Sparsity-Aware Protocol for ZK-friendly ML Models: Shedding Lights on Practical ZKML

Alan Li alan@brevis.network
Qingkai Liang victor@brevis.network
Mo Dong mdong@brevis.network

Abstract

As deep learning is being widely adopted across various domains, ensuring the integrity of models has become increasingly crucial. Despite the recent advances in Zero-Knowledge Machine Learning (ZKML) techniques, proving the inference over large Machine Learning (ML) models is still prohibitive. To enable practical ZKML, model simplification techniques like pruning and quantization should be applied without hesitation. Contrary to conventional belief, recent development in ML space have demonstrated that these simplification techniques not only condense complex models into forms with sparse, low-bit weight matrices, but also **maintain exceptionally high model accuracies** that matches its unsimplified counterparts.

While such transformed models seem inherently ZK-friendly, directly applying existing ZK proof frameworks still lead to suboptimal inference proving performance. To make ZKML truly practical, a quantization-and-pruning-aware ZKML framework is needed. In this paper, we propose SpaGKR, a novel sparsity-aware ZKML framework that is proven to surpass capabilities of existing ZKML methods. SpaGKR is a general framework that is widely applicable to any computation structure where sparsity arises. It is designed to be modular – all existing GKR-based ZKML frameworks can be seamlessly integrated with it to get remarkable compounding performance enhancements. We tailor SpaGKR specifically to the most commonly-used neural network structure – the linear layer, and propose the SpaGKR-LS protocol that achieves asymptotically optimal prover time. Notably, when applying SpaGKR-LS to a special series of simplified model - ternary network, it achieves further efficiency gains by additionally leveraging the low-bit nature of model parameters.

1 Introduction

Over the past decade, deep learning has achieved remarkable success across various applications, thanks to continuous improvements in model performance [6, 10, 17, 20, 25, 28, 38–40, 42, 46, 48, 49]. This progress has been driven by an explosion in data volume, significant advancements in hardware computation speeds, and continuous evolution in model architectures. As a byproduct, there has been a trend of model size increasing over these years. The number of parameters of transformer-based language models has increased from 60-200 million [49] to 175-540 billion [6, 10], a leap of more than 3 order of magnitudes. As deep learning models are increasingly deployed in sensitive areas such as healthcare [12, 36], legal [5, 8] and blockchain applications, ensuring the integrity of model predictions becomes crucial. A straightforward approach to verify model integrity would involve making the models public and allowing for external verification. Nonetheless, this is often infeasible as commercial models frequently involve proprietary or sensitive data.

To address the integrity challenge without compromising confidentiality, Zero Knowledge Machine Learning (ZKML) employs zero-knowledge proofs within the machine learning context [2, 4, 13, 15, 23, 24, 26, 27, 29, 52, 56]. Recently, ZKML has advanced significantly, evolving from models like decision trees [27, 56] to complex architectures such as CNNs [29, 33] and Transformers [1, 37]. Despite these advancements, current ZKML systems still face efficiency challenges, particularly with larger models. For example, a recent study [37] has shown that proving an single-world inference result of an LLM with 1.5 billion parameters takes 90 hours. Later, [45] managed to prove inference on LLMs of sizes up to 13 billion with proving time of 1-15 minutes. While this is a huge leap, considering that state-of-the-art large language models may have hundreds of billions, or even trillions, of parameters, there is an urgent need for new techniques to accelerate these processes.

In machine learning field, the substantial size of these large models presents difficulties in deploying them in environments with limited computational resources. Not only is hosting these large models challenging, but inference is also slow and costly. However, such large models are often heavily over-parameterized, i.e., there exists a huge number of redundant neurons that do not improve the accuracy of model performance. Moreover, most existing models are trained with 32-bit floating points (FP32), which provides greater precision than needed. Model pruning and quantization techniques [7, 11, 18, 19, 22, 30, 31, 35, 41, 51, 53] have been developed to address these issues by transforming dense, high-precision parameters (e.g., FP32) into sparse, lower-bit representations (e.g., 8-bit integers, INT8). This lead to dramatic reduction in computational demands while still maintaining the original model performance. As an example, [18] achieved a 10-fold reduction in parameter count through effective pruning without compromising accuracy.

While mildly-quantized (e.g., INT8) models have already been explored in the context of ZKML to accelerate proving process (for example, [13, 33]), limited attentions have been put on sparse models where there exist a large portion of model parameters are zeros and quantizations are done towards their extreme. In this paper, we explore such a specialized model category known as ternary networks. These models have parameters that adopt values from $\{-1, 0, 1\}$ [30, 35]. Despite the apparent simplicity of these models, one might anticipate a significant decline in performance. Surprisingly, recent studies (e.g., [35]) show that with meticulous quantization and training protocols, ternary networks can achieve performance levels comparable to their non-quantized counterparts. Such quantization was applied to models of size 3.9 billion and has not seen its limitation yet, enabling efficient inference of large models.

Ternary networks (with successful realization as [35] etc.) offer a dramatic increase in computational efficiency without compromising model accuracy. This efficiency gain is attributed to several factors:

1. The presence of 0-valued weights introduces sparsity in model parameters, reducing the number of necessary computations;
2. Operations in element multiplication are simplified to additions and subtractions, eliminating the need for multiplicative computations;
3. Operations involving 32-bit floating-point (FP32) are transitioned to 8-bit integer (INT8) operations significantly reduces computational overhead.

These characteristics of ternary networks makes them extremely ZK-friendly, thus making them an ideal solution for combining the realms of ML and ZK and achiving practical ZKML.

To leverage sparsity that naturally arises in ternary networks, we design a general sparsity-aware framework called SpaGKR. This approach offers significant efficiency improvements over traditional, sparsity-ignorant ZKML methods, and is flexible in that any sparsity-aware sumcheck protocols (such as the one used in Lasso [44] and our new SpaSum protocol) could be plugged in as a protocol subroutine. We refine SpaGKR for linear layers – the most prevalent structure in neural networks – by leveraging their sparse and unique structural properties to enhance efficiency. The resulting SpaGKR-LS protocol achieves asymptotically optimal prover times, which is proportional to the number of non-zero entries, making it highly efficient. By applying SpaGKR-LS to ternary networks, the SpaGKR-LS protocol's prover efficiency could be greatly improved – a paper-pencil analysis suggests a massive 45x acceleration compared to direct application of sparsity-ignorant protocols.

We list prover time complexities for linear layers based on existing GKR-related protocols as well as our proposed SpaGKR and its variants in Tab. 1. Note that while we have ternary networks as our target application in mind, both SpaGKR and SpaGKR-LS are orthogonal to existing GKR-based ZKML methods and can be integrated seamlessly to improve the efficiency of existing protocols applied to various models.

Our Contributions. To the best of our knowledge, we are the first to explicitly study (in the context of ZKML) ternary networks, a special category of ZK-friendly ML models. We specifically consider

Scheme	Exploiting Sparsity	Linear Layer-Specific	Prover work
GKR [47]	No	No	$\mathcal{O}(m^2 \cdot n^2)$
Giraffe [50]	No	No	$\mathcal{O}(m \cdot n \cdot \log(m \cdot n))$
Libra [54]	No	No	$\mathcal{O}(m \cdot n)$
SpaGKR w/ Giraffe [50]	Yes	No	$\mathcal{O}(\log(m \cdot n) \cdot N^{\emptyset})$
SpaGKR w/ Lasso [44] (Thm. 5)	Yes	No	$\mathcal{O}((\frac{\log(m \cdot n)}{\log(N^{\emptyset})} + 1) \cdot N^{\emptyset})$
SpaGKR w/ SpaSum (Thm. 4)	Yes	No	$\mathcal{O}((\log(\frac{m \cdot n}{N^{\emptyset}}) + 1) \cdot N^{\emptyset})$
SpaGKR-LS (Thm. 7)	Yes	Yes	$\mathcal{O}(N^{\emptyset})$

Table 1: Prover time for proving linear layer based on existing GKR protocols and our sparsity-aware SpaGKR and its variants. m and n denotes the sizes of weight matrices and $N^{\emptyset}$ represents number of nonzeros, which should be a (small) fraction of $m \cdot n$ under sparsity setting.

leveraging sparsity in quantized and prunned models for more practical ZKML prover constructions. In particular, our contributions include:

1. We study ZKML on ternary networks, which lies on the sweet spot of ZK and ML literatures. Such networks naturally comes with sparsity property and shift most of field multiplications to field additions/subtractions, thus being much more efficient than its non-quantized counterparts while maintaining a competitive model performance. We show a huge potential efficiency gain when the characteristics of such models are leveraged. Our paper-pencil analysis reveal a massive 45x gain of efficiency in the proving protocols under mild assumptions due to such shift;

2. We propose SpaGKR, a sparsity-aware general GKR protocol to handle general layer structure where weights are potentially sparse. The protocol could be plugged in with various sparsity-aware sumcheck as subroutines. Such sumcheck protocols include existing ones like Spark/Surge [44] and our newly-proposed SpaSum (Thm. 3). The resulting protocol would potentially be much faster when a large portion of model parameters are zeros;

3. We propose SpaGKR-LS, an asymptotically optimal sparsity-aware GKR protocol speicifically for linear layer, the most widely-used model component in neural networks. The prover time is *linear* – only dependent on the number of nonzero entries of model parameters. The gain of efficiency comes from taking advantage of both sparsity and layer structures. Moreover, our protocol invokes only one sumcheck procedure, thus will be very efficient when being used.

1.1 Related Work

Our work lies in the intersection of Zero-Knowledge Machine Learning and model pruning/quantization. We highlight related works in these two fields and defer readers to [26, 31, 55] for a more comprehensive literature review.

Zero-Knowledge Machine Learning (ZKML) ZEN [13] is one of the earliest ZKML work that studies verifiable inference schemes for neural networks. vCNN [29] combines quadratic arithmetic program (QAP) with polynomial QAP to support proofs for convolutional neural networks (CNNs). [23] proved an ImageNet-scale model with help of lookup arguments for non-linearities and reuse of sub-circuits across layers. zkCNN [33] supports faster CNN proof with GKR variant that has linear-time complexity for proving convolutions layers. [4] introduced a modular framework for sumcheck-based proofs to verify sequential operations in machine learning and image processing. [56] and [27] proposed efficient zero knowledge proof schemes for decision tree predictions. While many of existing work have considered leveraging quantization techniques to build faster proof system, rarely did they explicitly discuss the potential of leveraging sparsity brought by pruning and quantization together.

Model Pruning and Quantization Model pruning and quantization are effective techniques to reduce model size and accelerate inference while maintaining the original model capabilities. It has been extensively studied on CNNs [7, 18, 22, 30, 32, 41] and Transformer-based language models [11, 35, 51, 53]. Existing work on effective pruning will leave the model with fewer parameters, the ratio of which ranges from 50% to as few as 2% [14, 18, 22], thus reducing the computational need by a large magnitude. Pushing the quantization to extreme results in binary [7, 41, 51] and ternary networks [30, 35], where model weights are elements in $\{-1, 1\}$ or $\{-1, 0, 1\}$. Although many binary networks experience a reduction in model performances, ternary networks close this gap by incorporating an additional zero value as an option, thereby introducing sparsity into model parameters. This adaptation effectively transforms sum of floating-point multiplications into simple additions and subtractions, resulting in a substantial gain in efficiency during model inference while preserving overall model performance.

Zero-Knowledge Proofs Considering Sparsity There have been works considering sparsity setting, though not under ZKML context. Perhaps the closest one with our work is Spartan [43] and its subsequent work Lasso [3, 44], where authors proposed polynomial commitment scheme for sparse polynomials and later used it for lookup arguments and zkVMs. Our linear layer setting resembles their lookup setting but differs in key assumptions (thus uncomparable), which will be elaborated in Sec. 3. Notably, these protocols will invoke multiple sumchecks during their proving process, while our protocol involves a single sumcheck.

2 Preliminaries

2.1 Model Pruning and Quntization

The rapid escalation in model sizes has precipitated considerable redundancy in model parameters, manifesting either as parameters that minimally influence overall performance or as those providing excessively high precision. Pruning, a technique designed to eliminate such redundancies, selectively removes parameters that do not significantly affect model accuracy. Typically, the targeted weights for pruning are those near zero or those that are redundant. As a result, pruned models often exhibit sparsity, characterized by a high proportion of zero-valued parameters that are excluded from computations, thus optimizing inference efficiency. Concurrently, quantization simplifies high-precision values (e.g., FP32 or FP16) into a limited set of discrete values or integers (e.g., INT8), substantially reducing computational demands during inference. While these strategies can be applied across various neural network components, our focus herein is their implementation within the linear layer, the most frequently utilized building block in contemporary neural networks.

2.1.1 Sparse Quantized Linear Layers

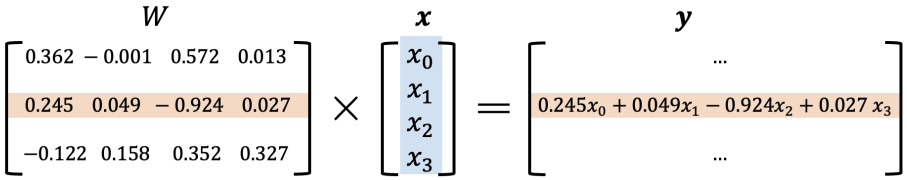
In the linear layer, the output y is computed by a matrix-vector product for linear transformation of input and a vector-vector addition for bias term. specifically, we have

$$\mathbf{y} = W\mathbf{x} + \mathbf{z}$$

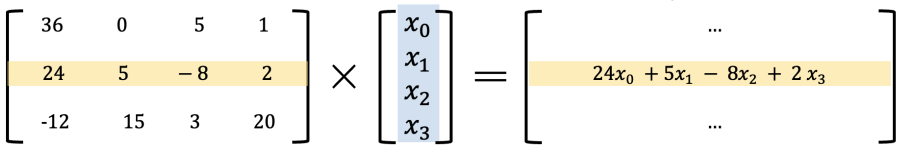
While this is the original definition of linear layer, we simplify it to just a linear transformation where W is concatenated with bias term $\mathbf{z}$ and $\mathbf{x}$ is concatenated with a 1. We slightly abuse the notation to still use W and $\mathbf{x}$ to denote the newly concatenated matrix/vector, resulting in

$$\mathbf{y} = W\mathbf{x}$$

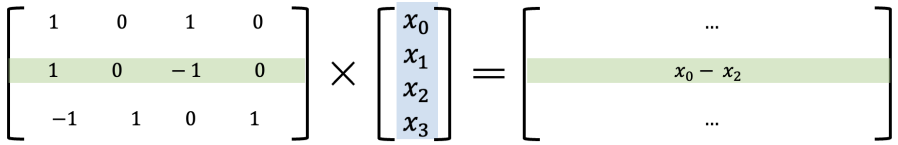
When pruning and quantization are applied, W become sparse parameters of integers with 8 or less bits and sum of high-precision products is transformed into sparse sum of low-precision products as shown in Fig. 1. Throughout this paper, we assume $W \in \mathbb{F}^{m \times n}$, $\mathbf{x} \in \mathbb{F}^n$ and $\mathbf{y} \in \mathbb{F}^m$.



(a) In the normal model with FP16 or FP32 parameters, computations of **y** are sums of high-precision products;



(b) With mild quantization (e.g., from FP16 to INT8), computations of **y** are transformed into sums of **low-precision** products;



(c) With extreme quantization (ternary networks), computations of **y** are transformed into **sparse** additions/subtractions, **excluding all multiplications**.

Figure 1: Comparison between normal linear layer, mildly-quantized linear layer and ternary networks.

2.1.2 Ternary Networks

In this paper, we examine a specialized category of models known as ternary networks [30, 35], characterized by extreme quantization where all weights are confined to $\{-1, 0, 1\}$. While this approach significantly enhances computational efficiency—making inference several times faster than full-precision variants—it typically incurs a notable reduction in performance, as model capacity and performance are generally positively correlated. However, the recently proposed model BitNet b1.58 [35] (referred to as BitNet in the rest of this manuscript) challenges this trend with two notable findings:

1. Contrary to expectations, this level of extreme quantization does not necessarily lead to a significant decrease in model capacity or accuracy. BitNet, through careful design and training, matches the performances of its full-precision (FP16) counterparts.

2. BitNet introduces a novel computation paradigm highly conducive to ZKML. It converts the sum of field multiplications into solely sparse field additions, eliminating all field multiplications. This transformation is illustrated in Fig. 1 and represents a paradigmatic shift that significantly enhances the efficiency of ZKML.

Given that BitNet variants not only recapture full model performance but also exhibit exceptional compatibility with ZKML, we advocate for their adoption as the go-to option of model choice within the ZKML framework. Our study aims to exploit the advantageous properties of sparsity and structure inherent in BitNet, anticipating that this exploration will illuminate the path to efficient ZKML and its practical application.

2.2 GKR and Sumcheck

Most existing proof systems in ZKML relies on GKR protocols [16] and their variants, of which sumcheck is the core computational process. Improvements on these protocols and computational process

will directly lead to improvements on prover efficiency. In this section, we present necessary preliminaries for sumcheck and GKR protocol.

2.2.1 Sumcheck

Sumcheck is a fundamental technique that has been widely used in various applications of zero knowledge proofs includking GKR. The problem is for the prover $\mathcal{P}$ to prove to the verifier $\mathcal{V}$ the summation of a polynomial $f(\cdot) : \{0,1\}^\ell \to \mathbb{F}$ on binary hypercube:

$$\sum_{b_1,\ldots,b_\ell \in \{0,1\}} f(b_1,\ldots,b_\ell)$$

While a direct computation for verifier will result in exponential time in ℓ, [34] proposed a sumcheck protocol that allows $\mathcal{V}$ to delegate most computation to $\mathcal{P}$, and $\mathcal{P}$ will do a series of computations and communications to convince $\mathcal{V}$ the correctness of summation.

sumcheck proceeds by iterations. During each iteration, the $\mathcal{P}$ will send to $\mathcal{V}$ a univariate polynomial that is a summation of a multivariate polynomial with certain subset of variables binding to certain values. Speifically, we define $\tilde{f}$ as MLE of f and

$$\begin{aligned} f_i(x) =& \sum_{b_{i+1},\ldots,b_\ell \in \{0,1\}} \tilde{f}(r_1,\ldots,r_{i-1},x,b_{i+1},\ldots,b_\ell) \\ =& \sum_{b_{i+1},\ldots,b_\ell \in \{0,1\}} g_i(x,b_{i+1},\ldots,b_\ell) \end{aligned}$$

where we define

$$g_i(b_i,\ldots,b_\ell) = \tilde{f}(r_1,\ldots,r_{i-1},b_i,\ldots,b_\ell)$$

as the bookkeeping table value for the i-th iteration. At the beginning of sumcheck process, $g_1(b_1,\ldots,b_\ell)$ is set to $f(b_1,\ldots,b_\ell)$ for $(b_1,\ldots,b_\ell) \in \{0,1\}^\ell$, and the computations of g_i's in the subsequent iterations follows the following lemma.

Lemma 1. *For bookkeeping table $\{g_i\}_{i=1}^\ell$ of vanilla sumcheck, g_{i+1} could be computed as*

$$\begin{aligned} g_{i+1}(b_{i+1},\ldots,b_\ell) =& \tilde{f}(r_1,\ldots,r_i,b_{i+1},\ldots,b_\ell) \\ =& (1-r_i)\cdot g_i(0,b_{i+1},\ldots,b_\ell) + r_i \cdot g_i(1,b_{i+1},\ldots,b_\ell) \end{aligned}$$

Thus for each set of $(b_{i+1},\ldots,b_\ell) \in \{0,1\}^{\ell-i}$, $g_{i+1}(b_{i+1},\ldots,b_\ell)$ can be computed in $\mathcal{O}(1)$ time. It follows that the whole g_{i+1} bookkeeping table could be computed in $\mathcal{O}(2^{\ell-i})$. Further, one could evaluate $f_i(0)$ and $f_i(1)$ in $\mathcal{O}(2\cdot 2^{\ell-i}) = \mathcal{O}(2^{\ell-i+1})$ time by

$$\begin{aligned} f_i(0) =& \sum_{b_{i+1},\ldots,b_\ell \in \{0,1\}} g_i(0,b_{i+1},\ldots,b_\ell) \\ f_i(1) =& \sum_{b_{i+1},\ldots,b_\ell \in \{0,1\}} g_i(1,b_{i+1},\ldots,b_\ell) \end{aligned}$$

One could then compute $f_i(r_i)$ as

$$f_i(r_i) = (1-r_i)\cdot f_i(0) + r_i \cdot f_i(1)$$

Computational cost by field operation counts Throughout this report, we analyze prover costs of various protocols by both time complexity and more refined field operation counts. We denote the arithmetic complexity of field multiplication by $\mathbb{M}$ and additions/subtractions by $\mathbb{A}$.

In the first iteration of vanilla sumcheck, the $g_1(b_1,\ldots,b_l)$ is simply set to $f(b_1,\ldots,b_l)$. It takes $(2^\ell - 2)\cdot\mathbb{A}$ to compute $f_1(0)$ and $f_1(1)$, and $2\cdot(\mathbb{A}+\mathbb{M})$ to compute $f_1(r_1)$. In the i-th iteration of

vanilla sumcheck where $i > 1$, it takes $2^{l-i+1} \cdot 2 \cdot (\mathbb{A} + \mathbb{M})$ to update the bookkeeping table from g_{i-1} to g_i, $(2^{l-i+1} - 2) \cdot \mathbb{A}$ to compute $f_i(0)$ and $f_i(1)$ and $2 \cdot (\mathbb{A} + \mathbb{M})$ to compute $f_i(r_i)$. Such recurrsion goes on until $i = l$. Summing everything up, the vanilla sumcheck will invoke the running time of

$$\begin{aligned}\Big(2^{\ell+1} - 2 \cdot (\ell+1)\Big) \cdot \mathbb{A} + (2^{\ell+1} - 4) \cdot (\mathbb{A} + \mathbb{M}) + 2 \cdot \ell \cdot (\mathbb{A} + \mathbb{M}) \\ = (4 \cdot 2^{\ell} - 6) \cdot \mathbb{A} + (2 \cdot 2^{\ell} + 2 \cdot \ell - 4) \cdot \mathbb{M}\end{aligned}$$

2.2.2 Product Sumcheck

A commonly-used variant of vanilla sumcheck is to prove a sum over the product of two functions, the process of which is called product sumcheck:

$$\sum_{b_1,\ldots,b_\ell \in \{0,1\}} f(b_1, \ldots, b_\ell) \cdot f'(b_1, \ldots, b_\ell)$$

Similar as vanilla sumcheck, we define

$$\begin{aligned} f_i(x) &= \sum_{b_{i+1},\ldots,b_\ell \in \{0,1\}} \tilde{f}(r_1, \ldots, r_{i-1}, x, b_{i+1}, \ldots, b_\ell) \cdot \tilde{f}'(r_1, \ldots, r_{i-1}, x, b_{i+1}, \ldots, b_\ell) \\ &= \sum_{b_{i+1},\ldots,b_\ell \in \{0,1\}} g_i(x, b_{i+1}, \ldots, b_\ell) \cdot g_i'(x, b_{i+1}, \ldots, b_\ell) \end{aligned}$$

where we define g_i and g_i' as the bookkeeping table for f and f' for the i-th iteration respectively. At the beginning of sumcheck procedure, $g_1(b_1, \ldots, b_\ell)$ is set to $f(b_1, \ldots, b_\ell)$ and $g_1'(b_1, \ldots, b_\ell)$ is set to $f'(b_1, \ldots, b_\ell)$ for $(b_1, \ldots, b_\ell) \in \{0,1\}^\ell$. Then we have the following lemma.

Lemma 2. *For bookkeeping table of product sumcheck we have*

$$\begin{aligned} g_{i+1}(b_{i+1}, \ldots, b_\ell) &= (1 - r_i) \cdot g_i(0, b_{i+1}, \ldots, b_\ell) + r_i \cdot g_i(1, b_{i+1}, \ldots, b_\ell) \\ g_{i+1}'(b_{i+1}, \ldots, b_\ell) &= (1 - r_i) \cdot g_i'(0, b_{i+1}, \ldots, b_\ell) + r_i \cdot g_i'(1, b_{i+1}, \ldots, b_\ell) \end{aligned}$$

For each set of $(b_{i+1}, \ldots, b_\ell) \in \{0,1\}^{\ell-i}$, $g_{i+1}(b_{i+1}, \ldots, b_\ell)$ can be computed in $\mathcal{O}(1)$ time, and the whole g_{i+1} bookkeeping table could be computed in $\mathcal{O}(2^{\ell-i})$. Further, one could evaluate $f_i(0)$, $f_i(1)$, $f_i'(0)$, $f_i'(1)$ in $\mathcal{O}(2 \cdot 2^{\ell-i}) = \mathcal{O}(2^{\ell-i+1})$ time by

$$\begin{aligned} f_i(0) &= \sum_{b_{i+1},\ldots,b_\ell \in \{0,1\}} g_i(0, b_{i+1}, \ldots, b_\ell) \cdot g_i'(0, b_{i+1}, \ldots, b_\ell) \\ f_i(1) &= \sum_{b_{i+1},\ldots,b_\ell \in \{0,1\}} g_i(1, b_{i+1}, \ldots, b_\ell) \cdot g_i'(1, b_{i+1}, \ldots, b_\ell) \end{aligned}$$

The computation of $f_i(r_i)$ then follows as

$$f_i(r_i) = (1 - r_i) \cdot f_i(0) + r_i \cdot f_i(1)$$

Computational cost by field operation counts Maintaining the bookkeeping tables requires twice the number of field operations as in vanilla sumcheck, which is $2 \cdot (2^{\ell+1} - 4) \cdot (\mathbb{A} + \mathbb{M})$; Computing $f_i(0)$ and $f_i(1)$ will require not only addition but also multiplication, which in total will be $\Big(2^{\ell+1} - 2 \cdot (\ell+1)\Big) \cdot (\mathbb{A} + \mathbb{M})$; Computing $f_i(r_i)$ will be the same as in vanilla sumcheck. Summing all things up, the product sumcheck will have the running time of

$$\begin{aligned}\Big(2^{\ell+1} - 2 \cdot (\ell+1)\Big) \cdot (\mathbb{A} + \mathbb{M}) + 2 \cdot (2^{\ell+1} - 4) \cdot (\mathbb{A} + \mathbb{M}) + 2 \cdot \ell \cdot (\mathbb{A} + \mathbb{M}) \\ = (6 \cdot 2^{l} - 10) \cdot (\mathbb{A} + \mathbb{M})\end{aligned}$$

2.2.3 GKR Protocol

With sumcheck as the core computational tool, [16] proposed an interactive proof protocol for circuits that are formed by layers. The proof is done recursively layer by layer. Following conventions of literature, let C be the circuit with d layers over finite field $\mathbb{F}$. In this note, we assume the input of $(i+1)$-th layer only comes from output of the i-th layer. The 0-th layer is the output layer while the d-th layer is the input layer. Let S_i be number of gates on the i-th layer and is a power of 2 and $s_i = \log(S_i)$. Function $V_i : \{0,1\}^{s_i} \to \mathbb{F}$ takes a binary string $b \in \{0,1\}^{s_i}$ and returns output of gate b on layer i. Originally the wiring predicates $add_{i+1}, mult_{i+1} : \{0,1\}^{s_i+2s_{i+1}} \to \{0,1\}$ take input of 3 binary string as inputs and outputs 1 iff the wiring exists. In this note, we use O^+_{i+1} and $O^\times_{i+1}$ to represent wiring predicates. Then V_i can be written as

$$\begin{aligned} V_i(c) &= \sum_{a,b\in\{0,1\}^{s_{i+1}}} O^+_{i+1}(c,a,b)\cdot(V_{i+1}(a)+V_{i+1}(b)) + O^\times_{i+1}(c,a,b)\cdot V_{i+1}(a)\cdot V_{i+1}(b) \\ &= \sum_{a,b\in\{0,1\}^{s_{i+1}}} f(c,a,b) \end{aligned}$$

General Gates While gates are usually assumed to be of fan-in= 2, in this note we focus on general gates where each gate can take more than 2 inputs. Such addition and multiplication gates have been mentioned in [21, 33]:

$$O^+_{i+1}(c,a) = \begin{cases} 1, & \text{if } V_{i+1}(a) \text{ is added to } V_i(c) \\ 0, & \text{otherwise} \end{cases}$$

$$O^\times_{i+1}(c,a,b) = \begin{cases} 1, & \text{if } V_{i+1}(a)\cdot V_{i+1}(b) \text{ is added to } V_i(c) \\ 0, & \text{otherwise} \end{cases}$$

Multilinear extension of layer i with generalized gates becomes

$$\tilde{V}_i(c) = \sum_{a\in\{0,1\}^{s_{i+1}}} \widetilde{O}^+_{i+1}(c,a)\cdot\tilde{V}_{i+1}(a) + \sum_{a,b\in\{0,1\}^{s_{i+1}}} \widetilde{O}^\times_{i+1}(c,a,b)\cdot\tilde{V}_{i+1}(a)\cdot\tilde{V}_{i+1}(b)$$

By definition of MLE we have

$$\begin{aligned} \tilde{f}(\mathrm{r}_1,\ldots,\mathrm{r}_l) &= \sum_{\mathrm{b}\in\{0,1\}^l}\prod_i[(1-\mathrm{r}_i)(1-\mathrm{b}_i)+\mathrm{r}_i\mathrm{b}_i]\cdot f(\mathrm{b}) \\ &= \sum_{\mathrm{b}\in\{0,1\}^l}\tilde{\beta}(\mathrm{r},\mathrm{b})\cdot f(\mathrm{b}) \end{aligned}$$

Plugging this into $\widetilde{O}^*$ we have

$$\widetilde{O}^+(\mathrm{r},\mathrm{u}) = \sum_{c\in\{0,1\}^{s_i},a\in\{0,1\}^{s_{i+1}}} \tilde{\beta}(\mathrm{r},c)\cdot\tilde{\beta}(\mathrm{u},a)\cdot O^+(c,a)$$

$$\widetilde{O}^\times(\mathrm{r},\mathrm{u},\mathrm{v}) = \sum_{c\in\{0,1\}^{s_i},a,b\in\{0,1\}^{s_{i+1}}} \tilde{\beta}(\mathrm{r},a)\cdot\tilde{\beta}(\mathrm{u},b)\cdot\tilde{\beta}(\mathrm{v},c)\cdot O^\times(c,a,b)$$

The Protocol The GKR protocol works as follows. $\mathcal{P}$ begins by sending the claimed output to $\mathcal{V}$, then V defines $\tilde{V}_0$ and computes $\tilde{V}_0(\mathbf{r})$ for a random $\mathbf{r} \in \mathbb{F}^{s_0}$. Then $\mathcal{V}$ and $\mathcal{P}$ will invoke a sumcheck protocol to prove the validity of $\tilde{V}_0(\mathrm{r})$, the problem of which will be reduced to $\mathcal{P}$ proving specific evaluations of $\tilde{V}_1(\cdot)$. In the i-th iteration, prover $\mathcal{P}$ sends the claimed output of $\tilde{V}_i(\cdot)$ to $\mathcal{V}$. $\mathcal{P}$ and $\mathcal{V}$ then invoke a sumcheck protocol to prove the validity of the evaluation. At the end of sumcheck, $\mathcal{V}$ needs an oracle access to $f_i(\mathrm{r},\mathrm{u},\mathrm{v})$, which is essentially $\tilde{V}_{i+1}(\mathrm{u})$ and $\tilde{V}_{i+1}(\mathrm{v})$. Note that (1) these two claims could be combined to one using standard techniques (e.g., [9, 16]); and (2) $\widetilde{O}^*(\cdot)$ could be computed locally by $\mathcal{V}$ since they only depend on the wiring pattern of the circuit. This procedure is done recursively until layer d, where $\mathcal{V}$ queries the oracle evaluations of $\tilde{V}_d$ and output accept or reject based on the result.

3 GKR Protocol for Linear Layers

We consider GKR protocol applied to linear layer of $\mathbf{y} = W\mathbf{x}$, where $W \in \mathbb{F}^{m\times n}$, $\mathbf{x} \in \mathbb{F}^n$ and $\mathbf{y} \in \mathbb{F}^m$.

Sparsity Assumption In the context of machine learning, it is natural to assume that for each *column* of the weight function has at least one nonzero entry. This is because if there exists some all-zero columns, it is simply ignoring some specific dimension of input and thus this dimension could be omitted during inference.

Relation to Lookups The definition of linear layer resembles an alternative definition of lookups [44]. In the context of lookups, W is essentially an indexing matrix and x is the lookup table. W is forced to have one nonzero value (more specifically, value of 1) on each *row* of the matrix. This makes it significantly different from our setting.

Layer Definition The only computation involved in linear layer is a sum of products between weights and data. By definition of GKR, assume V_i is the output of a linear layer, then for $c \in \{0,1\}^{s_i}$,

$$V_i(c) = \sum_{a,b\in\{0,1\}^{s_{i+1}}} O^{\times}_{i+1}(c,a,b) \cdot V_{i+1}(a) \cdot V_{i+1}(b)$$

3.1 Prover time for GKR on linear layer

By directly applying sumcheck protocol without considering any sparsity or structural information of the circuit, one would run the sumcheck for $2 \cdot s_{i+1}$ rounds. By using bookkeeping techniques in in [47], the prover time would be

$$\mathcal{O}(2^{2\cdot s_{i+1}}) = \mathcal{O}(S^2_{i+1}) = \mathcal{O}(m^2 \cdot n^2)$$

However, due to the special definition of linear layer, the wiring predicates are sparse, namely there are only $\mathcal{O}(m \cdot n)$ potential nonzero wiring predicates. Thus for $\mathrm{r} \in \mathbb{F}^{s_{i+1}}$, we have the following transformation:

$$V_i(\mathrm{r}) = \sum_{a,b\in\{0,1\}^{s_{i+1}}} \sum_{c\in\{0,1\}^{s_i}} \tilde{\beta}(\mathrm{r},c) \cdot O^{\times}_i(c,a,b) \cdot \tilde{V}_{i+1}(a) \cdot \tilde{V}_{i+1}(b)$$

It suffices for the prover to maintain only nonzero entries in the bookkeeping table [50], the size of which is $\mathcal{O}(m \cdot n)$ given sparsity of wiring predicates. Thus one could run the sumcheck for $2 \cdot s_{i+1} = \mathcal{O}(\log(m \cdot n))$ rounds, where in each round the summation will involve at most $\mathcal{O}(m \cdot n)$ non-zero terms. The total prover time of GKR [50] would be

$$\mathcal{O}(m \cdot n \cdot \log(m \cdot n))$$

In [54], authors propose a linear-prover-time protocol by dividing the sumcheck into two phases. Assume the sumcheck to be computed is

$$\sum_{x,y\in\{0,1\}^l} f_1(g,x,y) f_2(x) f_3(y)$$

The central idea is to rewrite the sum into two parts:

$$\sum_{x,y\in\{0,1\}^l} f_1(g,x,y) f_2(x) f_3(y) = \sum_{x\in\{0,1\}^l} f_2(x) \left(\sum_{y\in\{0,1\}^l} f_1(g,x,y) f_3(y) \right) = \sum_{x\in\{0,1\}^l} f_2(x) h_g(x)$$

where $h_g(x) = \sum_{y \in \{0,1\}^l} f_1(g,x,y) f_3(y)$. By leveraging sparsity of circuits with all fan-in= 2 gates the bookkeeping table for both f_2 and h_g could be initialized within $\mathcal{O}(2^l)$. It follows that the sumcheck over $x \in \{0,1\}^l$ could be done in $\mathcal{O}(2^l)$. Similarly summation over $y \in \{0,1\}^l$ could also be done in $\mathcal{O}(2^l)$. Thus the total running time of the sumcheck is $\mathcal{O}(2^l)$.

The whole procedure is built upon the foundation that the wiring is sparse in circuits with gates of fan-in= 2 In our case, though each gate could be of fan-in ≥ 2, the sparsity still applies. Specifically, we have

$$\begin{aligned}
&\sum_{a,b \in \{0,1\}^{s_{i+1}}} \widetilde{O}_i^{\times}(\mathrm{r}, a, b) \cdot \tilde{V}_{i+1}(a) \cdot \tilde{V}_{i+1}(b) \\
&\quad = \sum_{a \in \{0,1\}^{s_{i+1}}} \tilde{V}_{i+1}(a) \cdot \sum_{b \in \{0,1\}^{s_{i+1}}} \widetilde{O}_i^{\times}(\mathrm{r}, a, b) \cdot \tilde{V}_{i+1}(b) \\
&\quad = \sum_{a \in \{0,1\}^{s_{i+1}}} \tilde{V}_{i+1}(a) \cdot h_{\mathrm{r}}(a)
\end{aligned}$$

where

$$h_{\mathrm{r}}(a) = \sum_{b \in \{0,1\}^{s_{i+1}}} \widetilde{O}_{i+1}^{\times}(\mathrm{r}, a, b) \cdot \tilde{V}_{i+1}(b)$$

Then the corresponding sumcheck procedure could be divided into two phases.

Phase I In the first s_{i+1} rounds, $\mathcal{P}$ and $\mathcal{V}$ will run a sumcheck of the product of $\tilde{V}_{i+1}(\cdot)$ and $h_{\mathrm{r}}(\cdot)$, where both functions are on a as b is summed out. As long as the bookkeeping table for both $\tilde{V}_{i+1}$ and h_{r} could be initialized in $\mathcal{O}(2^{s_{i+1}})$ time, the first s_{i+1} rounds could be done within $\mathcal{O}(2^{s_{i+1}})$ time.

By expanding $h_{\mathrm{r}}(a)$ we have

$$\begin{aligned}
h_{\mathrm{r}}(a) &= \sum_{b \in \{0,1\}^{s_{i+1}}, c \in \{0,1\}^{s_i}} \tilde{\beta}(\mathrm{r}, c) \cdot \widetilde{O}_{i+1}^{\times}(c, a, b) \cdot \tilde{V}_{i+1}(b) \\
&= \sum_{(b,c) \in \mathcal{N}_a} \tilde{\beta}(\mathrm{r}, c) \cdot \widetilde{O}_{i+1}^{\times}(c, a, b) \cdot \tilde{V}_{i+1}(b)
\end{aligned}$$

where we define the set $\mathcal{N}_a \subseteq \{0,1\}^{s_i + s_{i+1}}$ such that $\widetilde{O}_{i+1}^{\times}(c,a,b) \neq 0$. Thus, to initialize bookkeeping table for $h_{\mathrm{r}}(a)$, it suffices to compute $h_r(a)$ where $a \in \{0,1\}^{s_{i+1}}$. For each a the computation could be done in $\mathcal{O}(|\mathcal{N}_a|)$ time, and given definition of wiring predicates we have

$$\sum_{a \in \{0,1\}^{s_{i+1}}} |\mathcal{N}_a| = \mathcal{O}(m \cdot n)$$

Thus the initialization of bookkeeping table for phase one could be done within $\mathcal{O}(m \cdot n)$ time. It follows that the sumcheck will take $\mathcal{O}(m \cdot n)$ time for Phase I.

Phase II After phase one, variables in a have been bounded to random numbers u. In the second phase, we need to invoke sumcheck over

$$\sum_{b \in \{0,1\}^{s_{i+1}}} \widetilde{O}_{i+1}^{\times}(\mathrm{r}, \mathrm{u}, b) \cdot \tilde{V}_{i+1}(\mathrm{u}) \cdot \tilde{V}_{i+1}(b) = \tilde{V}_{i+1}(\mathrm{u}) \sum_{b \in \{0,1\}^{s_{i+1}}} \tilde{V}_{i+1}(b) \cdot h_{\mathrm{r}}(b)$$

where

$$\begin{aligned}
h_{\mathrm{r}}(b) =& \widetilde{O}_{i+1}^{\times}(\mathrm{r}, \mathrm{u}, b) \cdot \tilde{V}_{i+1}(b) \\
&= \sum_{a \in \{0,1\}^{s_{i+1}}, c \in \{0,1\}^{s_i}} \tilde{\beta}(\mathrm{r}, c) \cdot \tilde{\beta}(\mathrm{u}, a) \cdot \widetilde{O}_{i+1}^{\times}(c, a, b) \cdot \tilde{V}_{i+1}(b) \\
&= \sum_{(a,c) \in \mathcal{N}_b} \tilde{\beta}(\mathrm{r}, c) \cdot \tilde{\beta}(\mathrm{u}, a) \cdot \widetilde{O}_{i+1}^{\times}(c, a, b) \cdot \tilde{V}_{i+1}(b)
\end{aligned}$$

where we define $\mathcal{N}_b \subseteq \{0,1\}^{s_i+s_{i+1}}$ such that $\widetilde{O}^{\times}_{i+1}(c,a,b) \neq 0$. By a similar argument as in Phase I, for each b the computation could be done in $\mathcal{O}(|\mathcal{N}_b|)$ time, and given definition of wiring predicates we have

$$\sum_{b\in\{0,1\}^{s_{i+1}}} |\mathcal{N}_b| = \mathcal{O}(m \cdot n)$$

Thus the initialization of bookkeeping table for Phase II could be done within $\mathcal{O}(m \cdot n)$ time. It follows that the sumcheck will take $\mathcal{O}(m \cdot n)$ time for Phase II.

Summing the running time for Phase I and Phase II together, the total running time for GKR [54] is

$$\mathcal{O}(m \cdot n)$$

4 SpaGKR: A General ZKML Framework Exploiting Sparsity

In the realm of machine learning (ML), the pruning and quantization of large models result in sparsity among the parameters. This characteristic of sparsity is extensively leveraged in the ML literature to enhance the efficiency of model inference. However, its application within the context of Zero-Knowledge Machine Learning (ZKML) has been underexplored. Particularly, no existing ZKML protocol explicitly capitalizes on model sparsity to improve proving efficiency to the best of our knowledge. In this section, we introduce a general ZKML framework based on GKR protocol exploiting sparsity. This framework optimizes proving efficiency by harnessing the sparsity inherent in the model. We specifically study its performance on sparse linear layers, the most widely-used structure in machine learning models. Our framework, referred to as SpaGKR, incorporates sumcheck as a core subroutine. It is designed to be versatile, allowing for any sparsity-aware sumcheck protocol to be integrated into the SpaGKR framework to expedite the proving process.

While the vanilla GKR protocol considers no sparsity within parameters, the key to make it sparsity-aware is to design gates that reflects the sparsity of parameters. Concretely, we modify the definition of wiring predicates as:

$$O_i^{+}(c,a) = \begin{cases} 1, & \text{if } V_{i+1}(a) \neq 0, \text{and is added to } V_i(c) \\ 0, & \text{otherwise} \end{cases}$$

$$O_i^{\times}(c,a,b) = \begin{cases} 1, & \text{if } V_{i+1}(a) \cdot V_{i+1}(b) \neq 0 \text{ and is added to } V_i(c) \\ 0, & \text{otherwise} \end{cases}$$

That is, we only allow for wirings to those gates with nonzero values. Note that such sparse-aware gates are extremely useful when the certain parts of gates are sparse. For example, most large neural network models' weights could be heavily pruned — the pruning rate could be more than 90%, i.e., 90%+ of weights is forced to be 0, and the sparsity will present in weight matrices. The resulting inference time would merely be a fraction to what it would be in original dense case, and shortly we will see that within our SpaGKR framework the prover efficiency will be greatly boosted by explicitly exploiting sparsity and linear structure. Note that SpaGKR is orthogonal to existing acceleration techniques of ZKML, and one could simply plug-in our setting in cases where sparsity presents and would result in better efficiency.

Given such sparsity-aware gates in SpaGKR, the next step is to plug in efficient sumcheck to prove the following:

$$\tilde{V}_i(\mathrm{r}) = \sum_{a,b\in\{0,1\}^{s_{i+1}}} \sum_{c\in\{0,1\}^{s_i}} \tilde{\beta}(\mathrm{r},c) \cdot \widetilde{O}_i^{\times}(c,a,b) \cdot \tilde{V}_{i+1}(a) \cdot \tilde{V}_{i+1}(b),$$

where r is the random challenge given by the verifier. While SpaGKR is flexible in that any sumcheck protocol could be directly plugged in, the question is how we could leverage sparsity in the sumcheck

to make the proving procedure more efficient. For example, given inspirations from sumcheck in Giraffe [50], in each round of sumcheck there are at most $\mathcal{O}(N_{i+1}^{\emptyset})$ non-zero entries, where $N_{i+1}^{\emptyset} = nnz(V_{i+1})$. As there are in total $2s_{i+1} = \mathcal{O}(\log(m \cdot n))$ rounds, the prover time would be

$$\mathcal{O}(N_{i+1}^{\emptyset} \cdot \log(m \cdot n))$$

Note that this could be a huge improvement compared to the naive application of Giraffe on linear layer (which has the prover time of $\mathcal{O}(m \cdot n \cdot \log(m \cdot n))$). If the model is sparse, it means $\frac{N^{\emptyset}}{m \cdot n}$ is small, thus it will be asymptotically more efficient than direct application of original protocol. If the model sparsity further satisfies $N^{\emptyset} = o(\frac{m \cdot n}{\log(m \cdot n)})$, using SpaGKR will be asymptotically more efficient than direct application of linear-prover-time protocols like Libra [54].

In the following, we will plug in two more efficient and sparsity-aware sumcheck protocols into SpaGKR and analyze their performance.

4.1 SpaGKR with SpaSum

We consider a sumcheck protocol that explicitly takes sparsity into consideration. Assume f is sparse on $\{0,1\}^{\ell}$: it evaluates to nonzero only on $nz(f) \subseteq \{0,1\}^{\ell}$. Then we propose a sparsity-aware sumcheck protocol called *SpaSum*, which is characterized by the following theorem:

Theorem 3. *The SpaSum algorithm where f's sparsity is characterized by $nz(f)$ will run in time $\mathcal{O}((c+1) \cdot |nz(f)|)$, where $c = \ell - \log(|nz(f)|)$*

We leave the concrete construction and proof to Appendix A.

Next, we show how we could efficiently utilize SpaSum to get efficient prover in SpaGKR. We divide the sumcheck procedure into two phases as follows.

Phase I In Phase I, there are $s_{i+1} = m \cdot n$ rounds and $\mathcal{P}$ and $\mathcal{V}$ will invoke a sumcheck of the product of $\tilde{V}_{i+1}(\cdot)$ and $h_{\mathrm{r}}(\cdot)$. The bookkeeping table for $\tilde{V}_{i+1}$ could be initialized in $\mathcal{O}(N_{i+1}^{\emptyset})$ time since there are $\mathcal{O}(N_{i+1}^{\emptyset})$ nonzero entries in $\tilde{V}_{i+1}$. For $h_{\mathrm{r}}(a)$ we have

$$h_{\mathrm{r}}(a) = \sum_{(b,c) \in \mathcal{N}_a} \tilde{\beta}(\mathrm{r}, c) \cdot \tilde{O}_{i+1}^{\times}(c, a, b) \cdot \tilde{V}_{i+1}(b)$$

To initialize bookkeeping table for $h_{\mathrm{r}}(a)$, it suffices to compute $h_r(a)$ where $a \in \{0,1\}^{s_{i+1}}$. For each a the computation could be done in $\mathcal{O}(|\mathcal{N}_a|)$ time, and given definition of wiring predicates we have

$$\sum_{a \in \{0,1\}^{s_{i+1}}} |\mathcal{N}_a| = \mathcal{O}(N_{i+1}^{\emptyset})$$

Thus the initialization of bookkeeping table for Phase I could be done within $\mathcal{O}(N_{i+1}^{\emptyset})$ time. Now $\mathcal{P}$ and $\mathcal{V}$ invokes a sumcheck of the product of $\tilde{V}_{i+1}(\cdot)$ and $h_{\mathrm{r}}(\cdot)$ where $\tilde{V}_{i+1}$ is sparse. With Thm. 3, the prover time will be

$$\mathcal{O}((p_{i+1}+1) \cdot N_{i+1}^{\emptyset})$$

where $p_{i+1} = \log(m \cdot n) - \log(N_{i+1}^{\emptyset}) = \log(\frac{m \cdot n}{N_{i+1}^{\emptyset}})$.

Phase II As for Phase II, a similar argument could be made and the resulting prover time will be of the same order of Phase I.

Putting everything together, we have the following theorem:

Theorem 4. *By exploiting the sparsity of linear layer problem and applying SpaSum Thm. 3, the SpaGKR will have the prover time of order*

$$\mathcal{O}((p_{i+1}+1) \cdot N_{i+1}^{\emptyset})$$

4.2 SpaGKR with Sumcheck in Lasso [44]

As we have mentioned, SpaGKR is a general framework where any sparsity-aware sumcheck protocols can be plugged in as a subroutine. While we could use SpaSum for implementation, one could also employ existing sparsity-aware sumcheck protocols, e.g., sparse-dense sumcheck used in Lasso as subroutines as it might be faster under certain cases. Specifically, we have the following theorem:

Theorem 5. *In Lasso [44] the authors also consider the case of proving sparse vector inner products. They propose an algorithm that runs at order of*

$$\mathcal{O}(c' \cdot |nz(f)|)$$

where $c' = \frac{\ell}{\log(|nz(f)|)}$

Comparisons. As mentioned in Thm. 5, the prover time for the sparse-dense sumcheck in Lasso runs in $\mathcal{O}(c' \cdot |nz(f)|)$ where $c' = \frac{\ell}{\log(|nz(f)|)} = \frac{\log(2^\ell)}{\log(|nz(f)|)}$ while SpaSum runs in $\mathcal{O}(c \cdot |nz(f)|)$ where $c = \ell - \log(|nz(f)|) = \log(\frac{2^\ell}{|nz(f)|})$ (Thm. 3). If $|nz(f)|$ is a constant, then both c and c' are of order linear in l; if $|nz(f)|$ is of some polynomial order of 2^ℓ then $\log(|nz(f)|)$ is bounded by a constant times ℓ, thus c will scale faster than c'; if $|nz(f)|$ is small, say, of order polylog of 2^l, then c' will scale faster than c. In reality, one could choose whichever that is fastest under certain cases for implementation.

By similarly dividing the sumcheck into two phases and plugging in sparse-dense sumcheck [44], we have the following corollary:

Corollary 6. *By exploiting the sparsity of linear layer problem and applying Thm. 5, the SpaGKR will have the prover time of order*

$$\mathcal{O}((p'_{i+1} + 1) \cdot N^{\emptyset}_{i+1})$$

where $p'_{i+1} = \frac{\log(m \cdot n)}{\log(N^{\emptyset}_{i+1})}$.

Note that till now SpaGKR still remains general and assumes no special structure of model. That is, even if the inference is not done via a linear layer, it can still be applied with minor modifications. However, such generalization ability comes at the cost of ignoring the special structure of linear layer.

5 SpaGKR-LS: GKR for Linear Layers Exploiting Structure

While we have the general circuit setting, the fact that linear layer circuit is highly structured is ignored. We redefine V_i to take circuit structure into consideration. Specifically, we define

$$V^{\mathrm{y}}_i(\cdot) : \{0,1\}^{\log(m)} \to \mathbb{F};$$
$$V^{\mathrm{x}}_{i+1}(\cdot) : \{0,1\}^{\log(n)} \to \mathbb{F};$$
$$V^{\mathrm{W}}_{i+1}(\cdot,\cdot) : \{0,1\}^{\log(m \cdot n)} \to \mathbb{F};$$

where function V^*_i encodes values of $*$. For $c \in \{0,1\}^{\log(m)}$, we have

$$V^{\mathrm{y}}_i(c) = \sum_{a \in \{0,1\}^{\log(n)}} V^{W}_{i+1}(c,a) \cdot V^{\mathrm{x}}_{i+1}(a)$$

It suffices for $\mathcal{P}$ to prove to $\mathcal{V}$ the value

$$\tilde{V}^{\mathrm{y}}_i(\mathrm{r}) = \sum_{a \in \{0,1\}^{\log(n)}} f_i(\mathrm{r},a) = \sum_{a \in \{0,1\}^{\log(n)}} \tilde{V}^{\mathrm{W}}_{i+1}(\mathrm{r},a) \cdot \tilde{V}^{\mathrm{x}}_{i+1}(a)$$

5.1 SpaGKR-LS Protocol

Expanding with MLE we could reduce the original problem to be a product sumcheck as

$$\begin{aligned}\tilde{V}_i^{\mathrm{y}}(\mathrm{r}) &= \sum_{a\in\{0,1\}^{\log(n)}} \sum_{c\in\{0,1\}^{\log(m)}} \tilde{V}_{i+1}^{\mathrm{x}}(a)\cdot\tilde{\beta}(\mathrm{r},c)\cdot\tilde{V}_{i+1}^{W}(c,a)\\ &= \sum_{a\in\{0,1\}^{\log(n)}} \tilde{V}_{i+1}^{\mathrm{x}}(a)\cdot h_r(a)\end{aligned}$$

where

$$\begin{aligned}h_{\mathrm{r}}(a) &= \sum_{c\in\{0,1\}^{\log(m)}} \tilde{\beta}(\mathrm{r},c)\cdot\tilde{V}_{i+1}^{W}(c,a)\\ &= \sum_{c\in\mathcal{N}_a} \tilde{\beta}(\mathrm{r},c)\cdot\tilde{V}_{i+1}^{W}(c,a)\end{aligned}$$

where we define $\mathcal{N}_a \subseteq \{0,1\}^{\log(m)}$ as the set of $c \in \{0,1\}^{\log(m)}$ such that there exists some $(c,a) \in \{0,1\}^{\log(m)+\log(n)}$ such that $\tilde{V}_{i+1}^{W}(c,a) \neq 0$. Thus the bookkeeping table for $\tilde{V}_{i+1}^{W}(c,a)$ could be initialized in $\sum_{a\in\{0,1\}^n} |\mathcal{N}_a| = \mathcal{O}(N_{i+1}^{\emptyset})$ time. As for $\tilde{\beta}(\mathrm{r},c)$, we discuss several cases with settings of $N_{i+1}^{\emptyset}$.

Case I: $N_{i+1}^{\emptyset} \geq m$**.** This is the most common case in the machine learning context where the number of nonzeros in a weight matrix is at the order of at least linear in number of rows in the weight matrix. In such case, $\tilde{\beta}(\mathrm{r},c)$ could be directly computed for all $c \in \{0,1\}^{\log(m)}$, which will take $\mathcal{O}(m)$ time. Thus the total initialization time for bookkeeping tables for $h_{\mathrm{r}}(a)$ for $a \in \{0,1\}^{\log(n)}$ is $\mathcal{O}(N_{i+1}^{\emptyset})$.

After initialization of bookkeeping tables, the sumcheck could be accomplished within $\mathcal{O}(n)$ time. Given $N_{i+1}^{\emptyset} = \Omega(n)$, the total running time for the prover becomes

$$\mathcal{O}(N_{i+1}^{\emptyset})$$

Case II: $N_{i+1}^{\emptyset} < m$ **.** If $N_{i+1}^{\emptyset} < m$, there exist $q \in \mathbb{F}$ such that $q > 1$ and $N_{i+1}^{\emptyset} = m^{\frac{1}{q}}$. This happens in some certain cases where m is large and a portion of rows in weight matrices are all zeros. While this is pretty uncommon in ML context, the setting is interesting in that, while naive initialization of bookkeeping table for $\tilde{\beta}(\mathrm{r},c)$ will take $\mathcal{O}(N_{i+1}^{\emptyset}\cdot\log(m))$ time, it could be further accelerated with specialized algorithm. The new procedure resembles Pippenger's algorithm for multiexponentiation and has been discussed in previous works [44] as well.

For all $\log(m)$ variables, we decompose them into q chunks where each chunk is of size $\log(N_{i+1}^{\emptyset})$. For each chunk, evaluating a subpart of $\tilde{\beta}(\mathrm{r},c)$ will require $\mathcal{O}(2^{\log(N_{i+1}^{\emptyset})}) = \mathcal{O}(N_{i+1}^{\emptyset})$ time to be fully computed. All these chunks could be initialized within $\mathcal{O}(q\cdot N_{i+1}^{\emptyset})$ time and then be stored into memory.

After computation of each subparts, for each c such that there exists $(c,a) \in \{0,1\}^{\log(m)+\log(n)}$ and $\tilde{V}_{i+1}^{W}(c,a) \neq 0$, one could simply compute $\tilde{\beta}(\mathrm{r},c)$ by multiplying each chunk in the memory together. Since there are q chunks and there are at most $N_{i+1}^{\emptyset}$ number of $\tilde{\beta}(\mathrm{r},c)$ to be computed, the total running time would be $\mathcal{O}(q\cdot N_{i+1}^{\emptyset})$.

Summing these two stages up, the initialization time for the bookkeeping table of $\tilde{\beta}(\mathrm{r},c)$ is $\mathcal{O}(q\cdot N_{i+1}^{\emptyset})$. The sumcheck afterwards could be accomplished within $\mathcal{O}(N_{i+1}^{\emptyset})$ time. Hence, the total running time for the prover becomes

$$\mathcal{O}(q\cdot N_{i+1}^{\emptyset})$$

In both cases, the prover time is *linear* in the size of input, thus is asymptotically optimal. For a clearer comparison, we list prover times under general GKR protocols as well as SpaGKR and SpaGKR-LS in Tab. 1.

Theorem 7. *By exploiting the sparsity and structure of linear layer problem, the SpaGKR-LS with aforementioned two phases invokes one sumcheck and will have the prover time of order linear in the sparsity, namely* $\mathcal{O}(N_{i+1}^{\emptyset})$.

Computational cost by operation counts We further analyze prover cost of the algorithm in Thm. 7 under mild assumption of $N^{\emptyset} > m$. During the initialization of bookkeeping table, $\tilde{\beta}(\mathrm{r}, c)$ could be computed for all $c \in \{0,1\}^{\log(m)}$ with by iterating over each dimension of c. In the first iteration, the computation involves $1 \cdot \mathbb{A}$ computation for computing $1 - \mathrm{r}_1$; In the i-th iteration, the computation involves $1 \cdot \mathbb{A}$ for computing $1 - \mathrm{r}_i$ and $2^i \cdot \mathbb{M}$ for computing 'leaves of the current tree'. Thus the total computation is $\log(m) \cdot \mathbb{A} + (2 \cdot m - 4) \cdot \mathbb{M}$ for computing $\tilde{\beta}$. To finish initialization of bookkeeping table for h_{r}, one should iterate through all $a \in \{0,1\}^{\log(n)}$ to compute $h_{\mathrm{r}}(a)$. For each a the computations involves $|\mathcal{N}_a| \cdot \mathbb{M} + (|\mathcal{N}_a| - 1)\mathbb{A}$ computation. Summing all things up, the initialization of bookkeeping table runs within

$$\begin{aligned}\log(m) \cdot \mathbb{A} + (2 \cdot m - 4) \cdot \mathbb{M} + &\sum_{a \in \{0,1\}^{\log(n)}} (|\mathcal{N}_a| \cdot \mathbb{M} + (|\mathcal{N}_a| - 1)\mathbb{A}) \\ =&(N^{\emptyset} - n + \log(m)) \cdot \mathbb{A} + (N^{\emptyset} + 2 \cdot m - 4) \cdot \mathbb{M}\end{aligned}$$

After initialization of bookkeeping tables, $\mathcal{V}$ and $\mathcal{P}$ will invoke a product sumcheck of dimension $\log(n)$. By Sec. 2.2.2, the running time will be

$$(6 \cdot n - 10) \cdot (\mathbb{A} + \mathbb{M})$$

Summing everything up, the total running time will be

$$(N^{\emptyset} + 5 \cdot n + \log(m) - 10) \cdot \mathbb{A} + (N^{\emptyset} + 6 \cdot n + 2 \cdot m - 14) \cdot \mathbb{M}$$

Benefits brought by sparsity-awareness We count number of operations from SpaGKR-LS's sparsity-ignorant counterpart to showcase how much benefit does the sparsity-awareness bring in terms of efficiency. If SpaGKR-LS is unaware of sparsity, the total computation for $\tilde{\beta}$ is still $\log(m) \cdot \mathbb{A} + (2 \cdot m - 4) \cdot \mathbb{M}$. When iterating through all $a \in \{0,1\}^{\log(n)}$, for each a the computation involves $m \cdot \mathbb{M} + (m - 1) \cdot \mathbb{A}$. Thus the total time for bookkeeping table initialization runs within

$$\begin{aligned}\log(m) \cdot \mathbb{A} + (2 \cdot m - 4) \cdot \mathbb{M} + &\sum_{a \in \{0,1\}^{\log(n)}} (m \cdot \mathbb{M} + (m - 1) \cdot \mathbb{A}) \\ =&(n \cdot (m - 1) + \log(m)) \cdot \mathbb{A} + (n \cdot m + 2 \cdot m - 4) \cdot \mathbb{M}\end{aligned}$$

Together with the product sumcheck of dimension $\log(n)$, the total running time will be

$$(n \cdot m + 5 \cdot n + \log(m) - 10) \cdot \mathbb{A} + (n \cdot m + 6 \cdot n + 2 \cdot m - 14) \cdot \mathbb{M}$$

To make the comparison more sensible, we plug a toy problem setting into the analysis. Assume that $m = 512$, $n = 1,024$ and sparsity ratio is 90%. We further assume that computing a field addition will take 1 time unit while a field multiplication will takes 10 time units. If we use sparsity-ignorant GKR, even if we exploit the structure property of the problem, the prover time will be around 5.84 million time units. If we use sparsity-aware SpaGKR-LS, the prover time will be around 653 thousand time units, a 9x acceleration.

5.2 Application to Ternary Networks

Applying SpaGKR-LS to ternary networks, we carefully study how much efficiency gain we will have leveraging both sparsity and extreme quantization. It suffices for $\mathcal{P}$ to prove to $\mathcal{V}$

$$\tilde{V}_i^{\mathrm{y}}(\mathrm{r}) = \sum_{a \in \{0,1\}^{\log(n)}} \tilde{V}_{i+1}^{\mathrm{x}}(a) \cdot h_r(a)$$

where

$$h_{\mathrm{r}}(a) = \sum_{c \in \mathcal{N}_a} \tilde{\beta}(\mathrm{r}, c) \cdot \tilde{V}^W_{i+1}(c, a)$$

While the asymptotic prover time is still $\mathcal{O}(N^{\emptyset}_{i+1})$, the concrete time will be less than that with general weights because $\tilde{V}^W_{i+1}(c, a) \in \{-1, 0, 1\}$ and will transform the sum of products to be additions and subtractions. Such savings will happen when we initialize the bookkeeping table for h_{r}, while all other analysis remain the same. Specifically, after computing $\tilde{\beta}$, when one iterates through all $a \in \{0,1\}^{\log(n)}$ to compute $h_{\mathrm{r}}(a)$, for each a the computation involves $(|\mathcal{N}_a| - 1)\mathbb{A}$ in contrast to the original $|\mathcal{N}_a| \cdot \mathbb{M} + (|\mathcal{N}_a| - 1)\mathbb{A}$ because all multiplications are eliminated. Thus the total savings would be

$$\sum_{a \in \{0,1\}^{\log(n)}} |\mathcal{N}_a| \cdot \mathbb{M} = N^{\emptyset} \cdot \mathbb{M}$$

Hence the resulting running time will be

$$(N^{\emptyset} + 5 \cdot n + \log(m) - 10) \cdot \mathbb{A} + (6 \cdot n + 2 \cdot m - 14) \cdot \mathbb{M}$$

If we again plug the analysis in a toy example, the running time of SpaGKR-LS on ternary network will be reduced to 129 thousands. Compared with non-ternary network where weights takes general values in the field, the acceleration will be already 5x. And compared with sparsity-ignorant protocols, even they are also exploiting structure, we still see a massive acceleration of 45x.

References

[1] Honey I SNARKed the GPT. URL https://hackmd.io/mGwARMgvSeq2nGvQWLL2Ww?ref=blog.spectral.finance.

[2] R. E. Ali, J. So, and A. S. Avestimehr. On polynomial approximations for privacy-preserving and verifiable relu networks. *arXiv preprint arXiv:2011.05530*, 2020.

[3] A. Arun, S. Setty, and J. Thaler. Jolt: SNARKs for Virtual Machines via Lookups. 2023.

[4] D. Balbás, D. Fiore, M. I. G. Vasco, D. Robissout, and C. Soriente. Modular sumcheck proofs with applications to machine learning and image processing. (2023/1342). Publication info: Published elsewhere. Minor revision. ACM CCS 2023.

[5] N. Bansal, A. Sharma, and R. Singh. A review on the application of deep learning in legal domain. In *Artificial Intelligence Applications and Innovations: 15th IFIP WG 12.5 International Conference, AIAI 2019, Hersonissos, Crete, Greece, May 24–26, 2019, Proceedings 15*, pages 374–381. Springer, 2019.

[6] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners. (arXiv:2005.14165).

[7] A. Bulat and G. Tzimiropoulos. XNOR-net++: Improved binary neural networks. (arXiv:1909.13863).

[8] I. Chalkidis and D. Kampas. Deep learning in law: early adaptation and legal word embeddings trained on large corpora. *Artificial Intelligence and Law*, 27(2):171–198, 2019.

[9] A. Chiesa, M. A. Forbes, and N. Spooner. A zero knowledge sumcheck and its applications. 2017. Publication info: Preprint. MINOR revision.

[10] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, P. Schuh, K. Shi, S. Tsvyashchenko, J. Maynez, A. Rao, P. Barnes, Y. Tay, N. Shazeer, V. Prabhakaran, E. Reif, N. Du, B. Hutchinson, R. Pope, J. Bradbury, J. Austin, M. Isard, G. Gur-Ari, P. Yin, T. Duke, A. Levskaya, S. Ghemawat, S. Dev, H. Michalewski, X. Garcia, V. Misra, K. Robinson, L. Fedus, D. Zhou, D. Ippolito, D. Luan, H. Lim, B. Zoph, A. Spiridonov, R. Sepassi,

D. Dohan, S. Agrawal, M. Omernick, A. M. Dai, T. S. Pillai, M. Pellat, A. Lewkowycz, E. Moreira, R. Child, O. Polozov, K. Lee, Z. Zhou, X. Wang, B. Saeta, M. Diaz, O. Firat, M. Catasta, J. Wei, K. Meier-Hellstern, D. Eck, J. Dean, S. Petrov, and N. Fiedel. PaLM: Scaling language modeling with pathways. (arXiv:2204.02311).

[11] T. Dettmers, M. Lewis, Y. Belkada, and L. Zettlemoyer. LLM.int8(): 8-bit matrix multiplication for transformers at scale. (arXiv:2208.07339).

[12] A. Esteva, A. Robicquet, B. Ramsundar, V. Kuleshov, M. DePristo, K. Chou, C. Cui, G. Corrado, S. Thrun, and J. Dean. A guide to deep learning in healthcare. *Nature medicine*, 25(1):24–29, 2019.

[13] B. Feng, L. Qin, Z. Zhang, Y. Ding, and S. Chu. ZEN: An optimizing compiler for verifiable, zero-knowledge neural network inferences. (2021/087). Publication info: Preprint. MINOR revision.

[14] E. Frantar and D. Alistarh. Sparsegpt: Massive language models can be accurately pruned in one-shot. In *International Conference on Machine Learning*, pages 10323–10337. PMLR, 2023.

[15] Z. Ghodsi, T. Gu, and S. Garg. SafetyNets: Verifiable Execution of Deep Neural Networks on an Untrusted Cloud. June 2017. arXiv:1706.10268 [cs].

[16] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. Delegating computation: Interactive proofs for muggles. 2018.

[17] I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. MIT press, 2016.

[18] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. (arXiv:1510.00149), .

[19] S. Han, J. Pool, J. Tran, and W. J. Dally. Learning both weights and connections for efficient neural networks. (arXiv:1506.02626), .

[20] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. (arXiv:1512.03385).

[21] W. Hu, T. Liu, Y. Zhang, Y. Zhang, and Z. Zhang. Parallel zero-knowledge virtual machine.

[22] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5mb model size. (arXiv:1602.07360).

[23] D. Kang, T. Hashimoto, I. Stoica, and Y. Sun. Scaling up trustless DNN inference with zero-knowledge proofs. (arXiv:2210.08674), .

[24] D. Kang, T. Hashimoto, I. Stoica, and Y. Sun. ZK-IMG: Attested images via zero-knowledge proofs to fight disinformation. (arXiv:2211.04775), .

[25] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc.

[26] M. Labs. The cost of intelligence: Proving machine learning inference with zero-knowledge. .

[27] M. Labs. Scaling intelligence: Verifiable decision forest inference with remainder. .

[28] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[29] S. Lee, H. Ko, J. Kim, and H. Oh. vCNN: Verifiable convolutional neural network based on zk-SNARKs. (2020/584). Publication info: Preprint. MINOR revision.

[30] F. Li, B. Liu, X. Wang, B. Zhang, and J. Yan. Ternary weight networks. (arXiv:1605.04711).

[31] T. Liang, J. Glossner, L. Wang, S. Shi, and X. Zhang. Pruning and quantization for deep neural network acceleration: A survey. June 2021.

[32] J. Lin, W.-M. Chen, Y. Lin, C. Gan, S. Han, et al. Mcunet: Tiny deep learning on IoT devices. *Advances in Neural Information Processing Systems*, 33:11711–11722, 2020.

[33] T. Liu, X. Xie, and Y. Zhang. zkCNN: Zero knowledge proofs for convolutional neural network predictions and accuracy. (2021/673). Publication info: Published elsewhere. CCS 2021.

[34] C. Lund, L. Fortnow, H. Karloff, and N. Nisan. Algebraic methods for interactive proof systems. *Journal of the ACM*, 39(4):859–868, Oct. 1992. ISSN 0004-5411.

[35] S. Ma, H. Wang, L. Ma, L. Wang, W. Wang, S. Huang, L. Dong, R. Wang, J. Xue, and F. Wei. The era of 1-bit LLMs: All large language models are in 1.58 bits.

[36] R. Miotto, F. Wang, S. Wang, X. Jiang, and J. T. Dudley. Deep learning for healthcare: review, opportunities and challenges. *Briefings in bioinformatics*, 19(6):1236–1246, 2018.

[37] Modulus. Chapter 14: The world's 1st on-chain LLM. URL https://medium.com/@ModulusLabs/chapter-14-the-worlds-1st-on-chain-llm-7e389189f85e.

[38] OpenAI. GPT-4 technical report. (arXiv:2303.08774).
[39] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever. Improving language understanding by generative pre-training. .
[40] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. Language models are unsupervised multitask learners. .
[41] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. XNOR-net: ImageNet classification using binary convolutional neural networks. (arXiv:1603.05279).
[42] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer. High-resolution image synthesis with latent diffusion models. (arXiv:2112.10752).
[43] S. Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. 2019. Publication info: A minor revision of an IACR publication in CRYPTO 2020.
[44] S. Setty, J. Thaler, and R. Wahby. Unlocking the lookup singularity with Lasso. 2023.
[45] H. Sun, J. Li, and H. Zhang. zkllm: Zero knowledge proofs for large language models. *arXiv preprint arXiv:2404.16109*, 2024.
[46] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. (arXiv:1409.3215).
[47] J. Thaler. Time-Optimal Interactive Proofs for Circuit Evaluation. 2013. Publication info: Published elsewhere. This is the full version of a Crypto 2013 paper by the same title.
[48] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. C. Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M.-A. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom. Llama 2: Open foundation and fine-tuned chat models. (arXiv:2307.09288).
[49] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. (arXiv:1706.03762).
[50] R. S. Wahby, Y. Ji, A. J. Blumberg, a. shelat, J. Thaler, M. Walfish, and T. Wies. Full accounting for verifiable outsourcing. 2017. Publication info: Published elsewhere. Major revision. CCS 2017.
[51] H. Wang, S. Ma, L. Dong, S. Huang, H. Wang, L. Ma, F. Yang, R. Wang, Y. Wu, and F. Wei. BitNet: Scaling 1-bit transformers for large language models. (arXiv:2310.11453).
[52] C. Weng, K. Yang, X. Xie, J. Katz, and X. Wang. Mystique: Efficient conversions for zero-knowledge proofs with applications to machine learning. (2021/730). Publication info: Published elsewhere. Minor revision. USENIX Security 2021.
[53] G. Xiao, J. Lin, M. Seznec, H. Wu, J. Demouth, and S. Han. SmoothQuant: Accurate and efficient post-training quantization for large language models. (arXiv:2211.10438).
[54] T. Xie, J. Zhang, Y. Zhang, C. Papamanthou, and D. Song. Libra: Succinct Zero-Knowledge Proofs with Optimal Prover Computation. 2019. Publication info: A minor revision of an IACR publication in CRYPTO 2019.
[55] Z. Xing, Z. Zhang, J. Liu, Z. Zhang, M. Li, L. Zhu, and G. Russello. Zero-knowledge Proof Meets Machine Learning in Verifiability: A Survey. Oct. 2023. arXiv:2310.14848 [cs].
[56] J. Zhang, Z. Fang, Y. Zhang, and D. Song. Zero knowledge proofs for decision tree predictions and accuracy. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 2039–2053. ACM. ISBN 978-1-4503-7089-9.

A SpaSum: Sparsity-aware Sumcheck

In this section, we focus on sparsity setting and analyze sparsity-aware sumcheck protocols. Similar setting has been considered in the literature (e.g., Spartan and Lasso [43, 44]) and we complement them with our algorithm called SpaSum.

Assume f is sparse on $\{0,1\}^\ell$: it evaluates to nonzero only on $nz(f) \subseteq \{0,1\}^\ell$. Then there are $|nz(f)|$ nonzero entries of initialized bookkeeping table g_1. Then we have the following lemma:

Lemma 8. *At i-th iteration, while the original bookkeeping table g_i is of size $2^{\ell-i+1}$, there should be at most $\min(2^{\ell-i+1}, |nz(f)|)$ nonzeros.*

Proof. The bookkeeping procedure will not increase number of nonzero entries in the bookkeeping table, thus the number of nonzeros should be always at most $|nz(f)|$ as g_1. Meanwhile, by condensing the bookkeeping table by half at each iteration, the upper limit of number of entries should be 2^{l-i+1}. Thus, for g_i there should be at most $\min(2^{l-i+1}, |nz(f)|)$ nonzeros.

□

Since the nonzero entries are limited for g_i, if $|nz(f)|$ is small enough then it is possible for the original sumcheck to execute in less total time than $\mathcal{O}(2^\ell)$. To show this, we define an auxiliary set

$$\mathcal{T}_i = \{(b_i, \ldots, b_l) \in \{0,1\}^{l-i+1} | \exists (b_1, \ldots, b_{i-1}) \in \{0,1\}^{i-1} \ \ s.t. \ \ f(b_1, \ldots, b_l) \neq 0\}$$

Then we have the following lemma:

Lemma 9. *For bookkeeping table g_i, it suffices to maintain $\mathcal{T}_i \subseteq \{0,1\}^{\ell-i+1}$ entries for g_i since only these entries could possibly be nonzeros while for all other $\{0,1\}^{\ell-i+1} \backslash \mathcal{T}_i$, g_i will evaluate to 0.*

Proof. The proof could be done by a simple induction. For $i = 1$ this is true by definition of f and g_1. Assume that it is true for $i \leq k$. For $i+1$,

$$g_{i+1}(b_{i+1}, \ldots, b_\ell) = (1 - r_i) \cdot g_i(0, b_{i+1}, \ldots, b_\ell) + r_i \cdot g_i(1, b_{i+1}, \ldots, b_\ell)$$

If $(b_{i+1}, \ldots, b_\ell) \notin \mathcal{T}_i$, then

$$g_i(0, b_{i+1}, \ldots, b_\ell) = g_i(1, b_{i+1}, \ldots, b_\ell) = 0$$

This leads to $g_{i+1}(b_{i+1}, \ldots, b_\ell) = 0$, which completes the proof. □

Theorem 10. *The SpaSum algorithm where f's sparsity is characterized by $nz(f)$ will run in time $\mathcal{O}((c+1) \cdot |nz(f)|)$, where $c = \ell - \log(|nz(f)|)$*

Proof. Given Lemma. 9, in each iteration we only compute and maintain $\mathcal{T}_i$ entries of g_i, where $|\mathcal{T}_i| \leq \min(2^{\ell-i+1}, |nz(f)|)$. Thus in the i-th iteration,

$$\begin{aligned} f_i(0) &= \sum_{b_{i+1}, \ldots, b_\ell \in \mathcal{T}_{i+1}} g_i(0, b_{i+1}, \ldots, b_\ell) \\ f_i(1) &= \sum_{b_{i+1}, \ldots, b_\ell \in \mathcal{T}_{i+1}} g_i(1, b_{i+1}, \ldots, b_\ell) \\ f_i(r_i) &= (1 - r_i) \cdot f_i(0) + r_i \cdot f_i(1) \end{aligned}$$

The time spent on the i-th iteration is $\mathcal{O}(\min(2^{\ell-i+1}, |nz(f)|)$. For the first c iterations where the size of bookkeeping table is greater than $|nz(f)|$, the time cost for each iteration is bounded by $\mathcal{O}(|nz(f)|)$. For the last $(\ell - c)$ iterations, the time cost for each iteration is bounded by $|nz(f)|, \frac{|nz(f)|}{2}, \frac{|nz(f)|}{4}, \ldots$. Summing them up, the time for sumcheck protocol becomes

$$c \cdot \mathcal{O}(|nz(f)|) + \mathcal{O}(|nz(f)|) + \frac{\mathcal{O}(|nz(f)|)}{2} + \frac{\mathcal{O}(|nz(f)|)}{4} + \ldots = \mathcal{O}((c+1) \cdot |nz(f)|)$$

□

Theorem 11. *In Lasso [44] the authors also consider the case of proving sparse vector inner products. They propose an algorithm that runs at order of*

$$\mathcal{O}(c' \cdot |nz(f)|)$$

where $c' = \frac{\ell}{\log(|nz(f)|)}$

A.1 Generalizations

We consider a generalization where $f^{(1)}, \ldots, f^{(k)}$ be ℓ-variate multilinear polynomials whose sparsity is characterized by $nz(f^{(i)})$. Let $f = \prod_{i=1}^{k} f^{(i)}$. We have the following lemma:

Lemma 12. *Applying the SpaSum to $f = \prod_{i=1}^{k} f^{(i)}$, the total running time would be*

$$\mathcal{O}\left(\sum_{i=1}^{k}(c_i+1)\cdot|nz(f^{(i)})|\right)$$

where

$$c_i = \ell - \log(|nz(f^{(i)})|)$$

Proof. For each $i \in [k]$, there are $|nz(f^{(i)})|$ nonzeros in the initial i-th bookkeeping table. It would suffice for the i-th bookkeeping table to keep track of these nonzeros during the sumcheck process. □

A useful special case of aforementioned generalization is to prove the inner product of two vectors

$$\sum_{b_1,\ldots,b_\ell\in\{0,1\}} f(b_1,\ldots,b_\ell)\cdot g(b_1,\ldots,b_\ell)$$

If both of them are dense on the hypercube, SpaSum will run in $\mathcal{O}(\ell)$ time, matching that of vanilla sumcheck. If their sparsities are characterized by $nz(f)$ and $nz(g)$ respectively, then the running time will become

$$\mathcal{O}\left((c_f+1)|nz(f)| + (c_g+1)|nz(g)|\right)$$

where $c_* = \ell - \log(|nz(*)|$ which will potentially be much faster than the vanilla (dense) sumcheck.