediate values without incurring overflow, particularly in operations such as matrix multiplications. The quantization scale is shared across all elements in each layer. After calibration, we ensure that the quantized model meets the specified accuracy requirements.

For instance, we observed that convolutional neural network models achieve high-accuracy inference with our quantization. As shown in Table 2, it incurs only a slight accuracy loss compared to the original neural networks. However, transformer neural networks, such as Llama-3 model, have more complex non-linear operations and thus require further optimization. To enhance its performance, we introduced temporary bit-width adjustments and piecewise lookup tables to ensure precise summation during the exponent operation in the softmax layer and the square root operation in RMS normalization. The lookup table stores integers that are closest to the results of floating-point operations, ensuring that these integers can accurately simulate floating-point calculations. These optimizations enabled our quantized Llama-3 model to achieve a 99.32% cosine similarity with the original floating-point model, demonstrating our approach's effectiveness in preserving inference accuracy while ensuring ZKP compatibility.

3.4 Hierarchical ZKP circuit optimizer

This section presents our hierarchical ZKP circuit optimizer in ZKPyTorch. Our optimization approach follows a hierarchical architecture, rather than focusing solely on the optimization of the final ZKP circuit. This method ensures we do not miss the chance to optimize high-level semantics, which has a far greater impact on efficiency than the detailed optimization at the ZKP circuit level. In this way, we made optimizations at the ML model level, primitive operation level, and circuit level.

ML model level. We optimize ZKP circuits from the perspective of the overall ML inference process, with a primary focus on optimizing batch processing for the inference. Unlike the computation process in ML, proving the correctness of ML inference leverages the fact that the output is already known. For instance, traditional large language models need to generate tokens sequentially using transformer neural networks, where each token depends on the previous one, requiring step-by-step computation. However, in a ZKML compiler, this sequential dependency can be decoupled, as the purpose of a ZKP circuit is not to compute the output but to verify its correctness.

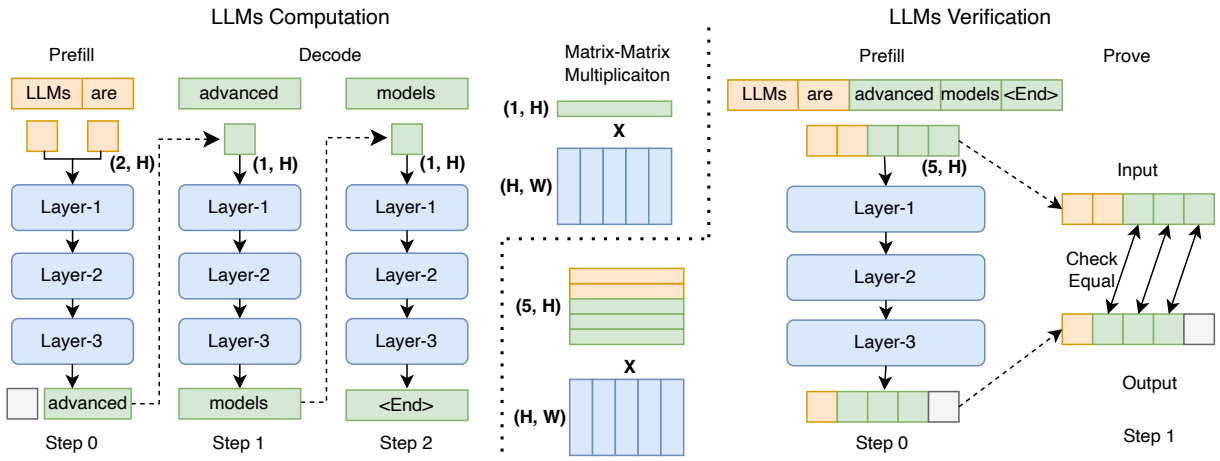

Figure 5. Batch verification of LLMs' computation.

As shown in Figure 5, traditional LLMs computation requires generating tokens sequentially. We employ LLMs verification by collecting the output tokens in batches, thereby ZKPs can be generated using a single proving circuit. In this way, the existing ZKP schemes for matrix multiplication can be fully leveraged. For example, in transformer neural networks, there are around L instances of 1-H-W matrix multiplication, where model weights form a matrix of size $H \times W$ and the activation constitutes another matrix of size $1 \times H$, with L representing the token length. Using sequential ZKP circuits for transformer matrix multiplications requires proving L separate operations, resulting in a total complexity of $O(LHW)$ gates. In contrast, our batch circuits enable matrix multiplication in transformer neural networks with a complexity of $O(WH + LH)$ gates, significantly reducing the overhead from multiple invocations of transformer layers. This approach can also be extended to convolutional neural networks for batch proving of image predictions.

Primitive operation level. ZKP circuits for primitive operations are the most crucial components of ZKML. Consequently, extensive researches [4, 13, 15, 16, 23] has focused on optimizing these operations. For example, ZKCNN [16] converts convolution operations into Fast Fourier Transform (FFT) operations, which are optimized to achieve linear-time complexity for proof generation by leveraging their inherent structure. ZKLLM [23] converts non-linear operations, such as softmax and GELU operations, used in large language models into table lookup operations, which significantly improves the proving efficiency for these non-linear operations.

These advancements significantly reduce the scale of ZKP circuits for neural networks, leading to more efficient proof generation. Unlike directly compiling the original computational process into addition and multiplication gates, these optimizations primarily take advantage of having the output available during proof generation to minimize gate requirements. Therefore, rather than introducing new optimization approaches, ZKPyTorch integrates existing techniques for primitive operations to enhance efficiency, ensuring compatibility with state-of-the-art methods while maintaining scalability for large-scale machine learning models.

Circuit level. Our circuit-level optimizations focus on parallelizing ZKP circuits, a target not addressed in previous ZKML compilers [9]. Since both traditional ML computations and ZKP generation naturally benefit from parallel processing, adapting ZKP circuits for multi-core hardware is crucial for efficiency, as it enables faster proof generation. To achieve this, we decompose the overall ZKP circuit into multiple parallel sub-circuits, allowing independent execution. By leveraging parallelism at the circuit level, we make proof generation more suitable for large-scale ML models.

The first optimization parallelizes batch execution of ML models, allowing them to process multiple inputs simultaneously. For instance, in convolutional neural networks, inference on hundreds of images can be verified in parallel, with each image's computation assigned to an independent ZKP sub-circuit. The second optimization focuses on parallelizing tensor operations, distributing element-wise computations, such as addition, multiplication, and activation functions, across multiple processing units to enhance efficiency. By efficiently utilizing hardware resources, this optimization reduces bottlenecks and enables seamless scaling to handle increasingly complex ML workloads.

During proving, certain data, such as the quantization scale of a tensor, mentioned in Section 3.3, must be shared across sub-circuits. To maintain their independence and avoid synchronization overhead, we employ a broadcasting approach, duplicating and distributing necessary data to each sub-circuit to eliminate dependencies. By structuring

computations in this parallelized manner, our strategy enables the potential of efficient proof generation on multi-core hardware, including CPUs and GPUs.

4 End-to-end User Cases

With ZKPyTorch, we can seamlessly integrate PyTorch, a leading machine learning (ML) framework, with Expander, a state-of-the-art ZKP engine. This integration enables AI developers to build cryptographically secure and verifiable neural networks while maintaining the flexibility and ease of use provided by PyTorch. By bridging the gap between traditional ML development and advanced cryptographic proof generation, ZKPyTorch significantly reduces the complexity of implementation while improving efficiency and scalability. In this section, we present end-to-end user cases about verifiable machine-learning-as-a-service and verifiable model valuation by employing ZKPyTorch.

Verifiable Machine-Learning-as-a-Service. As machine learning models become increasingly valuable, AI developers can build and deploy their own models on cloud platforms like Google Cloud [11], offering them as Machine Learning-as-a-Service (MLaaS). However, users often face challenges in verifying the correctness of model computations, while developers seek to protect their intellectual property by restricting access to the model's underlying details. ZKML addresses this challenge by enabling AI developers to provide verifiable computation results without revealing sensitive model information. However, traditional AI developers may lack expertise in constructing cryptographically secure ZKML solutions, limiting the widespread adoption of ZKML.

ZKPyTorch acts as a bridge. We use the original Llama-3 as input to construct a verifiable Machine-Learning-as-a-Service (MLaaS) system. As shown in Figure 6, AI developers can directly feed the Llama-3 model [12] into ZKPyTorch to construct a verifiable Machine Learning-as-a-Service (MLaaS) system. With ZKPyTorch integrating with the Expander engine, the framework automatically generates zero-knowledge proofs (ZKPs) that verify the correctness of inference results while safeguarding the confidentiality of the model. This streamlined process requires minimal cryptographic expertise, making the adoption of zero-knowledge machine learning (ZKML) more accessible and efficient. This approach enhances trust in cloud-based AI services, making them both secure and transparent. This is particularly valuable in high-security applications such as financial risk assessment, and medical diagnostics. By optimizing the computational overhead of ZKP generation, ZKPyTorch provides an efficient and scalable solution for real-world ZKML deployment.

Verifiable Model Valuation. Another key use case is verifiable model valuation. As AI models become increasingly valuable assets, ensuring their fair valuation while maintaining confidentiality is a critical challenge. Traditional model valuation relies on direct access to model parameters and

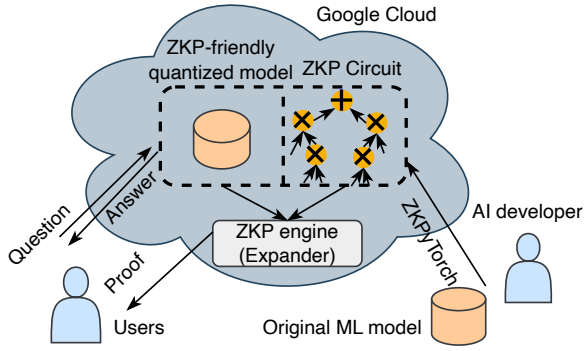

Figure 6. The use case of ZKPyTorch in variable Machine Learning-as-a-Service (MLaaS).

performance benchmarks, which can expose proprietary information and lead to potential misuse or replication of intellectual property. ZKML provides a novel solution by enabling verifiable model valuation through ZKPs, ensuring transparency in model assessment without revealing sensitive details. This approach allows stakeholders, such as AI developers and investors, to validate a model's worth based on cryptographic proofs rather than direct access to its parameters, safeguarding trade secrets while maintaining trust in the valuation process.

With ZKPyTorch, we can easily generate cryptographic proofs that verify key valuation metrics, such as model accuracy and robustness, without compromising the model's internal details. For example, CNNs are widely used in image classification and object detection, making them prime candidates for model valuation due to their widespread adoption and impact in real-world applications. We employ the VGG-16 model [22] as an example to achieve verifiable model valuation based on its classification accuracy on the CIFAR-10 dataset [6], ensuring a fair and secure evaluation. By leveraging ZKPyTorch, we translate PyTorch models into circuits compatible with Expander, enabling seamless integration of model valuation into ZKP workflows while maintaining computational efficiency and scalability. This methodology ensures that AI models can be assessed transparently without disclosing proprietary information.

5 Conclusion

As AI becomes increasingly integrated into critical domains, ensuring the verifiability of computations while preserving confidentiality is essential. In this paper, we propose ZKPyTorch, a hierarchically optimized compiler for zero-knowledge machine learning. ZKPyTorch bridges ML frameworks like PyTorch with ZKP engines such as Expander, enabling seamless model conversion and proof generation.

Through its automated preprocessing, ZKP-friendly quantization, and hierarchical circuit optimization, ZKPyTorch streamlines the development of ZKML, making it more accessible for AI developers while reducing computational overhead. Our work also demonstrates its effectiveness in converting and proving models like VGG-16 and Llama-3. Furthermore, our approach enhances the practicality of privacy-preserving AI deployments, making ZKML more viable across domains such as verifiable machine-learning-as-a-service and verifiable model valuation.

References

[1] Shuroug A Alowais, Sahar S Alghamdi, Nada Alsuhebany, Tariq Alqahtani, Abdulrahman I Alshaya, Sumaya N Almohareb, Atheer Aldairem, Mohammed Alrashed, Khalid Bin Saleh, Hisham A Badreldin, et al. 2023. Revolutionizing healthcare: the role of artificial intelligence in clinical practice. *BMC medical education* 23, 1 (2023), 689.

[2] Caffe2. 2025. Caffe2: a lightweight, modular, and scalable deep learning framework. https://github.com/facebookarchive/caffe2.

[3] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, et al. 2024. A survey on evaluation of large language models. *ACM transactions on intelligent systems and technology* 15, 3 (2024), 1–45.

[4] Bing-Jyue Chen, Suppakit Waiwitlikhit, Ion Stoica, and Daniel Kang. 2024. ZKML: An Optimizing System for ML Inference in Zero-Knowledge Proofs. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 560–574.

[5] Li Chen, Penghao Wu, Kashyap Chitta, Bernhard Jaeger, Andreas Geiger, and Hongyang Li. 2024. End-to-end autonomous driving: Challenges and frontiers. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2024).

[6] CIFAR-10. 2025. A collection of images that are commonly used to train machine learning and computer vision algorithms. https://www.cs.toronto.edu/~kriz/cifar.html.

[7] Expander. 2025. Expander: an open-source GKR prover designed for scaling large-scale parallel computing. https://github.com/PolyhedraZK/Expander.

[8] Boyuan Feng, Lianke Qin, Zhenfei Zhang, Yufei Ding, and Shumo Chu. 2021. ZEN: An Optimizing Compiler for Verifiable, Zero-Knowledge Neural Network Inferences. https://eprint.iacr.org/2021/087 Cryptology ePrint Archive, Paper 2021/087.

[9] Boyuan Feng, Zheng Wang, Yuke Wang, Shu Yang, and Yufei Ding. 2024. ZENO: A Type-based Optimization Framework for Zero Knowledge Neural Network Inference. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 450–464.

[10] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. 1989. The Knowledge Complexity of Interactive Proof Systems. *SIAM J. Comput.* 18, 1 (1989), 186–208.

[11] google. 2025. Google Cloud: Make smarter decisions with the leading data platform.. https://cloud.google.com.

[12] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. 2025. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2025).

[13] Meng Hao, Hanxiao Chen, Hongwei Li, Chenkai Weng, Yuan Zhang, Haomiao Yang, and Tianwei Zhang. 2024. Scalable Zero-knowledge Proofs for Non-linear Functions in Machine Learning. In *33rd USENIX Security Symposium (USENIX Security 24)*. 3819–3836.

[14] Justin D Harris and Bo Waggoner. 2019. Decentralized and collaborative AI on blockchain. In *2019 IEEE international conference on blockchain (Blockchain)*. IEEE, 368–375.

[15] Daniel Kang, Tatsunori Hashimoto, Ion Stoica, and Yi Sun. 2023. Scaling up Trustless DNN Inference with Zero-Knowledge Proofs. In *NeurIPS 2023 Workshop on Regulatable Machine Learning*. https://nips.cc/virtual/2023/80626

[16] Tianyi Liu, Xiang Xie, and Yupeng Zhang. 2021. zkCNN: Zero Knowledge Proofs for Convolutional Neural Network Predictions and Accuracy. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21)*. ACM, 2968–2985. https://doi.org/10.1145/3460120.3485379

[17] Tianyi Liu, Xiang Xie, and Yupeng Zhang. 2021. zkCNN: Zero Knowledge Proofs for Convolutional Neural Network Predictions and Accuracy. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2968–2985.

[18] Tao Lu, Haoyu Wang, Wenjie Qu, Zonghui Wang, Jinye He, Tianyang Tao, Wenzhi Chen, and Jiaheng Zhang. 2024. An efficient and extensible zero-knowledge proof framework for neural networks. *Cryptology ePrint Archive* (2024).

[19] ONNX. 2025. Open standard for machine learning interoperability. https://github.com/onnx/onnx.

[20] PyTorch. 2025. PyTorch: Tensors and Dynamic neural networks in Python with strong GPU acceleration. https://github.com/pytorch/pytorch.

[21] scikit learn. 2025. scikit-learn: machine learning in Python. https://github.com/scikit-learn/scikit-learn.

[22] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[23] Haochen Sun, Jason Li, and Hongyang Zhang. 2024. zkLLM: Zero Knowledge Proofs for Large Language Models. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) *(CCS '24)*. Association for Computing Machinery, New York, NY, USA, 4405–4419. https://doi.org/10.1145/3658644.3670334

[24] Tensorflow. 2025. Tensorflow: An Open Source Machine Learning Framework for Everyone. https://github.com/tensorflow/tensorflow.

[25] Chenkai Weng, Kang Yang, Xiang Xie, Jonathan Katz, and Xiao Wang. 2021. Mystique: Efficient conversions for {Zero-Knowledge} proofs with applications to machine learning. In *30th USENIX Security Symposium (USENIX Security 21)*. 501–518.

[26] Jiaheng Zhang, Zhiyong Fang, Yupeng Zhang, and Dawn Song. 2020. Zero Knowledge Proofs for Decision Tree Predictions and Accuracy. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20)*. Association for Computing Machinery, New York, NY, USA, 2039–2053. https://doi.org/10.1145/3372297.3417278