BatchZK: A Fully Pipelined GPU-Accelerated System for Batch Generation of Zero-Knowledge Proofs

Tao Lu[†,§], Yuxun Chen[†], Zonghui Wang[†,*], Xiaohang Wang[†], Wenzhi Chen[†,*], Jiaheng Zhang[§]
[†]Zhejiang University, China,
{lutao2020, chenyuxunzju, zhwang, xiaohangwang, chenwz}@zju.edu.cn
[§]National University of Singapore, Singapore,
jhzhang@nus.edu.sg

Abstract

Zero-knowledge proof (ZKP) is a cryptographic primitive that enables one party to prove the validity of a statement to other parties without disclosing any secret information. With its widespread adoption in applications such as blockchain and verifiable machine learning, the demand for generating zero-knowledge proofs has increased dramatically. In recent years, considerable efforts have been directed toward developing GPU-accelerated systems for proof generation. However, these previous systems only explored efficiently generating a single proof by reducing latency rather than batch generation to provide high throughput.

We propose a fully pipelined GPU-accelerated system for batch generation of zero-knowledge proofs. Our system has three features to improve throughput. First, we design a pipelined approach that enables each GPU thread to continuously execute its designated proof generation task without being idle. Second, our system supports recent efficient ZKP protocols with their computational modules: sum-check protocol, Merkle tree, and linear-time encoder. We customize these modules to fit our pipelined execution. Third, we adopt a dynamic loading method for the data required for proof generation, reducing the required device memory. Moreover, multi-stream technology enables the overlap of data transfers and GPU computations, reducing overhead caused by data exchanges between host and device memory.

We implement our system and evaluate it on various GPU cards. The results show that our system achieves more than 259.5× higher throughput compared to state-of-the-art GPU-accelerated systems. Moreover, we deploy our system in the verifiable machine learning application, where our system generates 9.52 proofs per second, successfully achieving sub-second proof generation for the first time in this field.

1 Introduction

Zero-knowledge proof (ZKP) [19] is a cryptographic primitive that enables one party to prove the validity of a statement to other parties without disclosing any secret information. For example, a machine-learning service provider could claim its outputs are indeed calculated from a well-trained model and then employ ZKP to prove the validity of this claim, while keeping the model's parameters confidential. In recent years, ZKP has received much attention from academia [29, 38, 49, 59, 61] and industry [11, 17, 42, 48], leading to its widespread adoption in private-critical applications such as blockchain [31, 46, 60], verifiable machine learning [5, 13, 35], and verifiable program analysis [10, 12].

With the growing deployment of ZKP, the demand for generating zero-knowledge proofs has increased dramatically. A report from Protocol Labs [32] states that the demand is expected to reach a staggering nearly 90 billion zero-knowledge proofs from the time ZKPs start to be applied in real-world applications to 2030. Moreover, generating zero-knowledge proofs is a compute-intensive task. For example, ZENO [13], a start-of-the-art ZKP scheme for verifiable neural networks, requires more than 40 seconds to generate a single proof for the prediction of the VGG-16 model [52] with a CIFAR-10 image [8] as input. Therefore, improving the efficiency of proof generation has become one of the most important topics in expanding the deployment of ZKP in practical applications.

On the one hand, at the theoretical level, a series of recent efficient ZKP protocols [6, 20, 49, 59, 61, 63] have been proposed. These protocols bypass the expensive operations used in previous ZKP protocols [3, 15, 22], such as number-theoretic transform (NTT) and multi-scaler multiplication (MSM). Instead, they utilize cheaper modules to minimize the overhead in proof generation. On the other hand, at the practical level, GPU is a powerful tool to further improve computational efficiency through tens of thousands of execution cores operating in parallel. Considerable efforts [7, 29, 36, 38] have been directed toward developing GPU-accelerated systems for proof generation.

However, these previous GPU-accelerated systems [7, 29, 36, 38] only explored how to efficiently generate a single proof, with the goal of reducing proof generation latency. These systems spend over-abundant GPU resources on individual proof generation, failing to optimize the overall throughput of batch generation. Improving throughput is critical in the industry as it means more proofs to be generated per unit of time, resulting in greater economic benefits. In addition, these systems only cover ZKP protocols [3, 15, 22] that rely on expensive computational modules

§ The first author conducted this work as a visiting scholar at NUS.
∗ Corresponding authers.

Table 1. The dominant computational modules in different ZKP protocols

ZKP Protocols	Year	MSM	NTT	Sum.	Merkle	Encoder
Groth [22]	2016	√	√	×	×	×
Hyrax [56]	2018	√	×	√	×	×
Plonk [15]	2019	√	√	×	×	×
Libra [59]	2019	√	×	√	×	×
Virgo [63]	2020	×	√	√	√	×
Brakedown [20]	2021	×	×	√	√	√
Virgo++ [62]	2021	×	√	√	√	×
Orion [61]	2022	×	×	√	√	√
HyperPlonk [6]	2023	×	×	√	√	√

like NTT and MSM. However, as shown in Table 1, recent ZKP protocols [6, 20, 49, 59, 61, 63] increasingly employ cost-effective modules, such as the sum-check protocol [37], Merkle tree [39], and linear-time encoder [24], to generate proofs. The calculation process of these cost-effective modules is completely different from NTT and MSM. Thus, the previous systems cannot be applied to ZKP protocols dominated by these cost-effective modules.

In summary, the challenges associated to GPU-accelerated systems for batch generation of zero-knowledge proofs are twofold. First, the system needs to provide a suitable scheme for effectively allocating GPU resources, including GPU execution cores and device memory, across different stages of the proving process to improve throughput. Second, the system should be well-suited to the calculation patterns of the modules utilized in recent ZKP protocols.

In this paper, we present a fully pipelined GPU-accelerated system for batch generation of zero-knowledge proofs. Our pipelined approach not only optimizes the utilization of GPU execution cores but also reduces the required device memory compared to the intuitive approach for generating proofs in parallel. Furthermore, to facilitate integration with a wider range of ZKP protocols [6, 20, 49, 59, 61, 63], we adopt a modular design. Specifically, we develop pipelined modules for Merkle tree, sum-check protocol, and linear-time encoder, respectively. These modules can work individually or together to support our fully pipelined ZKP system.

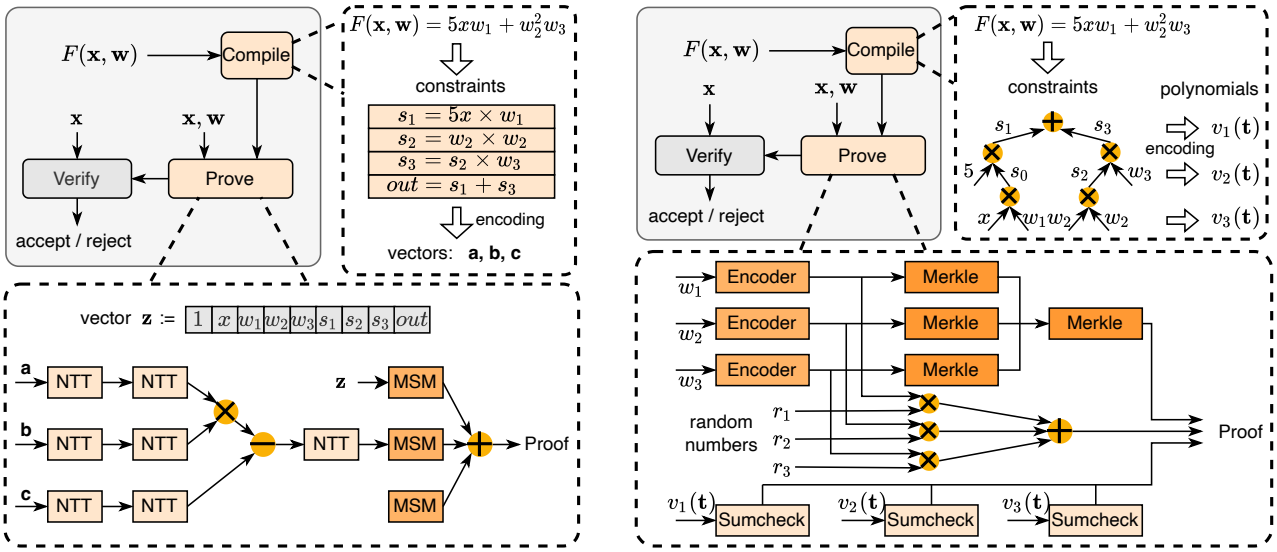
In our system, computational modules required for proof generation are not treated as units. Instead, we divide each module into multiple stages, with each stage processed by a dedicated GPU kernel. Thus, once GPU kernels are launched, they solely focus on completing their assigned tasks and cannot be scheduled to perform other operations. We manage the computation of each module by sequentially forwarding tasks through multiple GPU kernels, and the continuous workflow in this manner establishes the pipeline execution.

Our system achieves three features that improve throughput of proof generation. First, by dedicating each GPU kernel to a fixed task, we can precisely tailor the number of threads assigned to each kernel based on the scale of the task being processed. In addition, as threads employed by GPU kernels cannot be scheduled for other operations, they can be forced to continuously perform their designated tasks without idling. Second, our system supports recent efficient ZKP protocols with their computational modules: sum-check protocol, Merkle tree, and linear-time encoder. We customize these modules to fit the pipeline execution, improving their throughput. Third, the pipeline strategy enables our system to utilize a dynamic loading and storing method, reducing the required device memory. Specifically, our system only loads the data for a single proof in each cycle and dynamically transfers temporarily unneeded intermediate results back to host memory. Moreover, multi-stream technology enables the simultaneous execution of data transfers and GPU computations, reducing overhead caused by frequent data exchanges between host and device memory.

We deploy our pipelined system in the verifiable machine learning application. Unlike the traditional machine learning service, where service providers directly return prediction results to their customers without any integrity guarantee, the verifiable machine learning application requires service providers to additionally employ our system to generate proofs, convincing customers that the prediction results are correctly calculated from a well-trained model. Our pipelined system provides high throughput for proof generation, which is well-suited to this setting, where service providers need to continuously process customer inputs that come in like a flowing stream. As a result, our system can generate 9.52 proofs per second on an Nvidia GH200 card for the prediction of VGG-16 model [52] with CIFAR-10 images [8] as input. Remarkably, this is the first time that sub-second proof generation for verifiable machine learning has been achieved.

The following is a summary of our contributions:

- We use pipeline technology to improve the throughput of three computational modules on GPUs: Merkle tree, sum-check protocol, and linear-time encoder. These modules are increasingly being adopted in efficient ZKP protocols due to their cost-efficiency.
- We propose a fully pipelined GPU-accelerated system for batch generation of zero-knowledge proofs. By adopting recent efficient ZKP protocols and providing a suitable scheme for GPU resource allocation, our system achieves 259.5× higher throughput compared to state-of-the-art GPU-accelerated systems.
- We deploy our system in the verifiable machine learning application, where our system generates 9.52 proofs per second for the prediction of VGG-16 model with CIFAR-10 images as input, successfully achieving sub-second proof generation for the first time in this field.



(a) The first category of ZKP protocols (other works)

(b) The second category of ZKP protocols (our work)

Figure 1. Two categories of ZKP protocols.

2 Background

2.1 Zero-Knowledge Proof and its Applications

Zero-knowledge proof (ZKP) [19] is a cryptographic primitive that enables a prover to generate a proof π, proving to verifiers that a computation $y = F(x, w)$ is correctly calculated using a public input x and the prover's secret input w. With the function F capable of representing any arbitrary computation, ZKP has been found to play important roles in various privacy-critical applications [5, 10, 12, 13, 35, 46, 60].

However, generating zero-knowledge proofs is a compute-intensive task. To improve the efficiency of proof generation, a series of efficient ZKP protocols [6, 20, 61] have been proposed. These protocols bypass the expensive operations used in previous ZKP protocols [3, 15, 22], such as number-theoretic transform (NTT) and multi-scaler multiplication (MSM). Instead, they utilize cheaper sum-check protocol [37], Merkle trees [39], and linear-time encoders [24] to reduce the computational overhead. Figure 1 illustrates the workflow of two categories of ZKP protocols. Different from other GPU-accelerated systems [29, 36, 38] tailored for the first category of ZKP protocols, our work focuses on accelerating the second category. Notably, both two types of ZKP protocols belong to zkSNARK, a family of ZKP protocols with the three properties: (1) non-interactive: only a single message from the prover to the verifier; (2) zero-knowledge: the proof disclosing nothing about the prover's secret input; (3) succinctness: small proof size and fast verification. Compared to the first category of ZKP protocols, the proof size of the second category is relatively larger and reaches several MB.

The acceleration of ZKP protocols benefits many ZKP applications. Verifiable machine learning [5, 13, 35] is one of the most promising applications, where ZKP enables customers to verify that the output provided by the cloud vendor is indeed calculated from a particular model. In this verifiable computation $y = F(x, w)$, the function F represents the inference process of machine learning, with x and y being the input and output of the model. The prover's secret input w is the model parameters, which are considered intellectual property and kept secret from customers. Since the large scale of F in verifiable machine learning can lead to significant computational costs in proof generation, its hardware acceleration will be of great help to practical deployment. Another application is zkBridge [60], which utilizes ZKPs to prove the validity of cross-chain transactions in cryptocurrencies, avoiding trusting centralized committees. zkBridge service providers charge a handling fee for each transaction. Thus, generating more proofs for transactions per unit time (throughput) brings more income. Other applications include zkEVM [48] for extending Ethereum's capabilities and verifiable vulnerabilities [10] to prove the existence of vulnerabilities without revealing their location.

2.2 Merkle Tree

Merkle tree is a structure used to generate the hash value of input data and verify the integrity to ensure the data is undamaged and unaltered. Merkle tree serves as the fundamental building block in various applications, such as Btrfs File System [45], Amazon DynamoDB [53], and Blockchain [4]. In this paper, it is employed as a computational module used in the ZKP protocols that we focus on. The method to construct a Merkle tree is shown in Figure 2.

Specifically, it first divides input data into multiple blocks and then hashes each block to create the leaf node. Usually,

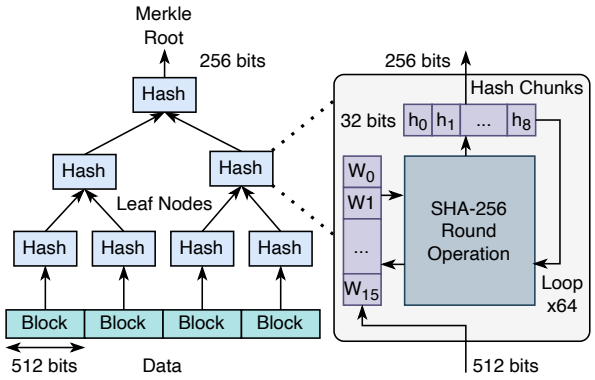


Figure 2. Merkle tree.

the hash method can be a cryptographic hash function such as SHA-256 [41], which produces a fixed-size 256-bit hash value from a 512-bit block. Next, it iteratively combines pairs of these hash values, converting the result to form the next layer of the tree. Continue this process until only one hash value remains at the top, known as the Merkle root, which is the global hash value of input data. The efficient verifiability is a characteristic of this hierarchical structure, as any change in the input data will alter the corresponding hash value and propagate up, ultimately changing the Merkle root.

2.3 Sum-check Protocol

The sum-check protocol [37] is a two-party protocol used to efficiently verify that the sum of a polynomial p over all points in the Boolean hypercube $\{0,1\}^n$ equals a certain value H with the formula $H = \sum_{(x_1,\ldots,x_n)\in\{0,1\}^n} p(x_1,\ldots,x_n)$. The literature [55] proposed an algorithm allowing one party to generate a sum-check proof in $O(2^n)$ time complexity. Based on this proof, the other party can verify that the sum is correct in $O(n)$ time complexity. Given that the time required to generate sum-check proofs is exponentially longer than the time required to verify, our work focuses on accelerating the proof generation process.

Algorithm 1 demonstrates the process of the sum-check proof generation. Briefly, the algorithm maintains a table throughout its execution. Initially, this table has a size of 2^n, with each entry containing the evaluation of the polynomial p over the points in the Boolean hypercube $\{0,1\}^n$. Next, this algorithm executes for n rounds. At each round, the table is updated based on a random number, and the size of the updated table is reduced to half of its original size. Continue this process until the table has fewer than two entries. The sum-check proof comprises two sums of the half-table entries at each round.

2.4 Linear-time Encoder

The linear-time encoder is a family of algorithms that can process and encode data into an error-correcting code in $O(N)$ time complexity with respect to the data size N. The linear-time encoder employed in ZKP protocols [6, 20, 61] is the Spielman encoder [54]. The encoding process of the Spielman encoder is shown in Figure 3.

Specifically, the Spielman encoder has a recursive encoding process consisting of a sequence of execution stages. As shown in Figure 3, each execution stage involves the use of two bipartite graphs, and each bipartite graph can be represented by a sparse matrix, where right vertices correspond to rows of the matrix and left vertices correspond to columns. A non-zero entry in the sparse matrix represents an edge between two vertices in the bipartite graph.

Once all bipartite graphs are represented as sparse matrices, the encoding process starts its recursive process. In each execution stage, it first performs the vector-matrix multiplication between the input vector and the sparse matrix converted from the first bipartite graph. The resulting vector from this multiplication is then forwarded to the subsequent stage, where it performs the same operations as in every stage and returns a vector. Afterward, back to the current stage, it executes another vector-matrix multiplication between the vector returned from the subsequent stage and the matrix converted from the second bipartite graph. The result, combined with the input vector and the vector returned from the subsequent stage, forms the output of this stage.

Algorithm 1 Sum-check Proof Generation [55]

Input: A multi-linear polynomial $p(x_1, x_2, ..., x_n)$. A initial table $\mathbf{A}$ of size 2^n with $\mathbf{A}[b] = p(b_1, ..., b_n)$, where $b = \sum_{i=1}^{n} b_i 2^{i-1}$ and $b_i \in \{0,1\}$. Random numbers $r_1, r_2, ..., r_n$.
Output: The proof $\pi = \{(\pi_{11}, \pi_{12}), (\pi_{21}, \pi_{22}), ..., (\pi_{n1}, \pi_{n2})\}$.

```
for i = 1, 2, ..., n do
    π_i1 = 0, π_i2 = 0
    for b = 0, 1, ..., 2^(n−i) − 1 do
        π_i1 = π_i1 + A[b]
        π_i2 = π_i2 + A[b + 2^(n−i)]
        A[b] = (1 − r_i) · A[b] + r_i · A[b + 2^(n−i)]
    end for
end for
return π = {(π_11, π_12), (π_21, π_22), ..., (π_n1, π_n2)}
```

3 Pipelined ZKP Modules on GPU

To make it easy for our system to integrate with a wider range of ZKP protocols [6, 20, 49, 50, 61, 63], we adopt a modular design. We develop GPU-accelerated ZKP modules, including Merkle tree, sum-check protocol, and linear-time encoder, in a pipeline manner. Our proposed approach not only maximizes the utilization of GPU execution cores but also reduces the required device memory compared to the intuitive parallel GPU execution methods [28, 51]. Details are shown in the following sections.

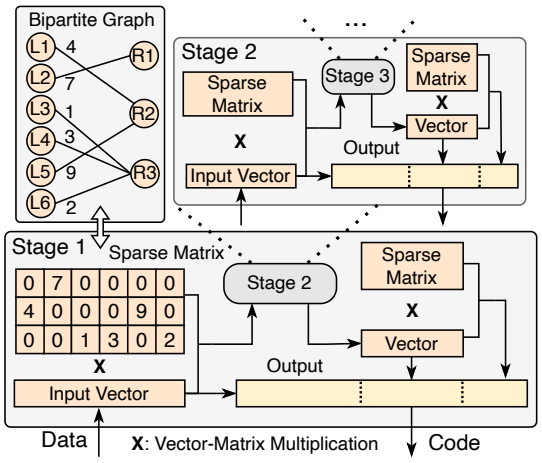
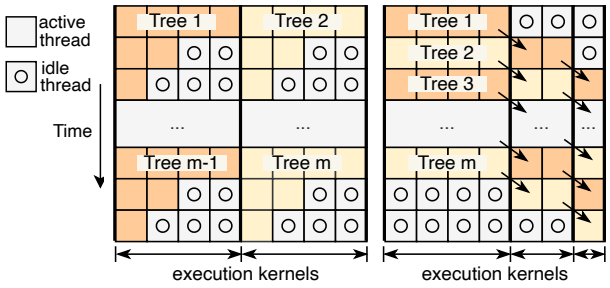


Figure 3. Linear-time encoder.



(a) The intuitive approach **(b)** Our pipelined approach

Figure 4. The workload of GPU cores in a naive method and our pipelined method to generate Merkle trees in batches.

3.1 Pipelined Merkle Tree

In this section, we present our pipelined GPU-accelerated module for batch generation of Merkle trees.

Section 2.2 presents the generation method for a single Merkle tree. For the input data consisting of N 512-bit blocks, the method are performed in $\log N$ rounds. For each round, the hash function like SHA-256 is used to convert the 512-bit blocks from the previous layer of the Merkle tree into 256-bit hash values for the next layer. These execution rounds must be performed serially, because the calculations at each layer depend on the values from the previous layer.

When considering batch generation of multiple Merkle trees, an intuitive method is to launch multiple execution GPU kernels, with each kernel generating a single Merkle tree. Figure 4a shows the workload of GPU threads in this method. Specifically, each GPU kernel requires N threads to generate the first layer of Merkle trees in parallel. However, as the workload required to build subsequent tree layers decreases, many threads become idle until the kernels finish building the entire trees. Re-scheduling these idle threads requires complex control logic and synchronization, which are resource-consuming on GPUs.

To maximize the utilization of GPU execution cores, we propose a pipelined method to batch generate Merkle trees. Figure 4b shows the workload of GPU threads in our method. Instead of building each Merkle tree using a single GPU kernel, we generate Merkle trees by streaming them through multiple GPU kernels, where each kernel is dedicated to the generation of a fixed layer. In this way, multiple layers are generated simultaneously without breaking the serial generation rule in a single tree. Consequently, except for the beginning and end stages of the pipeline, all GPU threads run continuously without idling.

In addition to GPU threads, device memory is another GPU resource that we need to utilize properly. When we generate thousands of Merkle trees in batches, it is a huge burden to load all data blocks of Merkle trees directly into the limited device memory. To reduce the required device memory, we employ a dynamic loading and storing method instead of pre-loading all data blocks for multiple trees. In the pipeline, our method only loads data blocks of a single tree at each time period. Simultaneously, GPU kernels construct Merkle trees for the data blocks that have been in the device memory by executing the hash function. Once the hash values of the next layer of the Merkle tree are calculated, the data for this layer is transferred back to host memory and released from device memory. The hash function iteratively converts 512-bit blocks into 256-bit hash values, effectively halves the memory space required for GPU kernels that generate specific tree layers. Consequently, these GPU kernels require a total of $2N \approx N + \frac{N}{2} + ... + 1$ blocks of device memory. In contrast, the method that loads all data blocks in advance requires mN block space, where m is the number of Merkle trees generated in parallel.

Moreover, we employ multi-stream technology to overlap the process of data transfer between CPU host memory and GPU device memory with hash computations performed by GPU threads. This approach ensures that the generation of Merkle trees within the pipeline occurs simultaneously with data loading and storing. Therefore, no additional time is spent on transferring data between CPU host memory and GPU device memory.

Finally, we optimize the storage structure in the execution of the SHA-256 hash function. As shown in Figure 2, SHA-256 requires its input block to be divided into sixteen 32-bit chunks. These chunks undergo complex SHA-256 round operations to update eight 32-bit hash chunks. Crucially, the size of chunks matches 32-bit registers of GPU execution cores. Therefore, instead of storing these 32-bit chunks in GPU global or shared memory, we force all chunks to be stored in the registers, which are the most efficient storage units in the hierarchical storage architecture of GPUs.

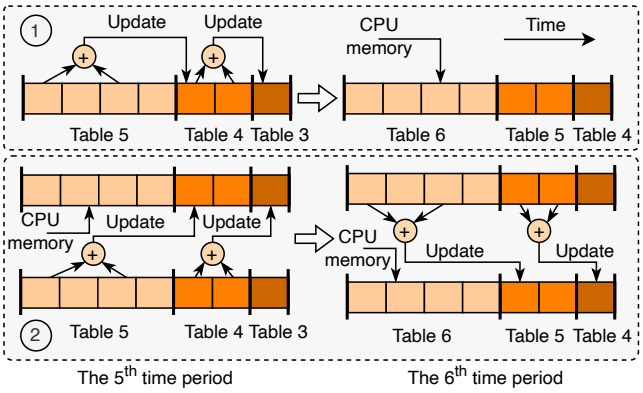


Figure 5. The device memory access patterns in the sum-check proof generation.

3.2 Pipelined Sum-check Protocol

In this section, we present our pipelined GPU-accelerated module for batch generation of sum-check proofs, which are used in sum-check protocols, enabling efficient verification that the sum of a polynomial $p(x_1, x_2, ..., x_n)$ over all points in the Boolean hypercube $\{0, 1\}^n$ equals a certain value.

The literature [55] proposed a method to generate a sum-check proof in $O(2^n)$ time complexity with details shown in Algorithm 1. This algorithm takes a table of size 2^n as input and executes n rounds. Each round begins by computing the sum of the table entries, then updating the table using a random number. After each round, the size of the updated table is reduced to half of its original size.

Obviously, the generation pattern of sum-check proofs closely resembles the pattern of Merkle trees. They both adopt a reduction method that iteratively transforms the original task into a half-scale task. Specifically, as shown in Figure 2, each Merkle tree consists of multiple layers, and the size of each upper layer is half of the lower layer. Thus, the generation of the upper layer is a half-scale task for the lower layer. Similarly, the sum-check in Algorithm 1 consists of multiple rounds, with the computational complexity of the next round being half of the previous round. This similarity allows us to leverage our design strategy from the pipelined module for Merkle trees to develop a pipelined module for sum-check proofs. To optimize GPU core utilization, we generate each sum-check proof by launching n GPU kernels, with each kernel dedicated to a fixed round of execution. By streaming the input tables for sum-check proofs through multiple GPU kernels, we achieve a workload distribution among the GPU threads similar to the pattern shown in Figure 4b, where all threads run continuously without idling except for the beginning and end stages of the pipeline.

However, unlike the computationally intensive task of generating Merkle trees, the generation of sum-check proofs is primarily intensive in terms of memory access. Specifically, when generating Merkle trees, GPU threads need to perform multiple complex SHA-256 round operations. In contrast, when generating sum-check proofs, the operations performed by threads become only several basic addition and multiplication, shifting the performance bottleneck from computational execution to memory access. Therefore, we should reduce memory access as much as possible, especially access to global memory, which is the slowest unit in the GPU hierarchical storage architecture.

In generating sum-check proofs, a part of memory access occurs when updating the input table, which initially holds 2^n entries and is stored in global memory. At each execution round, every GPU core is required to read at least two entries from the table and combine them with a random number to produce a single element. Figure 5 shows two approaches with the least memory access. The first approach iteratively processes the elements in the table and stores the results in the following location for the next sum-check proof. However, this approach could trigger memory race hazards among threads. Therefore, we employ the second approach, where two recyclable memory buffers are allocated to store these tables. As shown in Figure 5, during odd time periods, data is read from the lower buffer and written to the upper buffer. During even time periods, the reverse operation is allowed: reading from the upper buffer and writing to the lower. This alternating use of the two buffers every two periods ensures that reading and writing never occur simultaneously within the same buffer.

The other part of memory access arises when computing the sum of table entries. In this process, the table entries accessed by threads are the same as those accessed during the table updating process, thus eliminating the need for additional global memory access. After loading all table entries, we implement a well-studied sum method [26] to complete the subsequent task. Firstly, we store table entries into multiple shared memories and utilize a tree-based reduction technique to compute partial sums for the elements in these shared memories. Subsequently, the partial results are moved back to global memory, and the above steps are repeated until they are aggregated to produce the final sum.

3.3 Pipelined Linear-time Encoder

In this section, we present our pipelined GPU-accelerated module for batch generation of linear-time codes.

Section 2.4 gives the encoding process for converting input data into a linear-time code. The encoding process comprises a sequence of execution stages, each involving two vector-matrix multiplications. Especially, the vector used in the second vector-matrix multiplication is the output of the subsequent stage. It establishes a recursive dependency where each stage relies on the completion of its subsequent stage.

However, recursive functions are not suitable to perform on GPUs because each recursive call consumes stack space to store local variables and other state information necessary for the function execution. Due to the limited stack memory

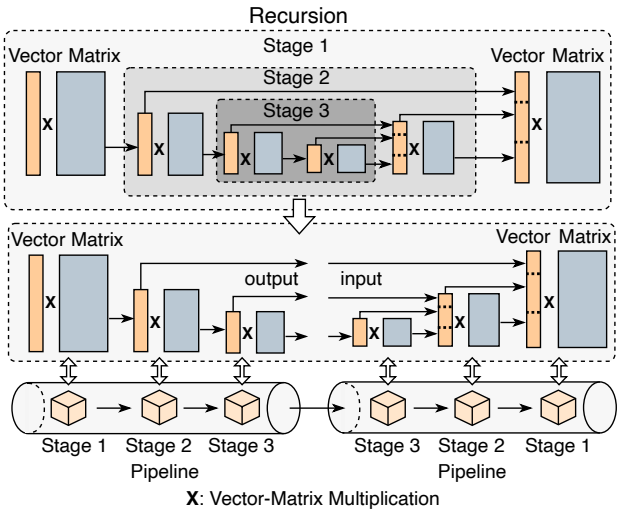


Figure 6. The workflow of the pipelined encoding process.

on GPUs, deeply nested recursive calls can quickly exhaust this space, resulting in stack overflow.

As shown in Figure 6, we split the encoding process into two parts. The first part performs the first vector-matrix multiplication in the encoding stages. In this process, the vector used in vector-matrix multiplication is from the previous stage. Thus, all stages can be executed in a sequential manner, where the size of vector-matrix multiplications decreases gradually. The second part performs the second vector-matrix multiplication, where the vector used in each stage comes from the output of its subsequent stage. In order to prevent recursive execution, we must perform these stages in reverse order, from the small scale to the large scale.

Next, we develop two pipelines to perform two parts of the encoding process, respectively. Each pipeline launches multiple GPU kernels, with each kernel dedicated to vector-matrix multiplication in a fixed stage. As shown in Figure 6, in each cycle, all encoding tasks within the pipeline concurrently execute matrix operations at their current stages, and then progress to their next stages. Two pipelines are interconnected, allowing the output from the first pipeline to feed into the second. The encoding tasks are complete only after they have passed through all stages of both pipelines, ultimately generating the required linear-time codes.

Remember that all matrices involved in the encoding process are converted from bipartite graphs, and they are sparse matrices. While numerous GPU optimization approaches [2, 16, 21] exist for multiplying vectors with sparse matrices, these approaches are tailored to matrices composed of 32-bit and 64-bit integers. In our setting, the elements in matrices are finite field elements, which can be treated as large integers whose bit-width typically ranges from 256 to 768.

In this setting, we can improve the workload balance across threads within GPU wraps. Each GPU wrap is a group of 32 threads that operate in a Single Instruction, Multiple Data (SIMD) manner. Typically, we assign each wrap the task of calculating the product between the vector and 32 matrix rows, with each thread processing a single row. Due to SIMD execution manner, the workload of a wrap is determined by the thread with the heaviest workload among the 32 threads. Therefore, it is crucial to group rows of similar length. In the linear-time encoder, each matrix row contains fewer than 256 non-zero elements, allowing the length of each row to be encoded in a single byte. We sort these row lengths using the bucket sorting algorithm [9], which is the optimal sorting method for data with a few distinct values. Finally, we assign every 32 rows of similar lengths to a wrap, effectively reducing the overhead caused by workload imbalances across threads within GPU wraps. Note that our approach cannot be directly used in matrices composed of regular 32-bit and 64-bit integers, because the high sorting cost relative to regular integer operations makes it uneconomical.

4 Fully Pipelined ZKP System on GPU

In this section, we introduce our fully pipelined ZKP system, designed to achieve two distinct targets compared to previous GPU-accelerated ZKP systems.

On the one hand, our system focuses on the acceleration of ZKP protocols that utilize computational modules, including the sum-check protocol, Merkle tree, and linear-time encoder, which makes our ZKP system more cost-effective than previous ZKP systems that rely on expensive modules like number-theoretic transform (NTT) and multi-scaler multiplication (MSM). On the other hand, our system is tailored to improve throughput in the batch generation of zero-knowledge proofs. In contrast, previous ZKP systems only focus on generating a single proof for the purpose of reducing latency. Improving throughput is critical in the industry as it means more proofs to be generated per unit of time, resulting in greater economic benefits.

Figure 7 outlines our GPU-accelerated ZKP system. At a high level, the workflow of our system follows the proof generation process shown in Figure 1, but with the original modules replaced by our pipelined modules discussed in Section 3. In details, we introduce our ZKP system through the following important features.

The start execution of our system. Initially, we store all the prover's input for proof generation in the CPU host memory, and our system employs a dynamic loading method to reduce the required GPU device memory. At the first execution cycle, our system only loads the prover's input required for the first proof to the device memory. This input is segmented into multiple pieces, each dispatched to the first processing stage of a pipelined linear-time encoder. After each cycle, the prover's input required for the next proof is loaded and dispatched to the pipelined encoders, while the ongoing tasks in our system move to the next stage. This

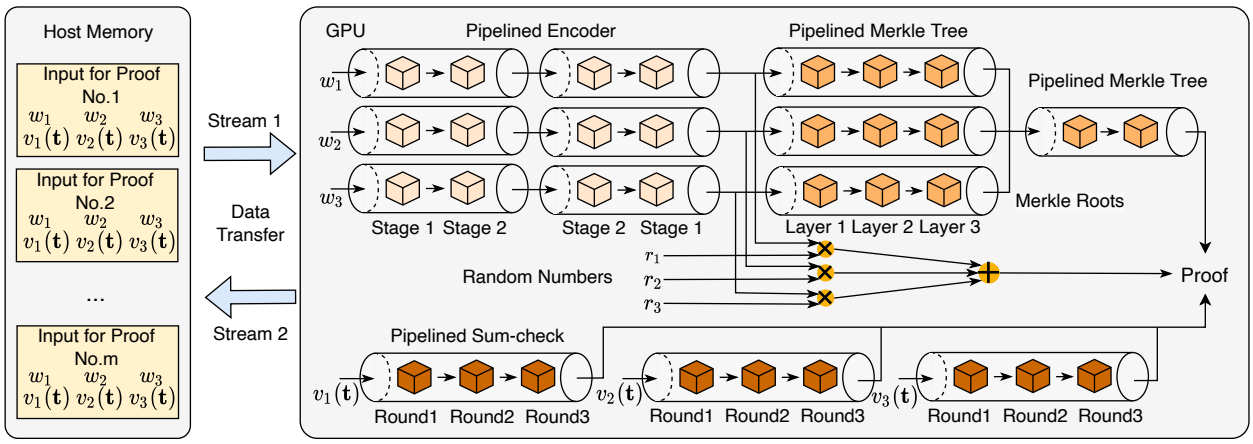


Figure 7. An overview of our fully pipelined GPU-accelerated system for batch generation of zero-knowledge proofs.

sequential process ensures that one proof generation task enters our system every cycle.

The generation task for each proof. Once the required prover's input for the proof generation is fed into the pipeline, our system processes the input through the linear-time encoder, Merkle tree, and sum-check modules in sequence. Specifically, the prover's input is initially divided into multiple segments, each encoded into an error-correction code using a linear-time encoder. Next, the codes are transmitted to Merkle tree modules, which generate multiple Merkle roots. These roots serve as leaf nodes for another Merkle tree module, ultimately yielding a single final root.

Afterward, the following task is carried out by the sum-check modules. The input to sum-check modules is two-fold. The first is random numbers, which are generated by pseudo-random generators using either the final Merkle root or the output from other sum-check modules as a seed. The second is the intermediate results from the proving function. These results are stored in the CPU host memory and encoded into polynomials through Lagrange interpolation [23]. Thus, the sum-check modules are required to load data from host memory in each cycle, similar to the linear-time encoders.

Ultimately, the proof is assembled using the final Merkle root, sum-check proofs, and a linear combination of linear-time codes. The other part of the proof used to verify the correct construction of Merkle trees does not require any calculations and is not depicted here.

The execution of our system at full workload. In our system, proof generation tasks consistently flow into the pipelines. When the first proof generation task is processed through all pipelined modules, our system reaches the full workload state. In this state, all pipelined modules are fully engaged in executing their designated tasks concurrently. In addition, the working manner of these modules, detailed in Section 3, ensures that all GPU threads are working continuously without being idle. At the end of each cycle, all ongoing tasks flow to their next stage, and one task that has progressed through all pipelined modules completes its proof generation and is pushed out of our system. Thus, it has room to introduce a new task at the next cycle. Our system maintains a full workload state until the proof generation is no longer required. At that point, our system completes all remaining tasks in the pipelines before shutting down.

The resource allocation in our system. To maximize GPU core utilization, we manually allocate resources to different modules to keep their throughput consistent. For example, when we apply $10,240$ threads on a GPU V100 card with $5,120$ CUDA cores. Since the amortized execution time ratio of three individual modules is around $35 : 12 : 113$, which is obtained based on tests using the real hardware device, we allocate $2,240 = 35 \times 64$, $768 = 12 \times 64$, and $7,296 = 113 \times 64$ threads to the linear-time encoder, Merkle tree, and sum-check modules, respectively. Inside each module, execution threads are allocated as described in Section 3. For example, when the input is $N = 2^{14}$ blocks to Merkle trees, we should perform $2N \approx N + \frac{N}{2} + \ldots + 1$ hashes, where the number of hashes in each layer is half of the previous layer. Therefore, when the Merkle tree module is allocated with a total of $M = 768$ threads, we will allocate $384 = \frac{M}{2}$ threads to the first layer with N hashes, $192 = \frac{M}{4}$ threads to the second layer with $\frac{N}{2}$ hashes, ..., and $3 = \frac{M}{2^8}$ threads to the 8-th layer with $\frac{N}{2^7}$ hashes. Other 3 threads handle the remaining layers with $\frac{N}{2^7}$ hashes. Each thread processes $\frac{2N}{M}$ hashes, ensuring equal workload per thread. This method is also employed in sum-check and linear-time encoder modules.

Data transfer between host and device memory. Given that our system employs a dynamic loading and storing method to reduce the required GPU device memory, necessitating frequent data transfers between host and device memory. These data transfers include the input to linear-time encoders and sum-check modules, and the output from the intermediate layers of Merkle trees. To reduce the overhead associated with these transfers, we employ multi-stream

technology, which enables the simultaneous execution of data transfers and GPU computations. In our setting, time spent on the computations exceeds that of data transfers, thus ensuring no time is lost waiting for data transfer.

5 Application

In this section, we present the deployment of our pipelined ZKP system in a verifiable machine-learning application. For a more intuitive description, we demonstrate this system in the context of Machine-Learning-as-a-Service (MLaaS), where the service provider (prover) convinces its customers (verifiers) that the prediction results are correctly calculated from a particular machine-learning model, while preserving the model's privacy.

Figure 8 shows an overview of our system for verifiable machine-learning applications. It consists of three components. The first is an interface for the service provider to interact with customers. All public data to both parties, including customer input, prediction results, and zero-knowledge proofs, are transmitted through this interface. Notably, the secret model should be guaranteed not to leak to customers via this interface. The second is the machine-learning engine, which provides prediction services similar to those in MLaaS. Upon receiving customer input, the engine calculates the prediction result using the well-trained model and returns the result to the customers. The engine can be implemented based on machine-learning platforms, such as Pytorch [43]. The third is our ZKP system. It generates zero-knowledge proofs, convincing customers that the prediction results are correctly calculated. Our pipelined system is well-suited to the MLaaS scenario, where the input from the customer is passed to the service provider like a flowing stream.

The system works as follows. In the preprocessing stage, the model parameters are used as input to generate a Merkle tree. The Merkle root is sent to customers, which is used as a commitment, as any change in the model parameters could ultimately change the Merkle root. Moreover, we compile the function for the model inference into a circuit based on the technology proposed in many recent works [5, 13, 35] for verifiable machine learning. The preprocessing stage should only be performed once, so the execution time of the preprocessing stage is not counted into the time to generate zero-knowledge proofs.

Next, the system enters the prediction phase performed by the machine-learning engine. In this stage, customers interact with the service provider as in the traditional MLaaS scenario, where customers send input to the service provider, and the machine-learning engine employs the committed model to calculate the prediction results and return them to the customers. Third, the service provider employs our pipelined ZKP system to generate proofs. Our system collects customer input and the intermediate results from the machine-learning engine and forwards this required data

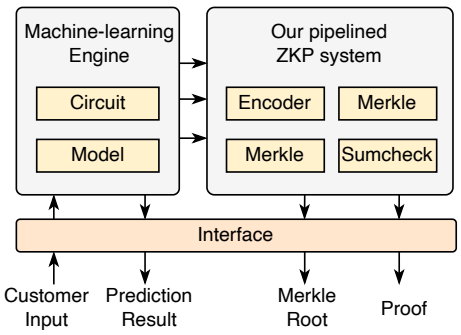


Figure 8. An overview of the verifiable machine-learning application accelerated by our pipelined ZKP system.

into the pipeline in order. The system follows the workload shown in Section 4 to generate zero-knowledge proofs and deliver them to customers through the interface. Finally, customers can verify the prediction results are correctly calculated from the committed model using these proofs.

The security of our system is based on the fact that the ML engine cannot substitute another model to perform inference. On the service provider side, we consider the ML engine and our ZKP system as trusted components because both components are controlled by the service provider. On the customer side, the Merkle root generated during preprocessing ensures that the model cannot be substituted because each inference process includes a ZK proof to prove that this Merkle root is correctly calculated from the committed model.

6 Evaluation

In this section, we present the evaluation of our pipelined GPU-accelerated ZKP system. We give our experimental setup in Section 6.1. Next, we present evaluation results of pipelined modules in Section 6.2. Finally, we give the performance of our overall ZKP system in Section 6.3.

6.1 Experimental Setup

Hardware Setup. We implement our pipelined system using CUDA and evaluate it on the latest Nvidia GH200 platform, which is equipped with an Nvidia GH200 Grace Hopper Superchip (GPU) and an Nvidia Grace chip with 72 Arm cores (CPU). The platform has GPU device memory of 96GB and CPU host memory of 480GB. While our evaluations are mainly performed on the GH200 card, our implementation does not rely on specific features exclusive to new-generation Nvidia GPUs, making it compatible with older Nvidia GPUs, such as Nvidia V100. In addition, our implementation involves data communication between host and device memory, for which a traditional PCIe 3.0/4.0 x16 is sufficient. Since the CPU baselines we compare in this paper are not adapted to the Nvidia Grace Arm chip, we perform these CPU baselines on Amazon EC2 using the c5a.8xlarge instance equipped with 32 vCPU and memory of 64GB.

Table 2. Baseline implementations used for evaluation

Modules	Schemes	Hardware	Languages
Merkle Tree	Orion [61]	CPU	C++
	Simon [51]	GPU	OpenCL
	Ours	GPU	CUDA
Sumcheck	Arkworks [1]	CPU	Rust
	Icicle [28]	GPU	CUDA
	Ours	GPU	CUDA
Encoder	Orion [61]	CPU	C++
	Ours-np	GPU	CUDA
	Ours	GPU	CUDA
ZKPs	Orion&Arkworks	CPU	C++&Rust
	Libsnark [47]	CPU	C++
	Bellperson [14]	GPU	OpenCL
	Ours	GPU	CUDA

Table 3. The throughput of Merkle tree modules

Size	Throughput (trees/ms)			Speedup	
	Orion[61] (CPU)	Simon[51] (GPU)	Ours (GPU)	vs. [61] (CPU)	vs. [51] (GPU)
2^{22}	2.140×10^{-3}	0.845	1.698	793.2×	2.01×
2^{21}	4.290×10^{-3}	1.412	3.356	782.3×	2.38×
2^{20}	8.600×10^{-3}	2.137	6.536	760.0×	3.06×
2^{19}	17.21×10^{-3}	3.003	12.658	735.7×	4.22×
2^{18}	34.45×10^{-3}	3.861	23.810	691.0×	6.17×

Baselines. Table 2 lists the baselines used for comparison in this section. They are all state-of-the-art works on CPU/GPU. Icicle [28] is an industrial product developed by Ingonyama [27], a well-known company focusing on the acceleration of ZKPs. Simon [51] is GPU-accelerated Merkle tree that can be used to scale Bitcoin [40]. Since there is no GPU implementation for the linear-time encoder, we employ "Ours-np" which denotes our linear-time encoder without using pipeline technology. Libsnark and Bellperson [14] are ZKP systems for the protocols that rely on NTT and MSM operations. Both systems are also used as baselines in GZKP [38], whose code has not been open-sourced. We evaluate the combination of Orion [61] and Arkwards [1] to show a CPU-based ZKP implementation that employs the same modules as ours, where Orion provides the linear-time encoder and Merkle tree, and Arkwards provides the sum-check module.

6.2 Evaluating Pipelined ZKP Modules

We benchmark the performance of our pipelined ZKP modules, including the Merkle tree, sum-check protocol, and linear-time encoder. For each module, we compare the state-of-the-art CPU and GPU implementations.

Table 4. The throughput of Sum-check modules

Size	Throughput (proofs/ms)			Speedup	
	Arkworks[1] (CPU)	Icicle[28] (GPU)	Ours (GPU)	vs. [1] (CPU)	vs. [28] (GPU)
2^{22}	0.382×10^{-3}	0.969	1.461	3823×	1.51×
2^{21}	0.773×10^{-3}	1.497	2.884	3729×	1.93×
2^{20}	1.583×10^{-3}	2.160	5.622	3552×	2.60×
2^{19}	3.241×10^{-3}	2.865	10.610	3274×	3.70×
2^{18}	6.497×10^{-3}	3.378	19.753	3040×	5.85×

Table 5. The throughput of Linear-time encoder modules

Size	Throughput (codes/ms)			Speedup	
	Orion[61] (CPU)	Ours-np (GPU)	Ours (GPU)	vs. [61] (CPU)	vs. np (GPU)
2^{22}	0.216×10^{-3}	0.031	0.182	844.7×	5.82×
2^{21}	0.643×10^{-3}	0.061	0.365	567.7×	5.98×
2^{20}	1.699×10^{-3}	0.114	0.726	425.2×	6.33×
2^{19}	3.510×10^{-3}	0.211	1.550	441.8×	7.36×
2^{18}	7.242×10^{-3}	0.328	3.115	430.2×	9.50×

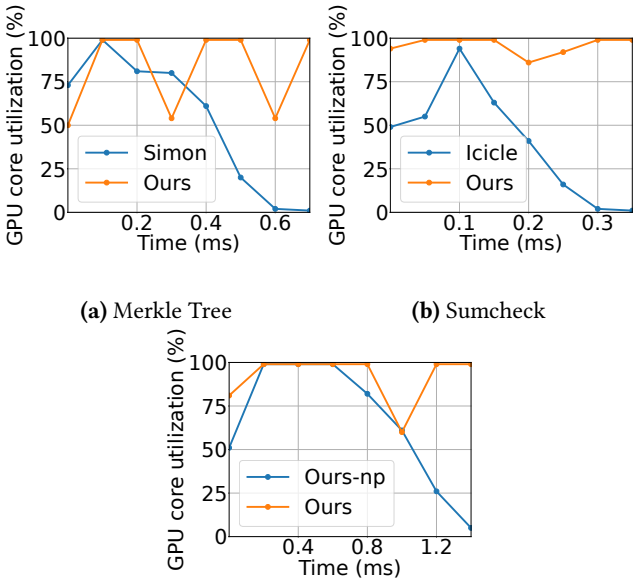
Merkle Tree. Table 3 gives the evaluation results of our pipelined Merkle tree modules. This experiment benchmarks the performance for generating Merkle trees using N 512-bit blocks as input, where N ranges from 2^{18} to 2^{22}. We compare our scheme to two other implementations, Orion [61] and Simon [51], with all using SHA-256 as the hash function.

As shown in Table 3, our pipelined module has high throughput. We achieved a speedup of more than 691.0× (up to 793.2×) compared to Orion's CPU-based implementation [61], and more than 2.01× (up to 6.17×) over Simon's GPU implementation [51]. Notably, our pipelined module maintains nearly linear throughput growth as the size of Merkle trees decreases. This starkly contrasts the Simon implementation, which struggles with the small trees when the tree size is similar to the number of GPU cores. In ZKPs, thousands of relatively small-sized Merkle trees should be generated.

Sum-check. Table 4 gives the evaluation results of our pipelined sum-check modules. This experiment benchmarks the performance for generating sum-check proofs for the multi-linear polynomial $p(x_1, x_2, ..., x_n)$, with the number of variables n ranging from 18 to 22. For convenience, we record the size $N = 2^n$, similar to the Merkle trees. We compare our scheme to two other implementations, Arkwords [1] and Icicle [28], both of which have been used in industry.

Table 6. The latency performance of different ZKP modules

Size	Modules	Schemes	Latency (ms)	Speedup
2^{18}	Merkle	Simon [51]	0.259	0.388×
		Ours	0.668	
	Sumcheck	Icicle [28]	0.296	0.325×
		Ours	0.911	
	Encoder	Ours-np	3.048	0.678×
		Ours	4.494	
2^{20}	Merkle	Simon [51]	0.468	0.161×
		Ours	2.913	
	Sumcheck	Icicle [28]	0.463	0.130×
		Ours	3.557	
	Encoder	Ours-np	8.760	0.396×
		Ours	22.14	



(a) Merkle Tree **(b)** Sumcheck

(c) Linear-time Encoder

Figure 9. GPU core utilization of different ZKP modules evaluated on Nvidia RTX 3090Ti with 10,752 CUDA cores.

As shown in Table 4, our pipelined module demonstrates superior throughput. We achieved a speedup of more than 3040× (up to 3823×) compared to Arkwords's CPU-based implementation [1], and more than 1.51× (up to 5.85×) over Icicle's GPU implementation [28]. Icicle [28] is an industrial product developed by Ingonyama [27], a well-known company focusing on the acceleration of ZKPs. Therefore, achieving such a speedup means that our implementation has great commercial value.

Linear-time encoder. Table 5 gives the evaluation results of our pipelined linear-time encoder modules. This experiment benchmarks the performance for encoding N finite field elements, which can be treated as large integers whose bit-width is set to 256-bit here. We present the evaluation results with N ranging from 2^{18} to 2^{22}. We compare our scheme to Orion [61] as well as "Our-np," which represents our linear-time encoder without using pipeline technology.

As shown in Table 5, our pipelined module demonstrates superior throughput. We achieved a speedup of more than 430.2× (up to 844.7×) compared to Orion's CPU-based implementation [61], and more than 5.82× (up to 9.50×) over our non-pipelined GPU implementation. Similar to our Merkle tree module, our encoder module also maintains nearly linear throughput growth in throughput as the input size decreases. As shown in Figure 1, in the ZKP protocols, the input data is split into multiple parts and forwarded to different encoders. Therefore, each part is relatively small, and our method has advantages in this scenario.

GPU Core Utilization. Figure 9 gives the GPU core utilization of three ZKP modules. Clearly, our pipelined schemes maintain high utilization throughout the computation of each module, whereas other approaches lacking pipelining exhibit a sharp drop in utilization. The decrease is primarily because many threads become idle while waiting for a few threads to complete their tasks, which is consistent with the theoretical thread execution model in Figure 4.

Latency. Table 6 gives the latency metrics for different ZKP modules. Compared to non-pipelined schemes, our approach involves a trade-off between latency and throughput: while the pipelined modules achieve high throughput, they exhibit larger latency. For example, the latency performance of our Merkle tree module is only 0.161× that of the benchmark set by Simon [51]. Therefore, exploring the possibility of improving the throughput without losing too much latency would be an important research direction in the future.

6.3 Evaluating our Pipelined ZKP System

In this section, the evaluation results of our fully pipelined ZKP system are presented. Our system is compared to three other ZKP systems, including Libsnark [47], Bellperson [14], and the combination of Orion [61] and Arkwards [1]. Specifically, Libsnark [47] is a CPU-based implementation for ZKP protocols [3, 15, 22] that relies on NTT and MSM operations. Bellperson [14] is a GPU-accelerated system that supports the same ZKP protocol as Libsnark. In addition, we evaluate the combination of Orion [61] and Arkwards [1] to show a CPU-based ZKP implementation with the same computational modules as ours, where Orion provides linear-time encoders and Merkle trees to commit input data, and Arkwards provides sum-check modules to additionally prove one function is correctly calculated.

Table 7 gives the evaluation results for our pipelined system, where the scale S denotes the number of multiplication

Table 7. The amortized execution time (in millisecond) for each proof generation in CPU/GPU-based systems

S	Platforms	Schemes	MSM	NTT	Proof	Schemes	Merkle	Sumcheck	Encoder	Proof
2^{18}	CPU	Libsnark	18.99×10^3	4.19×10^3	23.19×10^3	Orion&Arkwords	62.5	607.1	143.2	812.7
	GPU	Bellperson	970	267	1299	Ours	0.167	1.782	0.479	2.524
2^{19}	CPU	Libsnark	36.80×10^3	9.05×10^3	45.89×10^3	Orion&Arkwords	124.6	1291.3	295.9	1711.8
	GPU	Bellperson	1318	287	1933	Ours	0.286	2.713	0.833	4.021
2^{20}	CPU	Libsnark	65.32×10^3	20.25×10^3	89.67×10^3	Orion&Arkwords	249.8	2810.8	623.3	3684.0
	GPU	Bellperson	1805	342	2204	Ours	0.535	3.699	1.597	6.161
2^{21}	CPU	Libsnark	138.1×10^3	39.49×10^3	177.8×10^3	Orion&Arkwords	498.3	5856.9	1561.5	7916.7
	GPU	Bellperson	2876	442	3410	Ours	1.004	6.392	3.148	11.189
2^{22}	CPU	Libsnark	281.5×10^3	82.38×10^3	364.1×10^3	Orion&Arkwords	/	/	/	/
	GPU	Bellperson	6795	660	7591	Ours	1.922	10.817	6.270	20.305

gates in the circuit compiled from the function to be proved. In the experiments, we assess the amortized time required for producing each proof for the circuits with S multiplication gates, where S ranges from 2^{18} to 2^{22}. The evaluation results show that our pipelined system is very efficient. Our system achieves a speedup of more than 304.7× (up to 514.8×) compared to GPU-based ZKP implementation, Bellperson [14], and it achieves more than 332.0× (up to 707.5×) over the CPU-based implementation that has the same computational modules as our system. In addition, as shown in Table 10, our system requires less amortized device memory for each proof generation executed in parallel.

A breakdown of throughput improvement. The performance improvement of our work comes from two aspects: one is the adoption of new ZKP protocols, and the other is the deployment of a pipeline mode. We can estimate the breakdown of throughput improvement from new ZKP protocols and our pipeline design in Table 7. For example, when the circuit scale $S = 2^{20}$, the speedup (24.34×) between Orion&Arkwords and Libsnark reflects the contribution of new ZKP protocols. Thus, the extra speedup ($357.7/24.34 = 14.70\times$) between Bellperson and ours reflects the improvement coming from our pipeline design.

Performance across different GPUs. Our system is also compatible with older Nvidia GPUs. Table 8 displays the performance of our system across different GPUs at a circuit scale of $S = 2^{20}$. The results show that our pipelined system maintains great throughput on different GPUs. For instance, on the V100 GPU card, our system achieves a speedup of 259.5× compared to Bellperson. In addition, due to the adoption of new ZKP protocols, our work even achieves lower latency than Bellperson which utilizes old ZKP protocols. The effectiveness of overlapping GPU computation and CPU-GPU communication is shown in Table 9. The results show that the multi-stream technology significantly reduces communication overheads caused by large CPU-GPU data exchanges due to our dynamic data loading approach.

Table 8. The throughput (proofs/second) and latency (seconds) of ZKP systems across different GPUs

GPUs	Schemes	Latency	Speedup	Throu.	Speedup
V100	Bell. [14]	6.579	9.28×	0.152	259.5×
	Ours	0.709		39.44	
A100	Bell. [14]	3.817	10.29×	0.262	305.4×
	Ours	0.371		80.01	
3090Ti	Bell. [14]	2.967	9.36×	0.337	283.2×
	Ours	0.317		95.44	
H100	Bell. [14]	2.703	10.32×	0.370	288.6×
	Ours	0.262		106.8	

Application. In addition, we have deployed our GPU-accelerated system in a verifiable machine-learning application. We compare our implementation with three state-of-the-art works, including zkCNN [35], ZKML [5], and ZENO [13]. They are all CPU-based implementations for verifiable convolutional neural networks. We conducted evaluations using the same VGG-16 [52] network and the CIFER-10 [8] dataset that contains 10 classes of images, and the image size is $32 \times 32 \times 3$. The VGG-16 model we independently trained using Pytorch [43] can achieve an accuracy of 93.93% on the CIFER-10 dataset, outperforming the models utilized in all other ZKP implementations [5, 13, 35].

Table 11 presents the evaluation results for verifiable neural network systems, where our system generates 9.52 proofs per second for the prediction of the VGG-16 model with CIFAR-10 images as input. We achieve a remarkable speedup, being 458× faster than ZENO [13] and 5601× faster than ZKML [5]. To the best of our knowledge, our system is the first that achieves sub-second proof generation in this field.

Table 9. The amortized CPU-GPU communication time and GPU computation time in each cycle of the pipeline.

GPUs	CPU-GPU Connection	Comm. Size	Comm. Time	Comp. Time	Overall (Overlap)
V100	PCIe 3.0 x16	320MB	22.95ms	24.73ms	25.35ms
A100	PCIe 4.0 x16	320MB	10.44ms	12.41ms	12.50ms
3090Ti	PCIe 4.0 x16	320MB	10.50ms	10.42ms	10.56ms
H100	PCIe 5.0 x16	320MB	4.90ms	9.11ms	9.37ms

Table 10. The amortized device memory required for each proof generation executed in parallel

S	2^{18}	2^{19}	2^{20}	2^{21}	2^{22}
Bell. [14]	0.90GB	1.25GB	1.38GB	2.21GB	3.87GB
Ours	0.08GB	0.10GB	0.15GB	0.25GB	0.44GB

7 Related Works

Zero-knowledge Proof. Since zero-knowledge proof (ZKP) [19] was introduced by Goldwasser, Micali, and Rackoff, it has been used as an important cryptographic primitive. Due to its powerful effect, considerable efforts [3, 15, 22, 38, 64] have been directed toward developing ZKPs in both theory and practice. In recent years, a line of efficient ZKP protocols [6, 18, 20, 59, 61, 63] has been proposed. These protocols bypass the expensive operations used in previous ZKP protocols [3, 15, 22], such as number-theoretic transform (NTT) and multi-scalar multiplication (MSM). Instead, they employ cost-effective modules, such as the sum-check protocol [37], Merkle tree [39], and linear-time encoder [24], to generate zero-knowledge proofs. Our work focuses on the acceleration of new ZKP protocols with their cost-effective modules.

In addition, zero-knowledge proofs have been widely used in privacy-critical applications such as blockchain [31, 46, 60], verifiable machine learning [5, 13, 35], and verifiable program analysis [10, 12]. Verifiable machine learning is one of the most promising ZKP applications. When cloud vendors supply paid services using their well-trained models, ZKP schemes allow the customers to verify the outcomes provided by cloud vendors are indeed from the well-trained model, in the face of lazy or malicious vendors. Numerous studies [5, 13, 25, 30, 33, 35, 57] have been dedicated to developing efficient ZKP schemes for machine-learning neural networks. Our pipelined system can further improve their efficiency, achieving sub-second proof generation.

Hardware-accelerated ZKPs. Recently, a great number of works have implemented high-performance zero-knowledge proofs on certain hardware, including GPUs [7, 36, 38], ASICs [64], FPGAs [44, 65] and CPU clusters [34, 58]. These works focus on the acceleration of NTT and MSM operations used in previous ZKP protocols [3, 15, 22]. For example, the systems proposed in the works [36, 38, 58, 64] employ different hardware to accelerate the same ZKP protocol [22] proposed by Groth in 2016. However, recent ZKP protocols [6, 20, 49, 59, 61, 63] increasingly employ cost-effective modules, such as the sum-check protocol, Merkle tree, and linear-time encoder, to generate proofs. The calculation process of these modules is completely different from NTT and MSM. Thus, the previous systems cannot be applied to ZKP protocols dominated by these cost-effective modules.

Simon [51] and Icicle [28] introduced GPU-accelerated Merkle tree and the sum-check protocol, respectively. However, they employ an intuitive parallel algorithm that fails to fully utilize GPU cores, because their employed GPU threads should periodically enter the idle state in the execution procedure to wait for other threads completing their tasks, with details shown in Figure 4a. PipeZK [64] employs a pipeline execution manner to accelerate the NTT and MSM operations on ASIC, while our work employs the pipeline technology to accelerate the sum-check protocol, Merkle tree, and linear-time encoder on GPU.

Table 11. The performance of our pipelined system when it used in the verifiable machine-learning application

Schemes	zkCNN [35]	ZKML [5]	ZENO [13]	Ours
Throughput	0.0113	0.0017	0.0208	9.5220
Latency	88.3s	637s	48.0s	15.2s
Accuracy	90.30%	90.37%	84.19%	93.93%

8 Conclusion

In this work, we propose a fully pipelined GPU-accelerated system for batch generation of zero-knowledge proofs, and our system also supports three pipelined computational modules: the sum-check protocol, Merkle tree, and linear-time encoder. We customize these modules to fit the pipeline execution manner. By adopting recent efficient ZKP protocols and providing a suitable scheme for GPU resource allocation, our system has a considerable speedup over other state-of-the-art implementations, and it shows excellent performance in the verifiable machine-learning application.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful and constructive comments. This work was supported in part by the National Natural Science Foundation of China under Grants 92373205 and 62374146, in part by the National Key Research and Development Program of China No. 2023YFB4404404, in part by the Key Technologies R&D Program of Jiangsu (Prospective and Key Technologies for Industry) under Grant BE2023005-2.

References

[1] arkworks. Linear-time sumcheck protocol for multilinear polynomials and related addends. https://github.com/arkworks-rs/sumcheck.

[2] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the conference on high performance computing networking, storage and analysis*, pages 1–11, 2009.

[3] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P Ward. Aurora: Transparent succinct arguments for r1cs. In *Advances in Cryptology–EUROCRYPT 2019: 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19–23, 2019, Proceedings, Part I 38*, pages 103–128. Springer, 2019.

[4] Vitalik Buterin et al. Ethereum white paper. *GitHub repository*, 1:22–23, 2013.

[5] Bing-Jyue Chen, Suppakit Waiwitlikhit, Ion Stoica, and Daniel Kang. Zkml: An optimizing system for ml inference in zero-knowledge proofs. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 560–574, 2024.

[6] Binyi Chen, Benedikt Bünz, Dan Boneh, and Zhenfei Zhang. Hyperplonk: Plonk with linear-time prover and high-degree custom gates. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 499–530. Springer, 2023.

[7] Yutian Chen, Cong Peng, Yu Dai, Min Luo, and Debiao He. Load-balanced parallel implementation on gpus for multi-scalar multiplication algorithm. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(2):522–544, 2024.

[8] CIFAR-10. A collection of images that are commonly used to train machine learning and computer vision algorithms., 2024. Accessed: April 02, 2024.

[9] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

[10] Santiago Cuéllar, Bill Harris, James Parker, Stuart Pernsteiner, and Eran Tromer. Cheesecloth:{Zero-Knowledge} proofs of real world vulnerabilities. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6525–6540, 2023.

[11] Cysic. Hardware accelerating zero-knowledge proofs. https://cysic.xyz.

[12] Zhiyong Fang, David Darais, Joseph P Near, and Yupeng Zhang. Zero knowledge static program analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2951–2967, 2021.

[13] Boyuan Feng, Zheng Wang, Yuke Wang, Shu Yang, and Yufei Ding. Zeno: A type-based optimization framework for zero knowledge neural network inference. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 450–464, 2024.

[14] filecoin. bellperson is a crate for building zero-knowledge proofs with gpu acceleration. https://github.com/filecoin-project/bellperson.

[15] Ariel Gabizon, Zachary J Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *Cryptology ePrint Archive*, 2019.

[16] Michael Garland. Sparse matrix computations on manycore gpu's. In *Proceedings of the 45th annual design automation conference*, pages 2–6, 2008.

[17] Giza. Actionable ai for decentralized applications. build reliable, scalable and easy to integrate ai solutions for web3. https://www.gizatech.xyz/.

[18] Shafi Goldwasser, Yael Tauman Kalai, and Guy N Rothblum. Delegating computation: interactive proofs for muggles. *Journal of the ACM (JACM)*, 62(4):1–64, 2015.

[19] Shafi Goldwasser, Silvio Micali, and Chales Rackoff. The knowledge complexity of interactive proof-systems. In *Providing sound foundations for cryptography: On the work of shafi goldwasser and silvio micali*, pages 203–225. 2019.

[20] Alexander Golovnev, Jonathan Lee, Srinath Setty, Justin Thaler, and Riad S Wahby. Brakedown: Linear-time and field-agnostic snarks for r1cs. In *Annual International Cryptology Conference*, pages 193–226. Springer, 2023.

[21] Joseph L Greathouse and Mayank Daga. Efficient sparse matrix-vector multiplication on gpus using the csr storage format. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 769–780. IEEE, 2014.

[22] Jens Groth. On the size of pairing-based non-interactive arguments. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 305–326. Springer, 2016.

[23] G Grünwald. On the theory of interpolation. 1942.

[24] Venkatesan Guruswami and Piotr Indyk. Linear time encodable and list decodable codes. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, pages 126–135, 2003.

[25] Meng Hao, Hanxiao Chen, Hongwei Li, Chenkai Weng, Yuan Zhang, Haomiao Yang, and Tianwei Zhang. Scalable zero-knowledge proofs for non-linear functions in machine learning.

[26] Mark Harris. Optimizing parallel reduction in cuda. https://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf.

[27] ingonyama. Hardware accelerators for zero knowledge cryptography. https://www.ingonyama.com/.

[28] Ingonyama. Icicle is a library for zk acceleration using cuda-enabled gpus. https://github.com/ingonyama-zk/icicle.

[29] Zhuoran Ji, Zhiyuan Zhang, Jiming Xu, and Lei Ju. Accelerating multi-scalar multiplication for efficient zero knowledge proofs with multi-gpu systems. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 57–70, 2024.

[30] Daniel Kang, Tatsunori Hashimoto, Ion Stoica, and Yi Sun. Scaling up trustless dnn inference with zero-knowledge proofs. *arXiv preprint arXiv:2210.08674*, 2022.

[31] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE symposium on security and privacy (SP)*, pages 839–858. IEEE, 2016.

[32] Protocol Labs. The state of zero-knowledge proofs: From research to serious business., 2023. Accessed: April 19, 2024.

[33] Seunghwa Lee, Hankyung Ko, Jihye Kim, and Hyunok Oh. vcnn: Verifiable convolutional neural network based on zk-snarks. *IEEE Transactions on Dependable and Secure Computing*, 2024.

[34] Tianyi Liu, Tiancheng Xie, Jiaheng Zhang, Dawn Song, and Yupeng Zhang. Pianist: Scalable zkrollups via fully distributed zero-knowledge proofs. *Cryptology ePrint Archive*, 2023.

[35] Tianyi Liu, Xiang Xie, and Yupeng 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*, pages 2968–2985, 2021.

[36] Tao Lu, Chengkun Wei, Ruijing Yu, Chaochao Chen, Wenjing Fang, Lei Wang, Zeke Wang, and Wenzhi Chen. Cuzk: Accelerating zero-knowledge proof with a faster parallel multi-scalar multiplication algorithm on gpus. *Cryptology ePrint Archive*, 2022.

[37] Carsten Lund, Lance Fortnow, Howard Karloff, and Noam Nisan. Algebraic methods for interactive proof systems. *Journal of the ACM (JACM)*, 39(4):859–868, 1992.

[38] Weiliang Ma, Qian Xiong, Xuanhua Shi, Xiaosong Ma, Hai Jin, Haozhao Kuang, Mingyu Gao, Ye Zhang, Haichen Shen, and Weifang Hu. 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, Volume 2*, pages 340–353, 2023.

[39] Ralph C Merkle. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*, pages 369–378. Springer, 1987.

[40] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[41] Wouter Penard and Tim van Werkhoven. On the secure hash algorithm family. *Cryptography in context*, pages 1–18, 2008.

[42] Polyhedra. Empowering interoperability and computation via zk. bringing interoperability and scalability to web3 with cutting-edge zero-knowledge proof systems. https://www.polyhedra.network/.

[43] Pytorch. A fast, flexible experimentation and efficient production through a user-friendly front-end, distributed training, and ecosystem of tools and libraries., 2024.

[44] Andy Ray, Benjamin Devlin, Fu Yong Quah, and Rahul Yesantharao. Hardcaml msm: A high-performance split cpu-fpga multi-scalar multiplication engine. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 33–39, 2024.

[45] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):1–32, 2013.

[46] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE symposium on security and privacy*, pages 459–474. IEEE, 2014.

[47] scipr lab. This library implements zksnark schemes, which are a cryptographic method for proving/verifying, in zero knowledge, the integrity of computations. https://github.com/scipr-lab/libsnark.

[48] Scroll. Scroll seamlessly extends ethereum's capabilities through zero knowledge tech and evm compatibility. the l2 network built by ethereum devs for ethereum devs. https://scroll.io/.

[49] Srinath Setty. Spartan: Efficient and general-purpose zksnarks without trusted setup. In *Annual International Cryptology Conference*, pages 704–737. Springer, 2020.

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

[51] Simon. Gpu accelerated high-speed merkle tree computation for bitcoin. https://github.com/shilch/fastmerkle.

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

[53] Swaminathan Sivasubramanian. Amazon dynamodb: a seamlessly scalable non-relational database service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 729–730, 2012.

[54] Daniel A Spielman. Linear-time encodable and decodable error-correcting codes. In *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, pages 388–397, 1995.

[55] Victor Vu, Srinath Setty, Andrew J Blumberg, and Michael Walfish. A hybrid architecture for interactive verifiable computation. In *2013 IEEE Symposium on Security and Privacy*, pages 223–237. IEEE, 2013.

[56] Riad S Wahby, Ioanna Tzialla, Abhi Shelat, Justin Thaler, and Michael Walfish. Doubly-efficient zksnarks without trusted setup. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 926–943. IEEE, 2018.

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

[58] Howard Wu, Wenting Zheng, Alessandro Chiesa, Raluca Ada Popa, and Ion Stoica. {DIZK}: A distributed zero knowledge proof system. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 675–692, 2018.

[59] Tiacheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. Libra: Succinct zero-knowledge proofs with optimal prover computation. In *Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part III 39*, pages 733–764. Springer, 2019.

[60] Tiancheng Xie, Jiaheng Zhang, Zerui Cheng, Fan Zhang, Yupeng Zhang, Yongzheng Jia, Dan Boneh, and Dawn Song. zkbridge: Trustless cross-chain bridges made practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 3003–3017, 2022.

[61] Tiancheng Xie, Yupeng Zhang, and Dawn Song. Orion: Zero knowledge proof with linear prover time. In *Annual International Cryptology Conference*, pages 299–328. Springer, 2022.

[62] Jiaheng Zhang, Tianyi Liu, Weijie Wang, Yinuo Zhang, Dawn Song, Xiang Xie, and Yupeng Zhang. Doubly efficient interactive proofs for general arithmetic circuits with linear prover time. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 159–177, 2021.

[63] Jiaheng Zhang, Tiancheng Xie, Yupeng Zhang, and Dawn Song. Transparent polynomial delegation and its applications to zero knowledge proof. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 859–876. IEEE, 2020.

[64] Ye Zhang, Shuo Wang, Xian Zhang, Jiangbin Dong, Xingzhong Mao, Fan Long, Cong Wang, Dong Zhou, Mingyu Gao, and Guangyu Sun. Pipezk: Accelerating zero-knowledge proof with a pipelined architecture. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 416–428. IEEE, 2021.

[65] Baoze Zhao, Wenjin Huang, Tianrui Li, and Yihua Huang. Bstmsm: A high-performance fpga-based multi-scalar multiplication hardware accelerator. In *2023 International Conference on Field Programmable Technology (ICFPT)*, pages 35–43. IEEE, 2023.