Sparse-zkVC: Sparsity-Aware R1CS Synthesis for Zero-Knowledge Proofs of Sparse LLM Inference

Yanzhe Cheng*, Luxing Yang†, Wei Ren**
*Southwest University, Chongqing, China
Email (Yanzhe Cheng): orangeeeeee@email.swu.edu.cn
†Deakin University, Australia
*Corresponding Author: Wei Ren

***Abstract*—Zero-knowledge proofs for machine-learning inference are often limited by dense arithmetization: even after model pruning, R1CS generators may instantiate witness variables and linear-combination terms according to the original tensor shape. Sparse-zkVC extends zkVC's CRPC+PSQ matrix-multiplication generator with mask-guided synthesis. Given a public static pruning mask, it allocates variables only for retained weights and active prefix-sum states while advancing CRPC coefficients over the original dense index order. This preserves coefficient alignment, keeps the constraint count at the inner dimension d, and reduces the dominant variable term from hdm to $h \cdot NNZ$, where NNZ counts retained weights. The prototype removes unused variables and linear-combination terms during frontend synthesis. On the zkVC-aligned $128 \times 64 \times 49$ benchmark, 99% sparsity reduces variables from 412,736 to 12,191 and improves online proving time by 20.2×, with all generated proofs verifying successfully. Long-context and LLaMA-7B linear-operator stress tests characterize the resulting efficiency gains and remaining single-node memory limits.**

***Index Terms*—Zero-knowledge proofs, verifiable computation, ZKML, LLM pruning, sparsity-aware R1CS synthesis, MSM**

I. Introduction

Large language models (LLMs) increasingly run through cloud APIs, where users need assurance that outsourced inference was executed as claimed. Zero-knowledge machine learning (ZKML), using systems such as zk-SNARKs [1], [2], supports verifiable computation without exposing private parameters or inputs [3], [4]. Proof generation for high-dimensional operators nevertheless remains expensive.

Existing work addresses this cost at several levels. Early ZKML systems encode neural inference using arithmetic circuits and general proof systems [3], [5], [6]. Lasso and Jolt reduce table-heavy computation costs [7], [8], while HyperPlonk and Spartan improve proof-system scalability [9], [10]. At the operator level, zkLLM studies Transformer proof generation [11], and zkVC compresses matrix-multiplication constraints through Constraint-reduced Polynomial Circuits (CRPC) and Prefix-Sum Query (PSQ) [12]. Hardware accelerators further reduce the cost of multi-scalar multiplication (MSM) and fast Fourier transform (FFT) kernels [13], [14], [15].

Model pruning creates an additional opportunity to reduce proving work. SparseGPT, Wanda, SliceGPT, and related methods remove weights or structural components while limiting accuracy loss [16], [17], [18], [19]. In ordinary inference, the resulting masks allow an implementation to skip zero products and reduce memory movement. A ZKML frontend must explicitly incorporate this information into its generated constraints to obtain comparable savings in witness storage.

Many compilation pipelines construct arithmetic representations independently of the assigned values [9], [20]. A dense matrix generator may therefore allocate an intermediate variable for every product $x_{i,k}w_{k,j}$, including weights pruned to zero. In zkVC, CRPC+PSQ reduces the number of constraint rows, but the generator still allocates weights and prefix states across dense tensor dimensions. Those variables enlarge the assignment vector and contribute references to the linear combinations processed during key generation and proving. Reducing the constraint count consequently leaves a separate allocation cost to address.

Sparse-zkVC uses a public static pruning mask during frontend synthesis to remove these zero-weight branches. Its Sparse Dataflow Generator (SDG) allocates retained weights and active prefix states, reuses unchanged prefixes, and advances coefficients over the original dense index order. Figure 1 outlines how the generated circuit and assignment map connect to setup and proving. Because retained terms preserve their original coefficients, the sparse representation can be passed to the existing Groth16 backend. The resulting reduction in witness variables complements improvements to proof protocols and backend kernels.

Our contributions are:

- Mask-guided CRPC+PSQ synthesis that allocates retained weights and active prefix states, reuses unchanged prefixes, and reduces the dominant variable term from $\mathcal{O}(hdm)$ to $\mathcal{O}(h \cdot NNZ)$.
- Coefficient synchronization over the original dense index order, preserving the polynomial positions of retained terms despite sparse allocation.
- Sparsity ablation, thread scaling, long-context banded-attention stress tests, and LLaMA-7B linear-operator stress tests that characterize gains and memory limits.

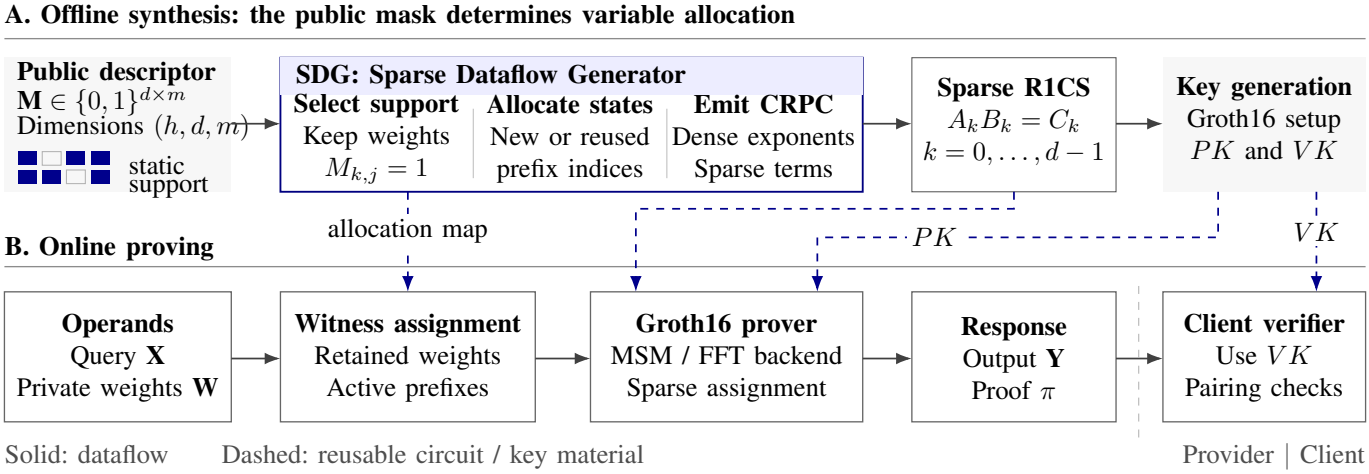


Fig. 1. SDG separates support selection, prefix allocation, and coefficient-preserving emission. Dashed edges carry the reusable allocation map, R1CS, and keys. The online workflow is deployment-oriented; the benchmark keeps matrix values auxiliary.

II. Preliminaries

A. Finite Fields and R1CS

Let $\mathbb{F}_p$ be a prime field and $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ cyclic groups of order p. A non-degenerate, efficiently computable pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ satisfies $e(g_1^a, g_2^b) = e(g_1, g_2)^{ab}$. Such pairings underpin Groth16 [1].

For public input $\mathbf{x}$ and private witness $\mathbf{w}$, an R1CS assignment is $\mathbf{z} = (1, \mathbf{x}, \mathbf{w})$. Matrices $\mathbf{A}, \mathbf{B}, \mathbf{C} \in \mathbb{F}_p^{n_c \times n_v}$ encode n_c constraints over $n_v = 1 + n_{\text{pub}} + n_{\text{priv}}$ variables, including the constant wire. Each row requires

$$(\mathbf{A}_i \cdot \mathbf{z})(\mathbf{B}_i \cdot \mathbf{z}) = \mathbf{C}_i \cdot \mathbf{z}. \tag{1}$$

This representation captures arithmetic multiplication and addition. In Groth16-like backends, multi-scalar multiplication (MSM) operations are a major prover cost, influenced by instantiated variables and nonzero linear-combination terms.

B. CRPC+PSQ Matrix Encoding

For $\mathbf{X} \in \mathbb{F}_p^{h\times d}$ and $\mathbf{W} \in \mathbb{F}_p^{d\times m}$, the output is $\mathbf{Y} = \mathbf{XW} \in \mathbb{F}_p^{h\times m}$. A naive encoding creates $\mathcal{O}(hdm)$ product constraints and intermediate variables. CRPC instead packs, for each inner position k,

$$LC_X^{(k)}(\tau) = \sum_{i=0}^{h-1} x_{i,k}\tau^{im}, \tag{2}$$

$$LC_W^{(k)}(\tau) = \sum_{j=0}^{m-1} w_{k,j}\tau^{j}. \tag{3}$$

Their product expands to

$$LC_X^{(k)}(\tau) LC_W^{(k)}(\tau) = \sum_{i=0}^{h-1}\sum_{j=0}^{m-1} x_{i,k} w_{k,j} \tau^{im+j}. \tag{4}$$

One row thus packs products associated with a fixed k, reducing the constraint count to $\mathcal{O}(d)$, or exactly 64 in the aligned benchmark. PSQ stores output prefix sums so the k-th check constrains the current-minus-previous prefix. Dense CRPC+PSQ still allocates traces at every (i, k, j), retaining $\mathcal{O}(hdm)$ allocated variables despite the reduced row count. This is the frontend bottleneck targeted here.

C. Threat Model and Public Mask

The static pruning mask $\mathbf{M}$ is public architectural metadata; retained weight values may remain private. The prototype constructs private matrix assignments and verifies the generated sparse R1CS. Selected inputs or outputs can be designated as primary inputs using the same mask-guided allocation rule.

We consider privacy adversaries seeking retained values and soundness adversaries exploiting skipped branches to prove an incorrect product. The mask is public and fixed during setup, and Groth16 is assumed sound and zero-knowledge for the emitted R1CS. Section IV discusses the fixed-base CRPC caveat.

III. Sparse-zkVC Design

The Sparse Dataflow Generator (SDG) adds mask-guided synthesis to zkVC's matrix generator, allocating weight and prefix variables at the positions retained by the mask.

A. System Architecture

Fig. 1 separates offline synthesis from online proving. SDG parses the public mask and dimensions, emits sparse R1CS, and generates proving and verification keys. The prover subsequently fills the allocated variables with input values, retained weights, and prefix sums. The optimization reduces variables and linear-combination terms entering Groth16 without changing the backend proof system.

B. Mask-Guided Sparse Synthesis

The public mask $\mathbf{M} \in \{0,1\}^{d\times m}$ has $\mathbf{M}_{k,j} = 1$ for retained weights and zero for pruned weights. SDG uses each mask entry to decide whether to allocate a weight variable and new prefix states. At a pruned position, the prefix retains its previous value and variable index.

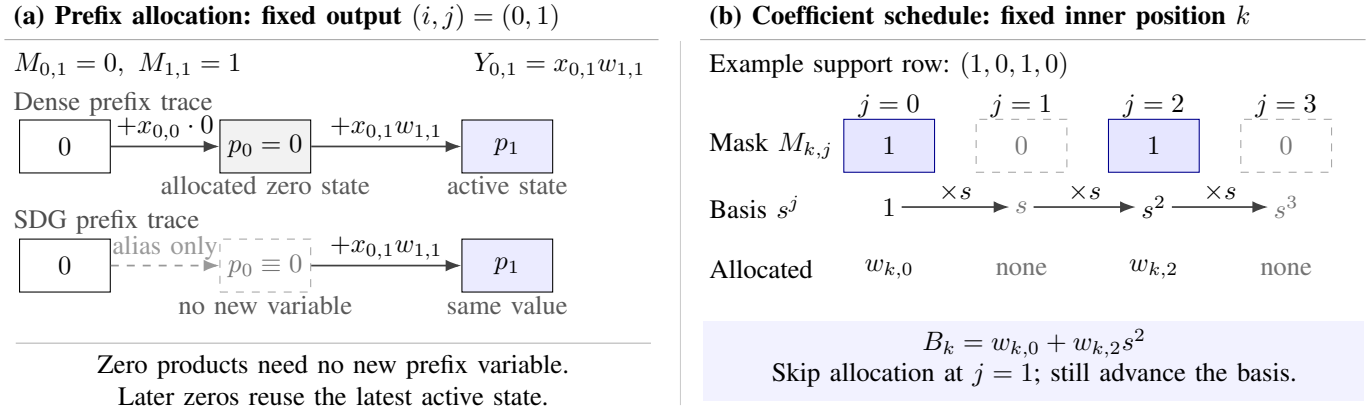


Fig. 2. Sparse-synthesis invariants. (a) The zero product needs no new prefix; the active product retains its value. (b) In a separate support row, omitted weights do not renumber exponents: $j = 2$ retains s^2. Blue fill marks retained positions or states; dashed outlines mark omitted positions or aliases.

Algorithm 1: SDG synthesis for CRPC+PSQ

Input: Dimensions (h, d, m); public mask $\mathbf{M}$
Output: Sparse R1CS and a witness-assignment plan

Allocate dense inputs $x_{i,k}$;
Allocate $w_{k,j}$ iff $\mathbf{M}_{k,j} = 1$;
for *each output position* (i, j) **do**
  $prefix \leftarrow 0$;
  for $k \leftarrow 0$ **to** $d - 1$ **do**
    if $\mathbf{M}_{k,j} = 1$ **then**
      Allocate a new prefix variable;
      Record its update from $prefix$ and $x_{i,k}w_{k,j}$;
      Set $prefix$ to the new variable;
    else
      Reuse the previous state if it exists;
$s \leftarrow 2$; $t \leftarrow s^m$;
for $k \leftarrow 0$ **to** $d - 1$ **do**
  Build $A_k = \sum_i x_{i,k}t^i$;
  Build $B_k = \sum_{j:\mathbf{M}_{k,j}=1} w_{k,j}s^j$;
  Build C_k from active prefix differences with coefficients $t^i s^j$;
  Advance powers over all dense positions, including skipped entries;
  Emit $A_k B_k = C_k$;
return R1CS and assignment plan;

The invariant is that pruning changes allocation only. CRPC powers still advance across all dense column positions j and row blocks i, including skipped entries. Every retained term therefore receives its original coefficient. Algorithm 1 summarizes this conditional allocation and coefficient synchronization.

The algorithm records variable indices and arithmetic updates during synthesis. Witness generation later evaluates these updates using the supplied matrices. This separation allows the public mask to determine the circuit before any private matrix values are assigned.

Witness storage. A witness variable expands the assignment vector and introduces indices used by linear combinations and proving-key material. Omitting zero weights and inactive prefixes shortens the assignment vector and removes their references from the emitted linear combinations.

C. Prefix States and Output Recovery

For each output coordinate (i, j), define the mathematical prefix

$$q_{i,k,j} = \sum_{\ell=0}^{k} M_{\ell,j}x_{i,\ell}w_{\ell,j}, \qquad q_{i,-1,j} = 0. \tag{5}$$

The generator represents this sequence with a variable index for each active update. When $M_{k,j} = 1$, it allocates a variable for $q_{i,k,j}$ and records the preceding prefix needed to assign its value. When $M_{k,j} = 0$, both prefixes use the same index. Leading pruned entries use the zero linear combination until the first active update. Consequently, a run of pruned entries introduces no additional prefix variables.

This representation gives a direct construction of the right-hand linear combination:

$$C_k = \sum_{i=0}^{h-1} \sum_{j:M_{k,j}=1} \left(q_{i,k,j} - q_{i,k-1,j}\right) s^{im+j}. \tag{6}$$

Here each prefix denotes its allocated variable or its reused predecessor. The first active term subtracts zero. Later terms reference the most recent active prefix, even when several inner positions have been skipped. Witness assignment follows the same recurrence, so its update order agrees with the indices used in C_k.

The final prefix represents $Y_{i,j}$. An entirely pruned output column produces zero outputs, and trailing pruned entries reuse the last active prefix. These cases need no additional output variables in the operator benchmark. A deployment that exposes separate output variables must link them to these final prefixes. Such output-binding constraints should be counted separately from the d packed operator rows reported here.

D. Illustrative Example

Consider $\mathbf{Y} = \mathbf{XW}$ for

$$\mathbf{W} = \begin{pmatrix} w_{0,0} & 0 \\ w_{1,0} & w_{1,1} \end{pmatrix}, \qquad \mathbf{M} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}. \tag{7}$$

The output $\mathbf{Y}_{0,1} = \mathbf{X}_{0,0}\mathbf{W}_{0,1} + \mathbf{X}_{0,1}\mathbf{W}_{1,1}$ includes a zero-weight branch. Dense synthesis allocates its temporary variable w_{tmp0}; SDG omits that variable and branch when $\mathbf{M}_{0,1} = 0$. Fig. 2(a) shows the resulting prefix reuse. Panel (b) separately illustrates coefficient synchronization across output columns: skipping a weight does not renumber the exponent of a later retained term.

E. Variable and Constraint Counts

Let $NNZ = \|\mathbf{M}\|_0$ count retained weights. SDG allocates dense inputs, retained weights, and active prefix states, giving the following counts, excluding the constant wire:

$$N_{\text{constraints}} = d, \tag{8}$$

$$N_{\text{vars}} = hd + NNZ + h \cdot NNZ, \tag{9}$$

$$N_{\text{vars}}^{\text{dense}} = hd + dm + hdm. \tag{10}$$

Thus the dominant variable term changes from $\mathcal{O}(hdm)$ to $\mathcal{O}(h \cdot NNZ)$ while the constraint count remains $\mathcal{O}(d)$. At $h = 128, d = 64, m = 49$, dense allocation gives $8192 + 3136 + 401408 = 412736$ variables. A 99% mask retaining 31 weights gives $8192 + 31 + 3968 = 12191$.

F. Sparsity and Remaining Allocation Cost

Let $\rho = NNZ/(dm)$ be the retained-weight fraction. The ratio between sparse and dense variable counts is

$$R_v(\rho) = \frac{hd + (h+1)\rho dm}{hd + (h+1)dm}. \tag{11}$$

Each additional pruned weight removes one weight variable and h prefix variables. The dense input term hd remains allocated, establishing a lower bound on the assignment size for this generator. For the aligned benchmark, the 31 retained weights account for 3,999 variables, while inputs account for 8,192. Inputs therefore occupy about 67.2% of the 99% sparse assignment. Further pruning has diminishing scope to reduce its total size.

The count depends on the number of retained weights; their positions determine which prefixes are reused and which exponents are attached to retained terms. Masks with the same NNZ thus have the same variable count under this allocation rule, although memory access and synthesis time may differ. Moreover, the pseudocode still visits dense index positions to preserve the coefficient schedule. The variable-count reduction alone does not establish an $\mathcal{O}(h \cdot NNZ)$ runtime bound for that traversal.

IV. Security Analysis

A. Correctness of Sparse Synthesis

Active entries retain their weight and prefix variables. Pruned products are fixed to zero by public topology and contribute no sparse linear-combination term. Because coefficient powers advance over the dense index order even when allocation is skipped, each retained weight receives its dense-encoding coefficient. The sparse constraint for k checks the same packed active-product sum as the dense constraint after removing zero products.

More explicitly, initialize both dense and sparse prefix sequences at zero. At an active position, both sequences add the same product; at a pruned position, both add zero and the sparse sequence reuses its current variable. Induction over k therefore gives equal prefix values for an honestly evaluated matrix product. Substituting these values into Eq. 6 yields the dense packed relation with its zero terms removed. This argument establishes preservation of the evaluated relation; the soundness of recovering individual products from a packed check requires the encoding assumptions discussed below.

B. Zero-Knowledge Considerations

The public mask reveals positions by design; only values assigned to retained private variables are protected. SDG uses the existing Groth16 blinding and proof format, so the privacy of the retained witness values follows from the backend property for the emitted public circuit. Evaluation keeps matrix values auxiliary to isolate operator cost. Deployment may expose or commit inputs and outputs without changing pruning logic. Input-dependent masks can reveal execution traces and are outside this construction.

C. Soundness Considerations

A prover must satisfy every emitted sparse CRPC+PSQ constraint. Pruned positions have no witness variables, preventing hidden values there from compensating for retained values. Rejection of incorrect active assignments nevertheless depends on the backend and the absence of coefficient collisions in the chosen CRPC encoding.

The prototype follows zkVC-style CRPC with a fixed evaluation base for performance measurement. A production protocol must establish that its packed checks bind the claimed matrix entries, for example through a justified challenge-and-commitment construction or suitable restrictions on the encoded values. Successful verification in this evaluation establishes satisfaction of the emitted R1CS. Establishing soundness for the full matrix-multiplication statement remains a requirement for deployment.

V. Performance Evaluation

The evaluation examines how mask-guided synthesis affects circuit size, execution time, and the largest operators that can be proved within the available memory. We first vary weight sparsity on a fixed matrix-multiplication benchmark to measure changes in variable allocation, setup time, and online proving time. An independent thread-scaling experiment then examines how backend parallelism affects the remaining cost at high sparsity. Finally, stress tests with LLaMA-7B projection dimensions and increasing attention context lengths identify the operators that fit within a 128 GB memory budget. Together, Tables I–IV show how savings in witness allocation translate into measured proving performance and where further decomposition is needed.

TABLE I
SPARSITY ABLATION ON THE ZKVC-ALIGNED BENCHMARK

Scenario	**Sparsity**	**Constraints**	**Variables**	T_F **(s)**	T_S **(s)**	T_P **(s)**	**Speedup**	**Result**
Dense Base	0%	64	412,736	0.3350	1.1280	0.3120	1.00×	PASS
Sparse-50	50%	64	210,464	0.2180	0.8210	0.1560	1.73×	PASS
Sparse-60	60%	64	169,958	0.0910	0.4880	0.1760	2.42×	PASS
Sparse-70	70%	64	129,452	0.0790	0.4770	0.1300	3.10×	PASS
Sparse-80	80%	64	89,075	0.0550	0.3670	0.0820	4.72×	PASS
Sparse-90	90%	64	48,569	0.0380	0.2320	0.0430	7.99×	PASS
Sparse-95	95%	64	28,316	0.0290	0.1210	0.0200	13.20×	PASS
Sparse-99	99%	64	12,191	0.0230	0.0640	0.0090	20.22×	PASS

TABLE II
THREAD SCALING FOR THE 99% SPARSE $128 \times 64 \times 49$ BENCHMARK.

Threads	**Variables**	T_F **(s)**	T_S **(s)**	T_P **(s)**
1	12,191	0.023	0.340	0.035
2	12,191	0.023	0.257	0.028
4	12,191	0.023	0.197	0.022
8	12,191	0.023	0.129	0.017
16	12,191	0.023	0.099	0.014

A. Experimental Setup

We evaluate the Sparse-zkVC prototype through a dedicated matrix-multiplication benchmark program that runs circuit construction, key generation, proof generation, and verification for each configuration. Mask-guided allocation occurs during frontend R1CS construction, before proving. The backend uses BN128 and Groth16. PASS denotes satisfaction of the generated R1CS and successful proof verification. We denote frontend time by T_F, setup time by T_S, and backend proving time by T_P. The reported online metric is $T_F + T_P$, excluding T_S.

B. zkVC-Aligned Sparsity Ablation

Table I reports one run per sparsity level with $h = 128, d = 64, m = 49$ and 16 threads. Speedup is calculated as the dense baseline time divided by the sparse time, using $T_F + T_P$ for both and excluding setup. All rows retain exactly 64 constraints, while variables decrease with retained weights. At 99% sparsity, SDG removes 400,545 variables; online time falls from 0.647 s to 0.032 s (20.22×), and setup from 1.128 s to 0.064 s as the key is generated over a smaller topology. At 50–80% sparsity, online speedups range from 1.73× to 4.72×.

The variable count falls by approximately 33.86×, exceeding the 20.22× reduction in combined online time. Frontend and backend proving times decrease by 14.57× and 34.67×, respectively. At 99% sparsity, the frontend contributes 0.023 s of the 0.032 s online metric, or about 71.9%. Once the witness is small, further reductions in backend work affect a smaller share of the measured total.

The 50–80% cases show gains at less aggressive pruning levels. Backend time rises from 0.156 s at 50% sparsity to 0.176 s at 60%, even though fewer variables are allocated. Single runs provide no estimate of timing variability, and model accuracy requires separate evaluation.

C. Thread Scaling at 99% Sparsity

Table II reports an independent experiment with 1 to 16 threads. With fixed sparse topology, variable count and frontend time remain stable, whereas setup and backend proving benefit from parallelism.

From one to sixteen threads, setup time decreases from 0.340 s to 0.099 s, a 3.43× speedup, and backend proving decreases from 0.035 s to 0.014 s, a 2.50× speedup. With frontend time fixed at 0.023 s, the combined online metric improves from 0.058 s to 0.037 s, or 1.57×. Frontend work therefore limits the combined speedup. Ratios in each independent experiment use its own baseline.

D. Long-Context Banded-Attention Stress Test

The stress test emulates banded attention with head dimension $d = 64$ and window width $W = 256$. The dense variable count scales as $\mathcal{O}(N^2)$, compared with approximately $\mathcal{O}(NW)$ for the banded sparse topology. Table IV reports the context-scaling results: dense proving reaches the 128 GB limit at 2K tokens, while sparse proving reaches 8K and fails at 16K. The experiment measures the context lengths supported by a static locality mask at the matrix-operator level.

Across the successful sparse cases, doubling context length approximately doubles the variable count. Proving time grows from 15.35 s at 1K to 86.81 s at 8K, a factor of 5.66 over an eightfold increase in context length. At 1K, where both runs succeed, the sparse case uses about one quarter of the variables and reduces proving time by 4.66×. At larger contexts, OOM prevents a timing comparison with the dense baseline.

E. LLaMA-7B Linear-Operator Stress Test

Table III stress-tests LLaMA-7B layer dimensions at batch size 32 and 70% sparsity. Attention projections are feasible

TABLE III

LLaMA-7B SINGLE-LAYER LINEAR BACKBONE STRESS TEST AT BATCH SIZE 32 AND 70% SPARSITY. ATTENTION PROJECTIONS ARE FEASIBLE ON ONE 128 GB NODE; FFN MATRICES STILL REQUIRE BLOCKWISE PROVING OR DISTRIBUTED AGGREGATION.

Operator	**Shape** $(h \times d \times m)$	**Dense scale**	**Sparse count**	**Status /** T_P
W_q Query	$32 \times 4096 \times 4096$	∼536M	166,225,484	PASS / 126.41s
W_k Key	$32 \times 4096 \times 4096$	∼536M	166,225,484	PASS / 122.99s
W_v Value	$32 \times 4096 \times 4096$	∼536M	166,225,484	PASS / 127.25s
W_o Output	$32 \times 4096 \times 4096$	∼536M	166,225,484	PASS / 133.25s
W_{gate} FFN	$32 \times 4096 \times 11008$	∼1.44B	∼433M	OOM
W_{up} FFN	$32 \times 4096 \times 11008$	∼1.44B	∼433M	OOM
W_{down} FFN	$32 \times 11008 \times 4096$	∼1.44B	∼433M	OOM

TABLE IV

LONG-CONTEXT ATTENTION STRESS TEST ON A 128 GB NODE. DENSE PROVING FAILS BEYOND 1K TOKENS, WHILE SPARSE-ZKVC VERIFIES UP TO 8K TOKENS UNDER A 256-WINDOW BANDED MASK.

Context Length	**Dense 0% Sparsity**		**Sparse-zkVC 256-Window Mask**		
N	**Variables**	**Prover (s)**	**Variables**	**Prover (s)**	**Status**
1K	67,239,936	71.46	16,859,136	15.35	PASS
2K	OOM	–	33,701,888	29.09	PASS
4K	OOM	–	67,387,392	49.71	PASS
8K	OOM	–	134,758,400	86.81	PASS
16K	OOM	–	OOM	–	OOM

on one 128 GB node; larger FFN matrices exceed its memory budget despite pruning. The four projection runs allocate the same number of variables and have backend proving times between 122.99 s and 133.25 s. The dense-scale column reports the leading hdm term; the approximate FFN sparse counts report the leading $h \cdot NNZ$ term. Exact projection counts include inputs and retained weights. These operator-level measurements motivate blockwise decomposition with recursive or folding-based aggregation.

VI. DISCUSSION

The FFN memory limit motivates partitioning wide matrices into blocks and aggregating their SDG proofs. Additional circuits would handle normalization, activation functions, and consistency between blocks.

A column partition provides one concrete decomposition: write $\mathbf{W} = [\mathbf{W}^{(1)} \ \cdots \ \mathbf{W}^{(B)}]$ and prove the corresponding output blocks $\mathbf{Y}^{(b)} = \mathbf{X}\mathbf{W}^{(b)}$. Each block uses the original input matrix and a subset of weight columns. Smaller blocks reduce the prefix variables needed for an individual proof, while repeated input bindings and per-proof setup increase total work. An aggregation layer must also ensure that all blocks use the same input and the intended model parameters. Block-size selection should account for peak memory, total setup and proving time, and aggregation cost.

VII. LIMITATIONS AND FUTURE WORK

SDG assumes a *static public mask*, matching invariant, non-sensitive pruning topology. Dynamic masks $\mathbf{M}(\mathbf{x})$, such as MoE routing or ReLU-induced activation sparsity, depend on private inputs or hidden states. Exposing them can reveal execution information; the present mechanism must treat dynamically zero values as active wires to preserve data obliviousness.

A future direction combines mask commitments with folding schemes such as Nova [21]. The prover could commit to sparse routing traces and conditionally aggregate active expert subcircuits, extending the approach to dynamic graphs. The aggregation protocol would need to bind each selected subcircuit to the committed routing decisions.

VIII. CONCLUSION

Sparse-zkVC uses a public pruning mask to allocate retained weights and active prefixes while preserving CRPC coefficients. The generator keeps d packed constraints and reduces the dominant variable term to $\mathcal{O}(h \cdot NNZ)$. At 99% sparsity on the $128 \times 64 \times 49$ benchmark, variables decrease from 412,736 to 12,191 and combined frontend and proving time improves by 20.22×. Larger operator tests demonstrate memory savings and identify FFN matrices as candidates for blockwise proving. Complete LLM verification additionally requires nonlinear-operator constraints and bindings between layers.

USE OF GENERATIVE AI

Limited or no AI contribution. Generative AI tools were used only for code debugging and minor technical assistance, such as identifying programming errors and explaining debugging issues. They were not used for developing the research ideas, designing the proposed algorithms, or conducting substantive experimental analysis. All authors take full responsibility for the code, experimental results, and the content of this paper.

REFERENCES

[1] J. Groth, "On the size of pairing-based non-interactive arguments," in *Advances in Cryptology–EUROCRYPT 2016*. Springer, 2016, pp. 305–326.

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

[3] C. Weng, K. Yang, X. Xie, J. Katz, and X. Wang, "Mystique: efficient conversions for zero-knowledge proofs with applications to machine learning," in *USENIX Security Symposium*, 2021, pp. 501–518.

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

[5] S. Lee, H. Ko, J. Kim, and H. Oh, "vCNN: verifiable convolutional neural network based on zk-SNARKs," *IEEE Transactions on Dependable and Secure Computing*, vol. 21, no. 4, pp. 4254–4270, 2024.

[6] T. Liu, X. Xie, and Y. Zhang, "zkCNN: zero knowledge proofs for convolutional neural network predictions and accuracy," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021.

[7] S. Setty, J. Thaler, and R. Wahby, "Unlocking the lookup singularity with Lasso," in *Advances in Cryptology–EUROCRYPT 2024*, 2024.

[8] A. Arun, S. Setty, and J. Thaler, "Jolt: SNARKs for virtual machines via lookups," in *Advances in Cryptology–EUROCRYPT 2024*, 2024.

[9] B. Chen, B. Bünz, D. Boneh, and Z. Zhang, "HyperPlonk: Plonk with linear-time prover and high-degree custom gates," in *Advances in Cryptology–EUROCRYPT 2023*, 2023.

[10] S. Setty, "Spartan: efficient and general-purpose zkSNARKs without trusted setup," in *Advances in Cryptology–CRYPTO 2020*. Springer, 2020, pp. 704–737.

[11] H. Sun, J. Li, and H. Zhang, "zkLLM: zero knowledge proofs for large language models," in *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security*, 2024.

[12] Y. Zhang *et al.*, "zkVC: fast zero-knowledge proof for private and verifiable computing," in *62nd ACM/IEEE Design Automation Conference (DAC)*, 2025.

[13] Y. Zhang *et al.*, "PipeZK: accelerating zero-knowledge proof with a pipelined architecture," in *ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 14–27.

[14] W. Ma *et al.*, "GZKP: a GPU accelerated zero-knowledge proof system," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, vol. 2, 2023, pp. 340–353.

[15] C. Wang and M. Gao, "UniZK: accelerating zero-knowledge proof with unified hardware and flexible kernel mapping," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2025, pp. 1101–1117.

[16] E. Frantar and D. Alistarh, "SparseGPT: massive language models can be accurately pruned in one-shot," in *International Conference on Machine Learning (ICML)*. PMLR, 2023.

[17] M. Sun, Z. Liu, A. Bair, and J. Z. Kolter, "A simple and effective pruning approach for large language models," in *International Conference on Learning Representations (ICLR)*, 2024.

[18] S. Ashkboos, M. L. Croci, M. Gennari do Nascimento, T. Hoefler, and J. Hensman, "SliceGPT: compress large language models by deleting rows and columns," in *The Twelfth International Conference on Learning Representations (ICLR)*, 2024.

[19] Y. Zhang *et al.*, "Plug-and-play: an efficient post-training pruning method for large language models," in *The Twelfth International Conference on Learning Representations (ICLR)*, 2024.

[20] B. Parno, J. Howell, C. Gentry, and M. Raykova, "Pinocchio: nearly practical verifiable computation," in *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[21] A. Kothapalli, S. Setty, and I. Tzialla, "Nova: recursive zero-knowledge arguments from folding schemes," in *Advances in Cryptology–CRYPTO 2022*, 2022.