GZKP: A GPU Accelerated Zero-Knowledge Proof System

Weiliang Ma*‡
Huazhong University of Science and Technology
Wuhan, Hubei, China
weiliangma@hust.edu.cn

Qian Xiong*‡
Huazhong University of Science and Technology
Wuhan, Hubei, China
qianxiong@hust.edu.cn

Xuanhua Shi†‡
Huazhong University of Science and Technology
Wuhan, Hubei, China
xhshi@hust.edu.cn

Xiaosong Ma
Hamad Bin Khalifa University
Doha, Qatar
xma@hbku.edu.qa

Hai Jin‡
Huazhong University of Science and Technology
Wuhan, Hubei, China
hjin@hust.edu.cn

Haozhao Kuang‡
Huazhong University of Science and Technology
Wuhan, Hubei, China
haozhaokuang@hust.edu.cn

Mingyu Gao
Tsinghua University
Beijing, China
gaomy@tsinghua.edu.cn

Ye Zhang
Scroll Foundation
Seychelles
ye@scroll.io

Haichen Shen
Scroll Foundation
Seychelles
haichen@scroll.io

Weifang Hu‡
Huazhong University of Science and Technology
Wuhan, Hubei, China
huwf@hust.edu.cn

ABSTRACT

Zero-knowledge proof (ZKP) is a cryptographic protocol that allows one party to prove the correctness of a statement to another party without revealing any information beyond the correctness of the statement itself. It guarantees computation integrity and confidentiality, and is therefore increasingly adopted in industry for a variety of privacy-preserving applications, such as verifiable outsource computing and digital currency.

A significant obstacle in using ZKP for online applications is the performance overhead of its proof generation. We develop GZKP, a GPU accelerated zero-knowledge proof system that supports different levels of security requirements and brings significant speedup toward making ZKP truly usable. For polynomial computation over a large finite field, GZKP promotes a cache-friendly memory access pattern while eliminating the costly external shuffle in existing solutions. For multi-scalar multiplication, GZKP adopts a new parallelization strategy, which aggressively combines integer elliptic curve point operations and exploits fine-grained task parallelism with load balancing for sparse integer distribution. GZKP outperforms the state-of-the-art ZKP systems by an order of magnitude, achieving up to 48.1× and 17.6× speedup with standard cryptographic benchmarks and a real-world application workload, respectively.

CCS CONCEPTS

• **Computer systems organization** → **Parallel architectures**; *Data flow architectures*; • **Security and privacy** → *Privacy-preserving protocols*.

KEYWORDS

zero-knowledge proof; GPU acceleration

ACM Reference Format:
Weiliang Ma, Qian Xiong, Xuanhua Shi, Xiaosong Ma, Hai Jin, Haozhao Kuang, Mingyu Gao, Ye Zhang, Haichen Shen, and Weifang Hu. 2023. 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 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3575693.3575711

*Both authors contributed equally to this research.
†Corresponding author.
‡Weiliang Ma, Qian Xiong, Xuanhua Shi, Hai Jin, Haozhao Kuang, and Weifang Hu are with National Engineering Research Center for Big Data Technology and System, Service Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China.

1 INTRODUCTION

Zero-knowledge proof (ZKP) is a cryptographic primitive through which one party can convince other parties the validity of a statement without disclosing any information beyond the correctness of the statement itself. For example, the owner of the machine learning model can declare that the model has reached a certain accuracy on a public data set, and then use the ZKP primitive to guarantee the validity of the declaration to others without disclosing any secret information (*e.g.*, parameters) about the model. ZKP could provide security guarantees about the correctness and consistency of machine learning predictions in real products [61].

As one of the most powerful cryptographic tools, ZKP has the potential to be used in many privacy-preserving applications, such as *verifiable outsourcing* [62], *verifiable machine learning* [61], *electronic voting* [64], and *online auction* [32]. As an up-and-coming field approaching commercial applications with increasingly powerful computer hardware, ZKP witnesses the establishment of a flurry of startups [17, 19, 21, 44, 65] in the past six years. Most of them leverage ZKP for the scalable and user-manageable privacy of cryptocurrency and Web3.

Protocols for ZKP have been studied for more than 30 years since its first introduction [37], and recent advances in cryptography have brought ZKP closer to practical use. In particular, *zero-knowledge succinct non-interactive arguments of knowledge* (zkSNARK) [40] is one of the most promising ZKP protocols. It can generate a succinct proof for any program (i.e., a few hundred bytes), and support fast verification of the proof. Due to these properties, zkSNARK has been deployed in real-world systems of *blockchains* and *cryptocurrencies* [7, 15, 17, 22, 44].

Despite such progress, ZKP is still notoriously hard to be widely adopted in industry due to the particularly large overhead of generating proofs. For example, a typical zkSNARK proof generation process in Zcash [22] involves billions of large integer addition and modular multiplication [52] on a large finite field where the bit-width is 381-bit. On mainstream servers today, each such modular multiplication would take 230 ns, and each large integer addition costs 43 ns. With billions of operations for one anonymous transaction [22], one such proof generation process would need over half a minute to finish. Such a long latency makes it impractical to use in scenarios like real-time payment and decentralized trading transactions with cryptocurrency, despite its great benefits in privacy protection.

Most of the existing ZKP systems are CPU-based [3, 23, 34, 48, 50, 54] and are forced to use clusters for larger-scale ZKP applications [60]. Recently emerged GPU-based ZKP systems [16, 18] take advantage of the massive hardware parallelism available on modern GPUs to accelerate key ZKP operations. However, existing GPU-based solutions fall short in maximizing ZKP processing efficiency in several aspects. They rely on excessive data shuffling in the GPU global memory to ensure contiguous memory accesses, whose cost quickly grows with increasing bit-width; they have not exploited the opportunities of aggressively eliminating redundant computations to improve performance; they cannot effectively handle the significant load imbalance caused by the sparse data distribution and the large bit-width integer used in current real-world ZKP applications; and their underlying mathematics library to support large finite field arithmetics is not optimized.

In this paper, we propose GZKP, a fast GPU accelerated zero-knowledge proof system that significantly reduces proof generation time in ZKP. GZKP is designed to tackle the challenge of ZKP on large finite fields and exploit the massive hardware parallelism offered by current GPU systems. GZKP is a general-purpose system that can serve a range of ZKP applications (such as Zcash [22] and Mina [19]) and hence can be easily integrated into existing ZKP libraries as a high-performance back-end.

GZKP alleviates the performance bottlenecks in the parallel execution of existing ZKP implementations. It avoids the use of frequent and expensive data shuffling, exploits opportunities to merge redundant operations, adopts customized scheduling schemes to improve load balance, and also integrates a highly optimized library for large integer arithmetics. More specifically, for the number theoretic transform (NTT) stage in proof generation (§ 3), GZKP promotes a cache-friendly memory access pattern, which allows threads to perform on-the-fly and internal data shuffling when transferring data between the global and shared memories. The global memory layout keeps stable. We coalesce the accesses from neighbor threads by visiting the global memory with a coarse-grained block-style pattern to ensure contiguous memory accesses. For the next stage of multi-scalar multiplication (MSM) computation (§ 4), we propose a novel point merging algorithm to effectively consolidate computation and alleviate redundant expensive operations. The algorithm leverages space-efficient data preprocessing, and applies a new parallelization strategy to reduce the serial execution bottleneck. GZKP also utilizes fine-grained task mapping to ensure load balance.

We implement GZKP using CUDA and evaluate it on Nvidia V100 GPUs. We compare GZKP with multiple state-of-the-art ZKP systems, both CPU-based (libsnark [50] and bellman [23]) and GPU-based (MINA [18] and bellperson [16]). Our results (§ 5) show that, compared with the state-of-the-art GPU-based ZKP system, GZKP achieves on average 13.2× and up to 17.6× speedup for real-world application workloads, and on average 33.6× and up to 48.1× speedup for standard cryptographic microbenchmarks. If we measure the two key stages separately, GZKP achieves on average 5.8× and up to 10.3× speedup in the NTT module, and on average 9.1× and up to 17.9× speedup in the MSM module.

In summary, this paper makes the following major technical contributions in GZKP:

- An NTT module that performs memory accesses with better cache efficiency, by replacing the increasingly expensive global-memory shuffle operation with flexible scheduling and internal shuffling;
- An MSM module that uses a new parallelization strategy to aggressively consolidate redundant integer elliptic curve point operations, with space-efficient data preprocessing and fine-grained task load balancing; and
- A GPU finite field library that supports arithmetic operations on large integers (*e.g.*, 256-bit ∼ 753-bit) by leveraging both integer and floating-point processing units.

This work focuses on emerging privacy-preserving applications based on zero-knowledge proof. Our proposed system, GZKP, brings

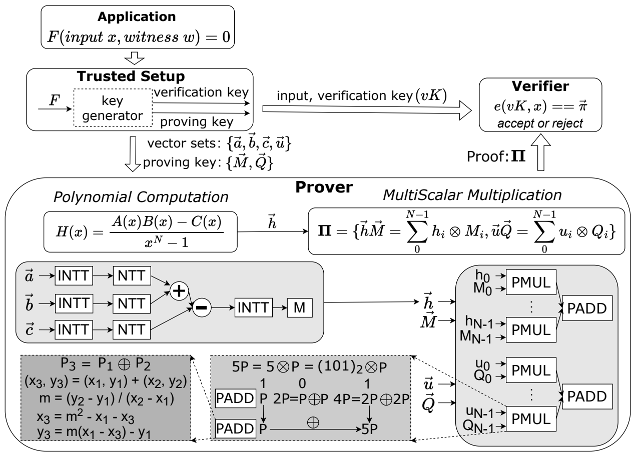
Application
$F(input\ x, witness\ w) = 0$
Trusted Setup
F
key generator
verification key
proving key
input, verification key (vK)
Verifier
$e(vK, x) == \vec{\pi}$
accept or reject
vector sets: $\{\vec{a}, \vec{b}, \vec{c}, \vec{u}\}$
proving key: $\{\vec{M}, \vec{Q}\}$
Proof: Π
Prover
Polynomial Computation
MultiScalar Multiplication
$H(x) = \frac{A(x)B(x) - C(x)}{x^N - 1}$
$\vec{h}$
$\Pi = \{\vec{h}\vec{M} = \sum_0^{N-1} h_i \otimes M_i, \vec{u}\vec{Q} = \sum_0^{N-1} u_i \otimes Q_i\}$
INTT
NTT
M
PMUL
PADD
$P_3 = P_1 \oplus P_2$
$(x_3, y_3) = (x_1, y_1) + (x_2, y_2)$
$m = (y_2 - y_1) / (x_2 - x_1)$
$x_3 = m^2 - x_1 - x_3$
$y_3 = m(x_1 - x_3) - y_1$
$5P = 5 \otimes P = (101)_2 \otimes P$
$2P = P \oplus P$
$4P = 2P \oplus 2P$
$5P$

Figure 1: Typical zkSNARK protocol workflow.

decisive improvement to the market, supporting a variety of ZKP systems used in the industry, such as Marlin [11], Plonk [31], and Sonic [51]. In addition, these two modules that we investigated (NTT and MSM) has applications in a wider scope of encryption algorithms: NTT is a key building block in homomorphic encryption algorithms [35] used by federated learning, while MSM is used in many pair-based proof systems [8, 40, 54]. Techniques proposed in this work thus contribute to this growing junction of computer systems and security/privacy, by significantly enhancing the effectiveness of GPU resource utilization for crucial components of private computation.

2 BACKGROUND

2.1 The zkSNARK Protocol

zkSNARK is one of the most prominent ZKP protocols. It consists of two parties: a *prover* and a *verifier*. The *prover* can convince the *verifier* that, "for given function F and input x, I know a secret witness w such that $F(x, w) = 0$," by generating a zkSNARK proof.

zkSNARK possesses three desirable properties: (1) *succinctness*, *i.e.*, small proof sizes (<1 KB) and fast verification (<10 ms) regardless of F's complexity, (2) *non-interactive*, *i.e.*, a single message from the *prover* to the *verifier*, and (3) *zero knowledge*, *i.e.*, a proof revealing nothing about the actual value of w while still able to pass the verification.

With these properties, recently zkSNARK has been widely adopted by the blockchain community [15, 17, 19, 22, 44]. The smart contract stored on a blockchain is a program performing the function F in Figure 1. Traditional blockchain applications require each node in a chain to perform the same smart contract for the chain's state updates. For example, for an online transaction, the nodes in the chain verify its validity by checking whether both parties' account information (secret witness w) meets the transaction condition. It is worth noting that a smart contract involving private datasets may reveal user privacy because anyone running their blockchain node can see and download all data stored on the chain. With the number of such contracts growing, all nodes will incur huge overhead and long latency, significantly limiting the throughput of latency-sensitive applications (*e.g.*, decentralized transactions).

With ZKP, users could generate succinct proofs for programs and then put the proofs on the chain: each node only needs to verify by checking the integrity of the proofs, a task much lighter than executing the smart contract. Also, these zero-knowledge proofs hide sensitive information involved in the smart contract computation. As a result, ZKP can greatly enhance *both* blockchain scalability and privacy protection.

As shown in Figure 1, various random parameters for the whole ZKP system are generated in a one-time setup phase including the proving key and the verification key for the *prover* and *verifier*. And with prover's secret witness w and public input x, the system outputs two kinds of data which will later be used in the *prover* and *verifier*:

- **Vector sets**: $\{\vec{a}, \vec{b}, \vec{c}, \vec{u}\}$. Each vector contains N large integers on a finite field F_r where r is a large prime. The bit-width of the large integer field here typically ranges from 256 to 753. The large integer r can be composed as $\sum_{i=0}^{m} r_i D^i$, where $D = 2^{64}$ denotes the base, so that the vector r_i can be stored in word-size integers. The dimension N could be up to billions and is typically a power of 2.[1]
- **The proving key and verification key**: The proving key consists of multiple point vectors $\{\vec{M}, \vec{Q}\}$, each containing N points on a pre-determined elliptic curve, expressed in the form of $\{(x, y) \in F_q \times F_q \mid y^2 = x^3 + ax + b\}$. F_q is another finite field where q is a large prime. Similar to F_r, the bit-width of F_q is also between 256-bit and 753-bit. The verification key is a short point vector with a few elliptic curve points. The basic operations of elliptic curves are *point-addition* (PADD)[2] and *point-multiplication* (PMUL) [41]. We use $\oplus$ and $\otimes$ to denote the PADD and PMUL operations in elliptic curves, respectively. As demonstrated in Figure 1, PADD is used for adding up two points on the elliptic curve and it needs to do a set of arithmetic operations over the same finite field. PMUL is defined by adding a point P to itself s times (through PADD), where s is an element over F_r. The figure illustrates how it can be split into a series of PADD in order of the integer's bit sequence by using the binary representation of the integer.

Then the *prover* uses the vectors sets and the proving key to generate a proof. The *prover* performs two-stage calculations successively, *polynomial computation* (POLY) and *multi-scalar multiplication* (MSM), to generate the proof.

In the POLY stage, the *prover* takes the vectors $\vec{a}, \vec{b}$, and $\vec{c}$ as inputs, and outputs the coefficients of polynomial $H(x)$, where $H(x) = (A(x)B(x) - C(x))/(x^N - 1)$. As shown in Figure 1, the state-of-the-art implementation of this stage consists of a series of NTT and its inverse (INTT) operations [16, 50], with more details given in Section 3.

In the MSM stage, the *prover* takes the output $\vec{h}$ of POLY, the scalar vector $\vec{u}$, and the proving key as inputs and computes multiple multi-scalar multiplications over the specified elliptic curves. Each

[1]General cases with arbitrary N values can use the power-of-2 flow as building blocks [12].
[2]Here, we use PADD to refer to elliptic curve point addition operations, including point doubling operations

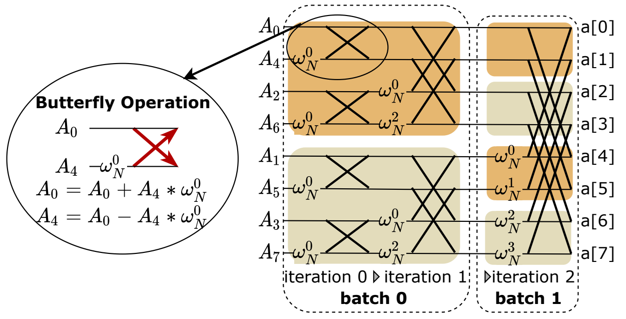

Figure 2: The computation process and strided data access pattern of 8-NTT.

MSM task can be expressed as $\vec{s} \cdot \vec{P} = \sum_{i=0}^{N-1}(s_i \otimes P_i)$, where $\vec{s}$ is a scalar vector ($\vec{h}$ or $\vec{u}$) and $\vec{P}$ is a point vector from the proving key ($\vec{M}$ or $\vec{Q}$). As shown in Figure 1, each $s_i \otimes P_i$ is calculated through PMUL and summed using PADD. These two inner-product results are sent out by the *prover* as its proof, to be checked by the *verifier*.

At last, the *verifier* can use the verifying key to quickly determine the validity of the proof. It usually takes a few milliseconds in zkSNARK.

2.2 NTT Computation

For a polynomial $A(x) = \sum_{i=0}^{N-1} A_i x^i$, with the coefficients $A_i \in F_r$, an NTT computes the vector $\vec{a} = \{A(1), A(\omega_N), \ldots, A(\omega_N^{N-1})\}$, where ω_N is the N-th root of unity in F_r. The state-of-the-art algorithm for NTT is through the Cooley-Tukey algorithm [14], it can reduce the complexity from $O(N^2)$ to $O(NlogN)$.

We denote NTT computation with scale N as N-NTT, which contains $n = \log N$ iterations with the Cooley-Tukey algorithm. Figure 2 illustrates the process of 8-NTT. Each iteration contains $N/2$ *butterfly operations*, the basic NTT operations. The zoom-in bubble in the figure gives the definition of the butterfly operation, which reads two inputs, performs a series of modular multiplications and additions (both of large integers in this case), and stores the result in place. After the last iteration, the original $\vec{A}$ vector (coefficients of the polynomial $A(x)$) contains the result vector $\vec{a}$.

Such NTT computation can be parallelized, illustrated in Figure 2 with colored backgrounds, with butterfly operations in consecutive iterations divided into multiple independent groups. The common practice [10, 16, 26, 36, 38, 46] is to divide the iterations into *batches* such that each group of independent butterfly operations (within one colored block in the figure) works on a sub-vector that can be accommodated in the shared memory. Each of such independent group is naturally mapped to a GPU block for parallel computation. As shown in the figure, when the input vector A is stored in an array in the global memory, the distance between the two inputs of a butterfly operation increases exponentially as the iterative computation progresses.

Such a strided access pattern with a growing stride size brings a large overhead. When hosted in the global memory for a larger 2^{24}-NTT with input bit-width of 256, these non-contiguous memory accesses occupy 13% of the overall execution time in batch 1 (batch 0 has contiguous accesses) and 53% in batch 2, despite performing the same amount of computation (confirming prior findings [16]). This is the reason why current implementations, including ours, partition the iterations into batches so that within each batch, the gradually enlarging strides can be accommodated within the small and fast shared memory.

Previous solutions [10, 16, 38] perform costly reordering of the A vector array at the beginning of each batch, so that the next iteration can use purely contiguous accesses. This step is called the shuffle stage. However, with the large bit-width used in ZKP, the many batches require more such shuffle stages, each costing 42% ~ 81% of the per-batch execution time.

2.3 MSM Computation

MSM performs high-dimensional vector inner-product, entailing many expensive PMUL operations on an elliptic curve. The cost of vector inner-product is proportional to the vector size N, which is up to several billion. Meanwhile, the PMUL operation on the elliptic curve is time-consuming because it involves a large number of modular multiplication and addition operations over the large finite field F_q as shown at the bottom of Figure 1. Therefore, MSM is the most computation-intensive part in ZKP proof generation. Existing CPU-based zkSNARK implementations spend over 70% of their total execution time on the MSM stage [50].

As mentioned earlier, exiting work [16, 18, 63] uses optimizations such as the *Pippenger* [55] and *Straus* [58] algorithms to reduce the MSM complexity. The core idea behind is to reduce the PMUL

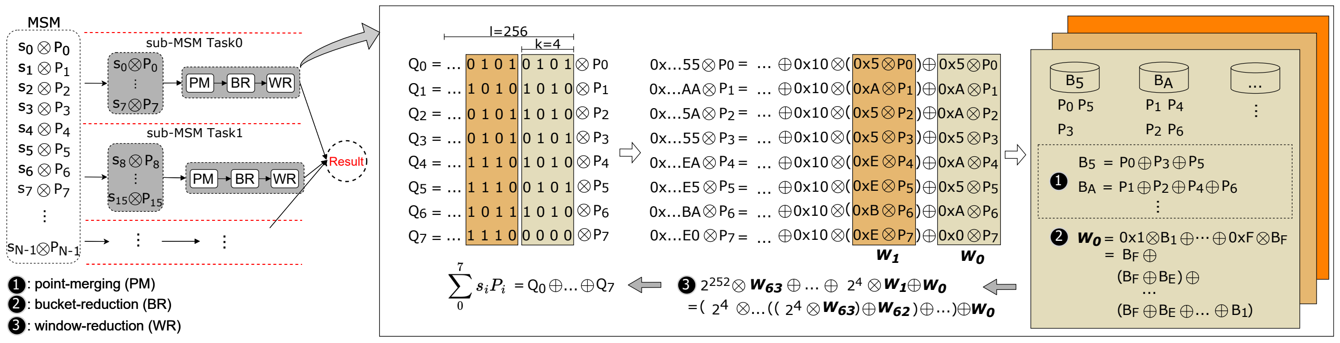
MSM
$s_0 \otimes P_0$
$s_1 \otimes P_1$
$s_7 \otimes P_7$
$s_{N-1} \otimes P_{N-1}$
sub-MSM Task0
sub-MSM Task1
PM
BR
WR
Result
❶: point-merging (PM)
❷: bucket-reduction (BR)
❸: window-reduction (WR)
l=256
k=4
$Q_0 = \ldots 0101\,0101 \otimes P_0$
$0x\ldots55 \otimes P_0 = \ldots \oplus 0x10 \otimes (0x5 \otimes P_0) \oplus 0x5 \otimes P_0$
W_1
W_0
$B_5 = P_0 \oplus P_3 \oplus P_5$
$B_A = P_1 \oplus P_2 \oplus P_4 \oplus P_6$
$W_0 = 0x1 \otimes B_1 \oplus \cdots \oplus 0xF \otimes B_F = B_F \oplus (B_F \oplus B_E) \oplus \cdots (B_F \oplus B_E \oplus \ldots \oplus B_1)$
$2^{252} \otimes W_{63} \oplus \ldots \oplus 2^4 \otimes W_1 \oplus W_0 = (2^4 \otimes \ldots ((2^4 \otimes W_{63}) \oplus W_{62}) \oplus \ldots) \oplus W_0$
$\sum_0^7 s_i P_i = Q_0 \oplus \ldots \oplus Q_7$

Figure 3: An example MSM computation process using Pippenger algorithm.

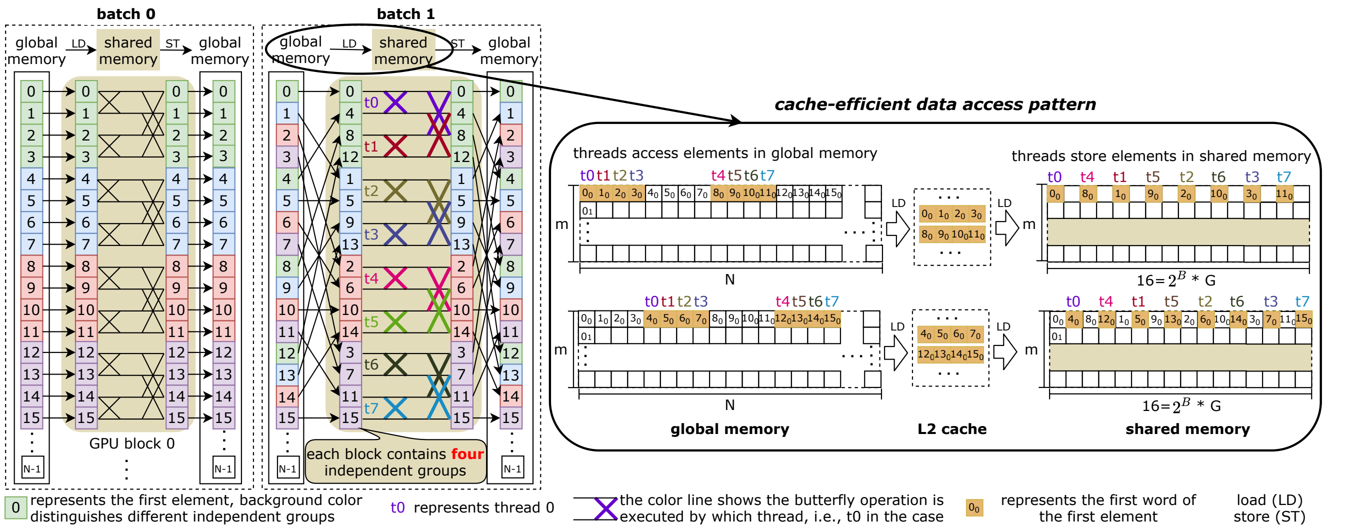
batch 0
batch 1
global memory
LD
shared memory
ST
GPU block 0
each block contains four independent groups
cache-efficient data access pattern
threads access elements in global memory
threads store elements in shared memory
global memory
L2 cache
shared memory
16=2^B * G
represents the first element, background color distinguishes different independent groups
t0 represents thread 0
the color line shows the butterfly operation is executed by which thread, i.e., t0 in the case
represents the first word of the first element
load (LD)
store (ST)

Figure 4: Iteration partitioning at $B = 2$. Here a GPU block contains $G = 4$ independent groups, each working on 2^B elements (in the same color). A total of $T = 2^B * G/2 = 8$ threads are required in each block. The zoom-in illustration on the right side shows the per-thread memory access pattern When loading data from the global memory to the shared memory.

operations with the distributive law. Figure 3 uses the Pippenger algorithm to illustrate this process. It represents each l-bit scalar s under the 2^k base, where k is the pre-specified *window size*. In the right part of Figure 3 ($N = 8$), k is 4 and l is 256, resulting in 64 windows. The PMUL operations within each window are then grouped by the scalar component into *buckets*. For example, in window w_0, different elements of $\vec{s}$ produce common components of `0xA` and `0x5`, producing a bucket 5 containing the points P_0, P_3, and P_5, and another bucket A. With the distributive law, one can therefore save computation by first summing up all the points in bucket j to get B_j within a window (*point-merging*), then calculating $\sum_{j=0}^{2^k-1} j \otimes B_j$ for window t to get W_t (*bucket-reduction*), and finally computing $\sum_{t=0}^{\lceil l/k \rceil -1} 2^{t*k} \otimes W_i$, where 2^{t*k} is a weight, to combine the results across windows (*window-reduction*).

The scale of MSM in ZKP systems is often over millions. Due to the large N, previous work explores data parallelism by decomposing the computation horizontally into multiple *sub-MSM* tasks, as shown in Figure 3 left. Each sub-MSM can be naturally assigned to a GPU block, where algorithms like Pippenger are further adopted to reduce the complexity, with concurrent processing of different windows by different threads, as shown in the figure. However, existing solutions have not explored the fact that point-merging tasks in the same window for different sub-MSMs can also be merged to reduce the subsequent reduction tasks, which is among our contributions in GZKP (Section 4).

3 GZKP DESIGN: THE POLY STAGE

In this section, we describe GZKP's new design for more efficient GPU-based large-scale NTT computation over large finite fields.

Cache-Efficient Global Memory Access. For N-NTT in ZKP, the total size of its input array (vectors $\{\vec{a}, \vec{b}, \vec{c}\}$ shown in Figure 1) varies from tens of MBs to GBs, with each array element in m machine-word-size (64-bit) integers. The input arrays are stored in the GPU global memory with a column-major layout [10] (*i.e.*, storing the first words of all N integers contiguously, then all the second words, until m). In our discussion next, we use the column index to identify an array element. For example, element 0 represents the m integers in the first column of this layout.

As said in Section 2.2, a group of independent butterfly operations in a batch are mapped to a GPU block for parallel computation. The block works on a subset of array elements, which are staged to the shared memory, also in a column-major layout. The GPU shared memory is the memory local to each streaming multiprocessor (SM), much faster yet smaller than the global memory. For example, each of the 80 SMs of the NVIDIA V100 model has a 48 KB shared memory, accessible to up to 1024 threads of the block running on the SM, at the word granularity. With one warp (usually 32 threads) accessing global memory simultaneously, it has been found that for vectors of large integers, the fine-grained interleaving among data accessed by different threads with such a column-major layout could offer significantly better performance than storing each integer contiguously [10]. This is confirmed by our own experiments, which show a 2× advantage of the former with a 753-bit integer size in global memory accessing.

With each warp accessing a subset of vector data for NTT, data transfer happens at the beginning and the end of each batch, to put relevant data into and evict back results from the shared memory. A major reason for existing GPU-based NTT designs to perform expensive data shuffles between batches [10, 16, 38] is to ensure that each warp has contiguous access to at least one GPU L2 cache line (32 B with V100), to fully utilize data brought into the L2 cache. However, the cost of such shuffle operations quickly increases when we get to the later iterations, as mentioned in Section 2.2.

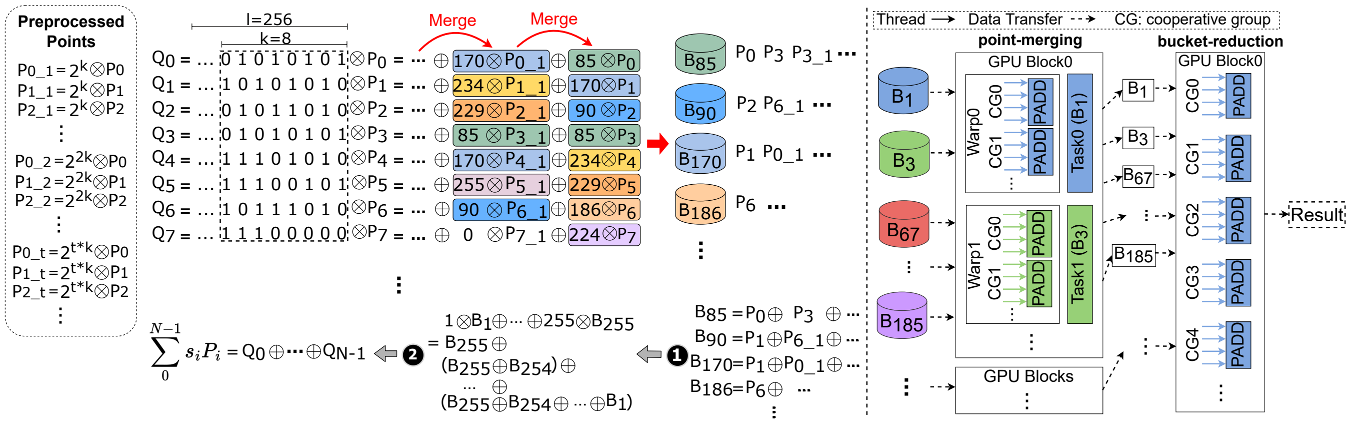

Figure 5: The bucket-based task-partitioning and task-mapping for the MSM computation

Considering the large scale of NTT operations in ZKP applications, GZKP follows a different, *shuffle-less* approach, keeping the vector in the original order in the global memory across all batches of iterations (Figure 4). It then has to address the conflict between (1) the fine-grained access (byte-level) of each thread due to the column-major layout, for better global memory access speed, and (2) the growing distance of array elements accessed by each GPU block, without data shuffling between batches. Otherwise, both issues would significantly lower the L2 cache line utilization.

GZKP addresses these challenges with a cache-friendly task partitioning scheme. Suppose we have B iterations within each batch, where GZKP divides the butterfly operations into $N/2^B$ independent groups, each of which works on 2^B elements. Figure 4 illustrates this with $B = 2$, where the 16 elements' butterfly computation form 4 such groups (grouped by background colors), each working on a disjoint subset of 4 elements. With existing GPU-based NTT designs, B is maximized (under the shared memory capacity constraint) to make such independent groups large. Each group is mapped to one GPU block, to allow large chunks of work to proceed in parallel.

Considering the aforementioned L2 cache efficiency problem, GZKP adopts a different design. Rather than having each GPU block work on one large group, we assign it with multiple smaller ones. In Figure 4, for example, the four independent groups are all assigned to GPU block 0. This allows GZKP threads to perform internal shuffling when transferring data between the global and shared memories: neighboring threads will coalesce their accesses by visiting the global memory with a coarse-grained block-style data distribution, then creating the fine-grained interleaving at the shared memory.

We illustrate more using the examples in Figure 4. Batch 0 on the left have contiguous access within each independent group, so the threads simply access the data needed, where neighboring threads will visit neighboring data. This stops being the case with batch 1 (with GZKP discarding data shuffling in the global memory), where vector elements have a stride of 4: the first group will be working on elements 0, 4, 8, and 12. Nevertheless, if we could schedule G independent groups together where each group processes 2^B elements, the overall required data nicely form 2^B length-G contiguous chunks in the global memory. Essentially, we transform these 2^B length-G chunks in the global memory to G length-2^B chunks in the shared memory. As long as G is sufficiently large, *e.g.*, at 4 or higher, accessing these chunks from the global memory ensures effective utilization of each L2 cache line. We use T threads to fetch these chunks in parallel in multiple rounds. Figure 4 example shows $G = 4$ and $B = 2$, as on the right side. In each round, the threads access contiguous elements from the global memory that can be well coalesced. They then write the data to the appropriate location in the shared memory, producing the desired strided interleaving pattern, promoting efficient access within each independent group in their NTT computation. As to be seen later in our experiments, such a design improves the NTT performance by up to 2.1×. When the current batch finishes, the reverse process is used to write data back to the global memory, with the original data order.

The internal shuffle operation above ensures contiguous memory access at the global memory side for L2 cache efficiency. By scattering them to the desired locations as dictated by the current batch's independent groups order (such as "0 4 8 12" for the first group), subsequent iterations of NTT computation in this batch could proceed with one thread's data placed close together. Such sequential and reverse-order interleaving of contiguous data segments further reduces the inter-bank access conflicts on the shared memory [20].

4 GZKP DESIGN: THE MSM STAGE

This section focuses on the MSM stage, which occupies most of the ZKP proof generation time. We present two major optimizations: (1) further computation consolidation beyond those exploited by existing approaches, and (2) load balancing techniques targeting scalar vectors containing large integer elements. It is worth noting that our MSM optimizations presented in this section is agnostic to elliptic curve types, and apply to both the two common types (G1 and G2) used in current ZKP systems [23, 50] despite their different PADD operation costs.

4.1 Computation Consolidation

As mentioned in Section 2.3, previous systems [16, 18] exploit window-based parallelism by processing the windows in parallel, as shown in Figure 3. Windows are partitioned horizontally into sub-MSMs. Each sub-MSM is mapped to a GPU block, where different windows within the same sub-MSM are processed by different threads. Using Pippenger algorithm in each sub-MSM, the points are first summed up in the buckets (point-merging), multiplied with the corresponding bucket scalars and summed (bucket-reduction), and finally accumulated across windows (window-reduction).

Unlike prior systems, GZKP exploits bucket-based task partitioning and fine-grained task-mapping to ensure load balance, as illustrated in Figure 5. While it still adopts a Pippenger-like algorithm (shown on the left side of the figure), GZKP pursues parallelization in a different manner by leveraging previously unexploited opportunities to further reduce the PMUL operations in MSM. The large bit-width of current ZKP applications enables a large number of windows available for concurrent processing. GZKP hence discards the sub-MSM partitioning and instead consolidates all computations in each "column". In this way, more points (across all sub-MSMs) can be merged in a bucket, thus saving PMUL operations from one per sub-MSM to one per entire "column".

We further adjust the ordering of the three steps in the algorithm and merge the point-merging tasks within different "columns" to eliminate the *window-reduction* step. We group all PMUL operations with the same scalar component across different windows into buckets, producing only 2^k cross-window point-merging tasks in our MSM workflow. However, the points across different windows cannot be directly added together because their bucket scalars are in different bit positions, i.e., with different weights. GZKP addresses this issue with a preprocessing scheme, as shown in the left-most panel in Figure 5. We pre-compute $2^{t*k} \otimes P_i$ to get P_{i_t} in advance, where t is the window index and 2^{t*k} is the window weight. Note that the original point vector P is generated in the ZKP system setup phase and remains unchanged for a specific application. Consequently, we can pre-compute $2^k \otimes P_i, 2^{2k} \otimes P_i, \ldots, 2^{(\lceil l/k \rceil - 1)*k} \otimes P_i$ for each point P_i, and save these preprocessed points in the GPU global memory. These "weighted" points can then be directly summed in the point-merging tasks. The downside of such preprocessing, however, is the memory space overhead required to store the pre-computed points: over 5 GB at the MSM scale of 2^{21} (with the large integer bit-width of 381-bit), and further grows as the MSM scale increases.

To better balance between the time and space saving in our design, we design a point merging algorithm based on a checkpoint strategy as shown in Algorithm 1. We set a preprocessing interval M to adjust the "frequency" of the preprocessed data, as a control knob to configure based on memory space constraints. GZKP only preprocesses the points with fixed weights, called *checkpoints* (*i.e.* $2^{M*k}, 2^{(2M)*k}, \ldots, 2^{\lfloor \frac{\lceil l/k \rceil}{M} \rfloor * M * k}$). Then during point-merging, it first finds the closest checkpoint to the needed point, and then perform at most $(M-1)*k$ PADD operations to get the desired point.

Note that the idea of preprocessing point vectors is also used in the Straus algorithm [58]. There the points (*i.e.* $2 \otimes P_i, 3 \otimes P_i, \ldots, (2^k - 1) \otimes P_i$) are also pre-computed to accelerate bucket-reduction. However, this scheme scales poorly, as the amount of pre-computation grows too fast with large N, even with a small k value. This is reflected in our performance comparison with MINA [18] (Table 7), which utilizes the Straus algorithm.

Algorithm 1: Point Merging Based On Checkpoints

```
Input: the MSM scale $N$,
  the preprocessing interval $M$,
  the windows size $k$,
  the bucket-info array $p\_index$,
  the preprocessed points (checkpoints) array $pre\_P$
Output: the result point $B$
for $i \leftarrow 0$ to $len(p\_index) - 1$ do
    $t = p\_index[i]/N$;
    $r = p\_index[i]\%N$;
    if $t\%M == 0$ then
        $B = B \oplus pre\_P[(t/M) * N + r]$;
    else
        $tmp = pre\_P[(t/M) * N + r]$;
        for $j \leftarrow 0$ to $(t\%M) * k$ do
            $tmp = tmp \oplus tmp$;
        end
        $B = B \oplus tmp$;
    end
end
```

The point-merging step is the most time-consuming, taking up 90% of the overall MSM execution. To effectively parallelize these cross-window point-merging tasks, GZKP defines the point-merging task as a basic work unit, which sums up all points in the corresponding bucket. Multiple point-merging tasks are assigned to a GPU block for parallel computation. We allocate at least one warp for a basic task unit to execute. As shown in the right side of Figure 5, the tasks corresponding to the buckets (*i.e.* $B_1, B_3, B_{67}, B_{185}, \cdots$) are mapped to the GPU block0, each processed by one warp. In addition, data parallelism within each point-merging task is harvested by our optimized finite field library (Section 4.3) accelerating the PADD operations using the CUDA cooperative groups (CGs) programming model [20]. Threads in a warp are decomposed into multiple CGs (containing 4 threads in the case demonstrated by figure), each parallelizing individual PADD operations. The CG size depends on the integer bit-width adopted by the application and needs to be a factor of 32 to ensure full thread utilization in the warp. The intermediate results from different CGs are stored in the GPU shared memory, before they are added up to get the final point-merging result, to be stored in the GPU global memory.

It is worth noting that the window size k is an important parameter that determines the parallelism of our MSM module as well as the point-merging workload distribution. In general, a larger window size helps reduce the total computational cost of the MSM using the Pippenger algorithm [4]. On the other hand, increasing the window size exponentially increases the number of point-merging tasks, which, when much higher than the number of GPU SMs, introduces non-trivial GPU scheduling overheads. Fortunately, for a specific ZKP application, the size of its MSM task is known. Therefore, GZKP performs profiling-based window configuration.

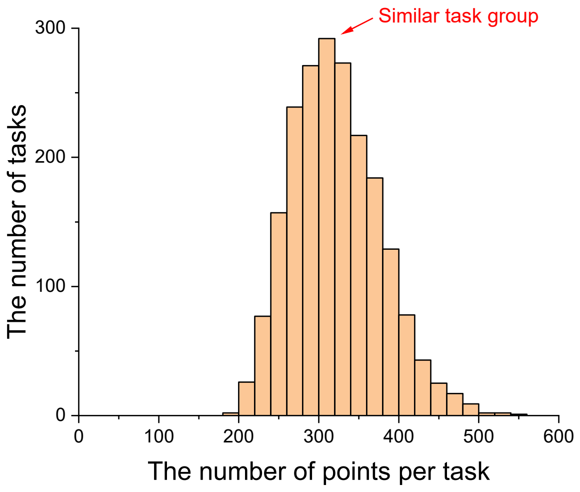

Figure 6: Workload distribution in the point-merging step. Tasks with the same workload (i.e., number of points assigned to the bucket) are grouped into a similar task group.

After the cross-window point-merging step, only one bucket-reduction task is left in our MSM module. The preprocessing scheme essentially turns all windows into one with no weight. Therefore, there are no more windows but 2^k buckets left. For the final step in our MSM module, we calculate $\sum_{j=0}^{2^k-1} j \otimes B_j$ by leveraging the parallel prefix sum algorithm [42], which converts certain sequential computations into equivalent parallel computations. The input array is from the point-merging step and is stored in GPU global memory. We divide it into blocks, each scanned by a single GPU block. The number of GPU blocks is determined by the input array size (2^k) and the number of threads needed for each PADD operation, again in CGs.

4.2 Workload Management

ZKP may exhibit imbalanced data distribution when working with large integers. In particular, the scalar vector $\vec{u}$ in real-world ZKP workloads is highly sparse. The system setup has a lot of bound checks and range constraints, which introduce many 0s and 1s to $\vec{u}$, and thus severe load imbalance in existing systems [16, 18]. For example, the points in bucket 0 require trivial processing and no subsequent point merging.

As an example, we analyze the workload distribution in the point-merging step with scalar vector $\vec{u}$, based on a Zcash MSM execution with a scale of 2^{17} and scalar bit-width of 256. As shown in Figure 6, there is up to 2.85× difference in the numbers of points across all buckets.

When switching from the previous window-based parallel execution to the more fine-grained bucket-based parallelization strategy in GZKP, such imbalance could get more serious as more points are merged into each bucket. On the other hand, GZKP takes advantage of its fine-grained, bucket-based work partitioning strategy to dynamically improve load balance. Given the fairly large number of buckets during point-merging, GZKP groups them by loads (*i.e.*, the number of points contained), ensuring that the tasks in the same group have similar workloads. The bars in the histogram of Figure 6 illustrate an example grouping. GZKP schedules these groups in order, starting from the buckets with the heaviest loads, ensuring that heavy buckets are not left as stragglers.

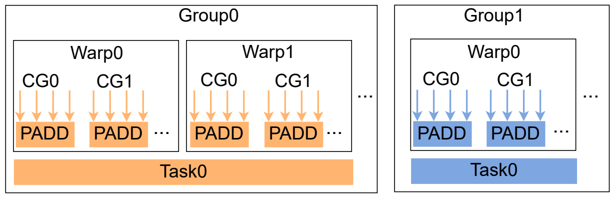

Figure 7: Fine-grained task mapping in GZKP for load balancing.

When scheduling the tasks in each task group to the underlying GPU hardware, GZKP performs *fine-grained task mapping* for workload-aware resource allocation. We allocate GPU warps to the tasks in the same group according to their average loads. As demonstrated in Figure 7, a task in group 1 is mapped to one warp, while another task in group 0 is mapped to two warps since it contains more points. Therefore, each warp will independently manage at most one task (bucket), and multiple CGs in a warp can execute the task in a data-parallel manner.

4.3 Optimized Finite Field Library

GZKP uses a highly optimized finite field library as the underlying tool to accelerate the basic operations in ZKP, mainly modular multiplications [52] and large integer additions. As mentioned earlier, a typical proof generation in Zcash [43] involves billions of these basic operations. Our library supports large integers, currently 256-bit ~ 753-bit, and can be further extended if needed. Due to the length limit, we only briefly introduce it here.

Current mainstream computing devices (*i.e.*, CPU, GPU) cannot directly support the above operations on large integers. Instead, one needs to split them into multiple device-native data types, *e.g.*, 4 machine-word-size (64-bit) integers for a 256-bit integer. The large finite field operations are then implemented using the arithmetic operations supported by the GPU in CUDA [20]. Multiple threads in a warp cooperate to complete the arithmetic operation and use the warp-level primitive to realize the data communication between threads. In this way, we can effectively deal with data carry signals between threads, and merge the multiple individual 64-bit results into the final 256-bit number.

Like earlier works [29, 30], our library exploits the floating-point processing units (the forte of GPUs), which would be otherwise idle in such integer computation tasks, to accelerate modular multiplication. For example, we choose base $D = 2^{52}$ to split a large integer into multiple 52-bit components, resulting in 15 machine-word-size (64-bit) integers to represent a 753-bit integer. The higher-order bits of these integers are filled with 0 to accommodate carry bits. We then convert them to double-precision floating point (DFP), followed by applying Dekker's Method [27] to complete the DFP multiplications and additions without rounding issues. We implement the butterfly operation for NTT and the PADD operation for MSM based on this library. We believe the library itself could be of independent interest beyond GZKP, and can also be applied to improve existing GPU-based ZKP systems, which we demonstrated through our baselines. We report the acceleration results of the finite field library on different modules (Figure 8 and Figure 10) in Section 5.3.

5 EVALUATION

5.1 Methodology

Experimental setup. Our experiments are conducted on a server with dual 2.00 GHz Intel Xeon Gold CPU 5117 processors (each with 28 logical cores) and 256 GB DRAM, running Ubuntu 18.04.2 with CUDA Toolkit version 11.4. It is equipped with four NVIDIA Tesla V100 GPUs (each with 32 GB memory) and one NVIDIA GTX 1080Ti GPU (with 11 GB memory).

Baselines. we compare GZKP with multiple state-of-the-art ZKP systems (using CPU or GPU) based on the zkSNARK protocol, as list in Table 1. Among them, libsnark [50], a C++ library, is to our knowledge the most well-known zkSNARK library, supporting multiple elliptic curves as well as parallel proof generation on multiple CPU cores. MINA [18] is a GPU-accelerated ZKP system, but only accelerates the MSM stage. In addition, bellman and bellperson are recent zkSNARK implementations in Rust, adopted by Zcash [22] and Filecoin [15], respectively. They have similar overall design, while the former is CPU-based and the latter GPU-based. Also, libsnark and GZKP support ALT-BN128, BLS12-381, and MNT4753 elliptic curves, with bit-width of 256, 381, and 753, respectively. MINA only supports the MNT4753 elliptic curve, while both bellperson and bellman support BLS12-381. We evaluate the performance of GZKP with different elliptic curves. For each elliptic curve, we select the best GPU system ("Best-GPU", or "BG") as the baseline, along with the best CPU solution ("Best-CPU", or "BC"), especially when the particular curve is not supported by any prior GPU solutions.

Workloads. We use multiple real-world workloads to evaluate the overall performance of GZKP. Among them, the zkSNARK workloads are generated by xJsnark [47, 49], a high-level framework for developing applications based on zkSNARK, integrating libsnark as a backend for proof generation. The Zcash workload is generated through the tool included in its code repository [24]. The scalar vector $\vec{u}$ in these real-world workloads is sparse. In addition, we use synthetic data generated by libsnark to evaluate the performance of different GZKP modules.

Table 1: Baselines

	Platforms	Supported curves
GZKP	GPU	ALT-BN128, BLS12-381, MNT4753
MINA [18]	GPU	MNT4753
bellperson [16]	GPU	BLS12-381
libsnark [50]	CPU	ALT-BN128, BLS12-381, MNT4753
bellman [23]	CPU	BLS12-381

5.2 Overall Performance

zkSNARK. First, we evaluate the end-to-end proof-generation time (sum of the POLY and MSM stages) using the zkSNARK workloads on one V100 GPU card. More specifically, the actual zkSNARK execution [50] contains seven NTT operations in the POLY stage and five MSM operations in the MSM stage, for generating one proof. We create such zkSNARK workloads with the 753-bit MNT4753 curve.

As shown in Table 2, GZKP achieves, on average, 48.1× and 33.6× speedup against the best-performing CPU-based system (libsnark) and the best-performing GPU-based system (MINA), respectively. Since MINA only performs the MSM stage on the GPU, its overall execution time is obtained by adding libsnark's POLY stage time to its MSM stage time. While MINA provides quite limited improvement over the best CPU solution, GZKP is able to provide a speedup between 14.0× and 48.1× over the former. The reason is that with the scalar vector $\vec{u}$ being sparse in real-world workloads, MINA

Table 2: Performance results (in seconds) of different zkSNARK workloads on V100 GPU with MNT4753 (753-bit) elliptic curve

Application	Vector size	Best-CPU		Best-GPU		GZKP		Overall Speedup (Best-CPU)	Overall Speedup (Best-GPU)
		POLY	MSM	POLY	MSM	POLY	MSM		
AES	16383	0.85	0.83	0.85	0.59	0.004	0.099	16.3×	14.0×
SHA-256	32767	0.97	1.14	0.97	0.90	0.005	0.066	29.8×	26.3×
RSAEnc	98303	3.58	3.77	3.58	1.86	0.022	0.12	53.2×	39.4×
RSASigVer	131071	2.57	4.77	2.57	1.63	0.024	0.13	46.7×	26.7×
Merkle-Tree	294911	10.03	12.33	10.03	3.72	0.06	0.22	78.2×	48.1×
Auction	557055	19.46	14.27	19.46	5.41	0.15	0.37	64.3×	47.4×

Table 3: Performance results (in seconds) of Zcash workload on V100 GPU with BLS12-381 (381-bit) elliptic curve

Zcash workload	Vector size	Best-CPU		Best-GPU		GZKP		Overall Speedup (Best-CPU)	Overall Speedup (Best-GPU)
		POLY	MSM	POLY	MSM	POLY	MSM		
Sapling_Output	8191	0.17	0.21	0.052	0.26	0.001	0.033	11.1×	9.2×
Sapling_Spend	131071	0.43	1.07	0.16	0.50	0.003	0.09	16.7×	7.1×
Sprout	2097151	4.05	9.61	0.69	2.24	0.049	0.25	46.3×	9.8×

suffers from significant load imbalance, while GZKP's design avoids such a problem.

Zcash. Next, we report evaluation results with a Zcash workload, using the 381-bit BLS12-381 curve on one V100 GPU. There are two types of trading methods (*sprout*, *sapling*) [43] with Zcash, which involve different proof generation activities. We report results for both.

As shown in Table 3, GZKP achieves, on average, 24.7× and 8.7× speedup against BC (bellman) and BG (bellperson), respectively. To make a shielded transaction, a combination of those workloads is required. Thus, GZKP could reduce the latency of proof generation for each shielded transaction in Zcash by 37.1× against bellman and provide a speedup of 9.2× over bellperson. The reason is that GZKP accelerates the computation of the two stages better through the new design compared with bellperson, especially improving the performance of the more time-consuming MSM stage by 8×. Here the improvement echoes the zkSNARK results.

Multi-GPU tests. We also evaluate the end-to-end performance of GZKP with four V100 GPUs by using the Zcash workload. The best-performing GPU-based ZKP system (bellperson) also supports multiple GPU cards for MSM stage. For the POLY stage, we assign data-independent NTT operations to different GPU cards for parallel computation. For the MSM stage, we decompose the computation horizontally into 4 smaller sub-MSM tasks, where each task uses all our proposed optimizations, and then assign each of them to a GPU. As shown in Table 4, although additional data communication overhead is introduced between multiple cards, GZKP reduces the proof-generation time further by an average of 2.1× on the basis which only uses one GPU card. Also, due to better scalability, GZKP achieves, on average, 13.2× speedup against BG (bellperson).

Table 4: Performance results (in seconds) of Zcash workload on four V100 GPUs with BLS12-381 (381-bit) elliptic curve

Zcash workload	Vector size	Best-GPU		GZKP		Overall Speedup
		POLY	MSM	POLY	MSM	
Sapling_Output	8191	0.052	0.24	0.0004	0.023	12.7×
Sapling_Spend	131071	0.16	0.31	0.0017	0.049	9.3×
Sprout	2097151	0.69	1.08	0.027	0.074	17.6×

5.3 Breakdown Analysis

We perform breakdown analysis to assess the impact of individual GZKP optimizations, for the two ZKP stages separately.

NTT operation. Here we examine the execution time of a single NTT operation, the major computation within the POLY stage. We use synthetic data generated by libsnark with the MNT4753 and BLS12-381 curves. Note that BLS12-381 curves use 256-bit integers and 256-bit NTT operations in the POLY stage. As shown in Table 5, GZKP achieves, on average, 343.0× and 5.8× speedup against the BC (libsnark) with MNT4753 curve and the BG (bellperson) with BLS12-381 curve on the V100 GPU, respectively. GZKP execution time for a single NTT operation scales linearly with the NTT size as expected, while libsnark fails to achieve so, due to its redundant computation for ω_N^i in each butterfly operation. GZKP avoids this cost by preprocessing before the NTT operation. Specifically, we use a parallel strategy more suitable for GPU. As shown in Figure 2, each iteration i contains 2^i unique ω_N-related values. Our design only needs to store these unique preprocessing values once, without redundant storage. We also avoid the memory access overhead by ensuring that the threads read the preprocessing values continuously. However, such preprocessing is not suitable for libsnark because of its specific parallel strategy, which is not exactly the same as Figure 2. We have modified libsnark to precompute all ω_N-related values, but these values actually take up 16× memory size, e.g., up to 24 GB in 2^{24}-NTT. The performance improvement is only 1.5× on average. Since the preprocessing values increase the overall memory footprint and the data transfers from memory, the extra memory access overhead dwarfs the saved computations.

Table 6 lists results of similar tests, but on a lower-end GPU (GTX1080Ti), with fewer SMs and lower memory bandwidth. GZKP still achieves on average a speedup of 8.9× against bellperson with BLS12-381 curve. The latter is more severely affected by the reduced memory bandwidth. Note that we ignore the preprocessing time required for the butterfly operation for fairness, which is executed serially on the CPU in bellperson and takes on average 5× latency in Table 5, while GZKP performs it in parallel on the GPU. The speedup ratio of GZKP compared to bellperson varies for different NTT sizes. The reason is that with the fixed division of a GPU block to process independent groups, bellperson suffers from sub-optimal block division, while GZKP avoids such problems.

To take a deeper look, we investigate the improvement brought by major GZKP proposals for the POLY stage, using the BLS12-381

Table 5: Performance results (in milliseconds) of single NTT operation on the V100 GPU

NTT Scale	753-bit		256-bit	
	Best CPU	GZKP	Best GPU	GZKP
2^{14}	102	0.15 (697.4×)	0.37	0.05 (8.0×)
2^{16}	212	0.49 (434.4×)	0.48	0.09 (5.3×)
2^{18}	565	1.91 (295.4×)	2.89	0.28 (10.3×)
2^{20}	2110	7.46 (282.5×)	5.19	1.07 (4.8×)
2^{22}	8180	33.67 (242.9×)	12.69	4.96 (2.5×)
2^{24}	32517	141.40 (230.0×)	46.74	20.99 (2.2×)
2^{26}	131441	602.53 (218.1×)	665.84	91.05 (7.3×)

Table 6: Performance results (in milliseconds) of single NTT operation on the GTX1080Ti GPU

NTT Scale	753-bit		256-bit	
	Best CPU	GZKP	Best GPU	GZKP
2^{14}	102	0.33 (305.0×)	0.52	0.06 (9.4×)
2^{16}	212	1.16 (182.8×)	0.98	0.18 (5.5×)
2^{18}	565	6.21 (91.01×)	14.64	0.70 (20.9×)
2^{20}	2110	27.26 (77.30×)	23.80	2.87 (8.3×)
2^{22}	8180	119.82 (68.24×)	70.50	12.83 (5.5×)
2^{24}	32517	539.25 (60.30×)	234.59	56.18 (4.2×)

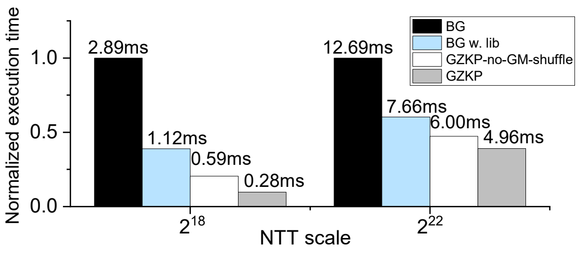

Figure 8: Breakdown analysis of single NTT operation with 256-bit BLS12-381 curve

curve as an example. Here the BG (bellperson) contains a shuffle stage, assigns each GPU block with only one independent group, and creates batches by grouping every 8 iterations. First, to show the improvement brought by our optimized finite field library, we implement a BG-like solution accelerated by this library ("BG w. lib"). Then, we evaluate an intermediate solution ("GZKP-no-GM-shuffle") that removes the shuffling on the global memory. Finally, the full GZKP contains all our proposed optimizations. It adopts smaller independent groups as the work granularity, creates batches by grouping fewer iterations, and performs internal shuffling that promotes continuous memory accesses (Section 3).

As shown in Figure 8, when the NTT scale is 2^{22}, BG w. lib achieves 1.6× speedup against BG. GZKP can achieve another 1.5× speedup against BG w. lib. In particular, for BG, when the NTT scale is 2^{18}, the last batch has 2 iterations but has 2^{16} GPU blocks, and each block only contains 2 threads. Because GPU scheduling uses warp (32 threads) as the unit, there are 30 threads idling. The scheduling overhead is also high due to such many GPU blocks. As the number of iterations in the last batch increases, the influence of such sub-optimal block division in BG gradually reduces. With flexible GPU block assignment, the performance of GZKP's NTT module is almost linear with the NTT scale.

MSM operation. Then, we examine the execution time of a single MSM operation, the major computation within the MSM stage. Similarly, we conduct the experiment on the dense synthetic data created by libsnark with the 753-bit MNT4753, 381-bit BLS12-381, and 256-bit ALT-BN128 curves. As shown in Table 7, GZKP achieves, on average, 12.4× and 7.0× speedup against the best-performing GPU-based ZKP systems (MINA) for MNT4753 curve and (bellperson) for BLS12-381 curve on V100 GPU, respectively. Table 8 gives the results on the lower-end GTX1080Ti, where GZKP still achieves on average 4.3× and 6.1× speedup against the BG MINA and bellperson, respectively. For ALT-BN128 curve, GZKP achieves on average 26.6× speedup against the best-performing CPU-based ZKP system (libsnark).

In addition, we evaluate the memory usage of the GZKP's MSM module on the V100 GPU with different elliptic curves, as shown in Figure 9. Here "GZKP-MNT4" and "GZKP-BLS" refer to GZKP with the MNT4753 curve (adopted by MINA) and the BLS12-381 curve (adopted by bellperson), respectively. For MNT4753 curve, GZKP's memory usage grows slower than that of MINA. When the MSM scale exceeds 2^{22}, MINA execution fails due to insufficient GPU global memory. For BLS12-381 curve, GZKP does consume more memory than bellperson, but stays stable beyond 2^{22} and

Table 7: Performance results (in seconds) of single MSM operation (G1-type) on the V100 GPU

MSM Scale	753-bit		381-bit		256-bit	
	Best GPU	GZKP	Best GPU	GZKP	Best CPU	GZKP
2^{14}	0.36	0.02(17.9×)	0.032	0.004(8.0×)	0.07	0.004(18.1×)
2^{16}	0.48	0.05(9.2×)	0.052	0.007(7.4×)	0.18	0.006(29.3×)
2^{18}	1.99	0.16(12.1×)	0.14	0.020(7.2×)	0.45	0.015(30.1×)
2^{20}	7.2	0.60(12.1×)	0.53	0.062(8.5×)	1.48	0.045(32.9×)
2^{22}	28.1	2.66(10.7×)	1.35	0.24(5.6×)	4.90	0.17(28.7×)
2^{24}	-	11.3	6.55	1.10(6.0×)	17.27	0.72(23.8×)
2^{26}	-	40.7	24.42	4.00(6.1×)	65.70	2.79(23.5×)

Table 8: Performance results (in seconds) of single MSM operation (G1-type) on the GTX1080Ti GPU

MSM Scale	753-bit		381-bit		256-bit	
	Best GPU	GZKP	Best GPU	GZKP	Best CPU	GZKP
2^{14}	0.35	0.08(4.5×)	0.093	0.015(6.2×)	0.07	0.007(10.3×)
2^{16}	1.00	0.20(5.0×)	0.20	0.032(6.3×)	0.18	0.013(13.5×)
2^{18}	2.71	0.71(3.8×)	0.64	0.073(8.8×)	0.45	0.032(14.1×)
2^{20}	10.07	2.51(4.0×)	1.43	0.26(5.4×)	1.48	0.10(14.5×)
2^{22}	-	11.91	5.16	1.02(5.2×)	4.90	0.37(13.2×)
2^{24}	-	46.83	19.86	4.16(4.8×)	17.27	1.50(11.5×)

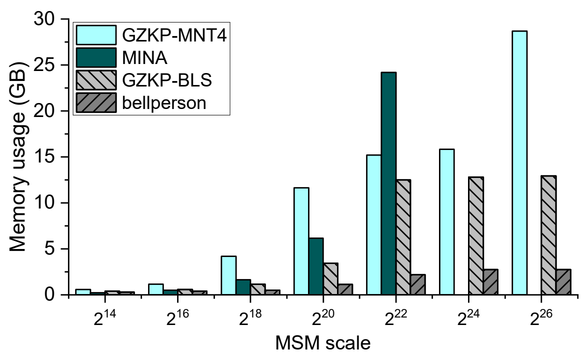

Figure 9: MSM memory usage with different curves on V100

well within the total global memory size, due to Algorithm 1 that adapts to GPUs of different memory size. As GZKP significantly outperforms bellperson, this shows that it is able to utilize the available global memory size for better performance, while bellperson under-utilizes the GPU memory.

To take a deeper look, we investigate the improvement brought by major GZKP proposals for the MSM stage, using the BLS12-381 curve on the V100 GPU as an example. Here the BG (bellperson) is the baseline, which explores the data parallelism by decomposing the MSM task horizontally into multiple sub-MSM tasks. Since BG executes the point-merging and bucket-reduction steps on the GPU side and the last window-reduction step on the CPU side, we

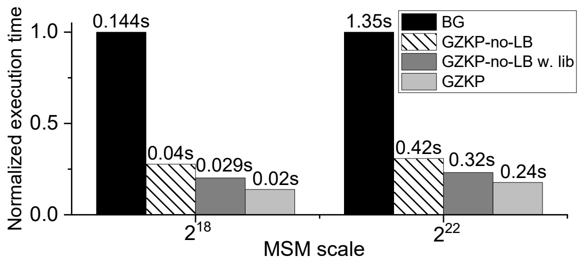

Figure 10: Breakdown analysis for MSM module with BLS12-381 (381-bit) elliptic curve on one V100 GPU

use GZKP's MSM module to evaluate the performance improvement brought by the GPU-based finite field library. We evaluate two intermediate solutions, "GZKP-no-LB" and "GZKP-no-LB w. lib". The former implements the optimized MSM computation process by leveraging bucket-based task partitioning. The latter uses the optimized finite field library based on the former. Finally, the GZKP contains all our proposed optimizations, including computation consolidation in Section 4.1 and workload management in Section 4.2.

As shown in Figure 10, with the MSM scale 2^{22}, the breakdown analysis shows that GZKP-no-LB achieves 3.25× speedup compared to BG. And the GZKP-no-LB w. lib further improves the performance by 33% over GZKP-no-LB. After enabling load balance, GZKP achieves 5.6× speedup against the BG. The main improvement for the MSM module in GZKP is due to the computation consolidation optimization. Besides, Our MSM module ensures load balance, whether for sparse $\vec{u}$ or dense $\vec{h}$, through fine-grained task mapping.

6 RELATED WORK

zkSNARK protocol and implementation of ZKP systems. Recently, many works [2, 3, 5, 6, 9, 34, 39, 51, 54, 57, 60] have made lots of efforts to optimize the zkSNARK protocol and make it more practical. Since different ZKP applications [7, 15, 19, 56] use different elliptic curves, there are many ZKP system implementations based on the zkSNARK protocol (*i.e.* libsnark [50], bellman [23], MINA [18] and bellperson [16]), and most of these implementations only support specific curves. GZKP supports different types of elliptic curves to serve a range of applications by leveraging the optimized finite field library, and the modular design enables GZKP to be easily integrated into the existing ZKP systems as a high-performance back-end to achieve fast proof generation.

Hardware acceleration for ZKP system. There are a few designs improving the performance of ZKP with hardware acceleration, such as FPGA [25, 28, 63] and GPU [16, 18]. A recent work called PipeZK [63] is proposed to accelerate zkSNARK through customized hardware accelerator, which pipelines the key operations (*i.e.* NTT and MSM) in proof generation. These designs cannot work well with the increasing scale of the input vector. Current GPU-based ZKP systems (*i.e.* MINA [18], bellperson [16]) in industry accelerate the prover computation by parallelizing those key operations. These systems suffer from huge data shuffling overhead caused by large bit-width integers and severe load imbalance caused by sparse data distribution. GZKP outperforms them as we have demonstrated earlier.

GPU acceleration for NTT and MSM. Another body of previous works [10, 26, 36, 38, 46] accelerate the NTT operation on GPU and involve data shuffling to ensure continuous memory accesses during computation, whose cost grows fast with increasing bit-width. These designs cannot meet the scalability requirements of NTT operation in ZKP, where the NTT scale is up to a million, and the bit-width of elements is larger than 256-bit. Some related works [1, 13, 33, 53, 59] accelerate the basic PADD/PMUL operations using GPUs to speedup elliptic curve cryptography. These acceleration methods can not directly applied to ZKP MSM due to the computation scale (millions of PMUL/PADD) and the scalar bit-width (up to 753-bit). GZKP reduces the calculation complexity of the MSM by leveraging Pippenger-like algorithms [4, 55, 58] and speedup the basic PADD operation using the optimized finite field library.

7 DISCUSSION AND FUTURE WORK

GZKP focuses on accelerating privacy-preserving applications based on ZKP protocols and provides good performance as demonstrated in Section 5. Here we discuss GZKP's wider application and its correctness.

NTT-based encryption algorithms. NTT is a key building block of many homomorphic encryption (HE) algorithms, in addition to ZKP. These two types of cryptographic algorithms have different performance requirements. ZKP focuses more on the latency of a single NTT operation, as the multiple NTTs are difficult to parallelize in many protocols [11, 51]. Therefore, in GZKP we use all the compute resources of a GPU for a single NTT operation, plus asynchronous data copying to reduce host-GPU data transfer overhead. In contrast, HE usually runs multiple NTT operations on the GPU simultaneously (NTT batching [38, 45]) to achieve higher throughput and GPU utilization, enabled by the smaller NTT scales and the independent nature of multiple NTTs. Our design adopts smaller independent groups as the task granularity, making it suitable for throughput-oriented NTT applications with the aforementioned batching techniques. We plan to expand GZKP to support HE algorithms in our future work.

Verification of GZKP. Note that GZKP focuses on efficient parallel acceleration of core operations (NTT and MSM) in the ZKP protocol. It does not modify the ZKP protocol itself, hence has no impact on its security. Furthermore, proof generation on private user input data usually runs on the user's local machine. Implementation bugs may generate wrong proofs, which could be addressed by formal verification of GZKP. We will leave this as future work.

8 CONCLUSION

Zero-knowledge proof can guarantee the integrity and confidentiality of transactions and is increasingly used in applications like cryptocurrency. However, its high computation overhead hinders its deployment in online services. In this paper, we present GZKP, a GPU-based ZKP system that offers at least an order of magnitude improvement over current state-of-the-art solutions. Our results and experience reveal that there are a significant amount of opportunities for improving the parallel execution efficiency of ZKP proof generation. Meanwhile, ZKP requires targeted optimizations to handle large integer operations that dominate its execution.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers and the shepherd for their valuable comments. We thank Dong Zhou and Xiaoxu Guo for their participation. This work was supported in part by the National Key R&D Program of China under Grant 2020AAA0108501, and the Key R&D Program of Hubei under Grant 2020BAA020.

REFERENCES

[1] Samuel Antao, Jean-Claude Bajard, and Leonel Sousa. 2010. Elliptic curve point multiplication on GPUs. In *ASAP 2010-21st IEEE International Conference on Application-specific Systems, Architectures and Processors*. IEEE, 192–199. https://doi.org/10.1109/ASAP.2010.5541000

[2] Eli Ben-Sasson, Iddo Bentov, Alessandro Chiesa, Ariel Gabizon, Daniel Genkin, Matan Hamilis, Evgenya Pergament, Michael Riabzev, Mark Silberstein, Eran Tromer, and Madars Virza. 2017. Computational Integrity with a Public Random String from Quasi-Linear PCPs. In *Advances in Cryptology – EUROCRYPT 2017*, Jean-Sébastien Coron and Jesper Buus Nielsen (Eds.). Springer International Publishing, Cham, 551–579. https://doi.org/10.1007/978-3-319-56617-7_19

[3] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. 2014. Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture. (2014), 781–796. https://dl.acm.org/doi/10.5555/2671225.2671275

[4] Daniel J Bernstein, Jeroen Doumen, Tanja Lange, and Jan-Jaap Oosterwijk. 2012. Faster batch forgery identification. In *International Conference on Cryptology in India*. Springer, 454–473. https://doi.org/10.1007/978-3-642-34931-7_26

[5] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. 2012. From Extractable Collision Resistance to Succinct Non-Interactive Arguments of Knowledge, and Back Again. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference* (Cambridge, Massachusetts) *(ITCS '12)*. Association for Computing Machinery, New York, NY, USA, 326–349. https://doi.org/10.1145/2090236.2090263

[6] Manuel Blum, Paul Feldman, and Silvio Micali. 1988. Non-Interactive Zero-Knowledge and Its Applications. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*. Association for Computing Machinery, New York, NY, USA, 103–112. https://doi.org/10.1145/62212.62222

[7] Joseph Bonneau, Izaak Meckler, Vanishree Rao, and Evan Shapiro. 2020. Coda: Decentralized cryptocurrency at scale. *Cryptology ePrint Archive* (2020). https://eprint.iacr.org/2020/352

[8] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. 2020. ZEXE: Enabling Decentralized Private Computation. In *2020 IEEE Symposium on Security and Privacy (SP)*. 947–964. https://doi.org/10.1109/SP40000.2020.00050

[9] Benedikt Bünz, Alessandro Chiesa, Pratyush Mishra, and Nicholas Spooner. 2020. Proof-carrying data from accumulation schemes. *Cryptology ePrint Archive* (2020). https://eprint.iacr.org/2020/499

[10] Liangyu Chen, Svyatoslav Covanov, Davood Mohajerani, and Marc Moreno Maza. 2017. Big Prime Field FFT on the GPU. In *Proceedings of the 2017 ACM on International Symposium on Symbolic and Algebraic Computation* (Kaiserslautern, Germany) *(ISSAC '17)*. Association for Computing Machinery, New York, NY, USA, 85–92. https://doi.org/10.1145/3087604.3087657

[11] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. 2020. Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS. In *Advances in Cryptology – EUROCRYPT 2020*, Anne Canteaut and Yuval Ishai (Eds.). Springer International Publishing, Cham, 738–768. https://doi.org/10.1007/978-3-030-45721-1_26

[12] Eleanor Chu and Alan George. 1999. *Inside the FFT black box: serial and parallel fast Fourier transform algorithms*. CRC press.

[13] Aaron E. Cohen and Keshab K. Parhi. 2010. GPU accelerated elliptic curve cryptography in GF(2m). In *2010 53rd IEEE International Midwest Symposium on Circuits and Systems*. 57–60. https://doi.org/10.1109/MWSCAS.2010.5548560

[14] James W Cooley and John W Tukey. 1965. An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation* 19, 90 (1965), 297–301. https://doi.org/10.2307/2003354

[15] Filecoin Corp. 2022. https://filecoin.io/.

[16] Filecoin Corp. 2022. bellperson: gpu parallel acceleration for zk-snark. https://github.com/filecoin-project/bellperson.

[17] J.P. Morgan Quorum Corp. 2022. https://www.goquorum.com/.

[18] MINA Corp. 2019. GPU Groth16 prover (3x faster than CPU). https://github.com/MinaProtocol/gpu-groth16-prover-3x.

[19] Mina Corp. 2022. https://minaprotocol.com/.

[20] NVIDIA Corp. 2021. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[21] StarkWare Corp. 2022. https://starkware.co/.

[22] Zcash Corp. 2022. https://z.cash/.

[23] ZKCrypto Corp. 2022. bellman: a crate for building zk-SNARK circuits. https://github.com/zkcrypto/bellman.

[24] Zcash Corp. 2022. librustzcash: a (work-in-progress) set of Rust crates for working with Zcash. https://github.com/zcash/librustzcash.

[25] Zcash Corp. 2022. Zcash FPGA acceleration engine. https://github.com/ZcashFoundation/zcash-fpga.

[26] Wei Dai and Berk Sunar. 2015. cuHE: A homomorphic encryption accelerator library. In *International Conference on Cryptography and Information Security in the Balkans*. Springer, 169–186. https://doi.org/10.1007/978-3-319-29172-7_11

[27] Theodorus Jozef Dekker. 1971. A floating-point technique for extending the available precision. *Numer. Math.* 18, 3 (1971), 224–242. https://doi.org/10.1007/BF01397083

[28] Benjamin Devlin. 2022. FPGA SNARK prover targeting the bn128 curve. https://github.com/bsdevlin/fpga_snark_prover.

[29] Jiankuo Dong, Fangyu Zheng, Niall Emmart, Jingqiang Lin, and Charles Weems. 2018. sDPF-RSA: Utilizing floating-point computing power of GPUs for massive digital signature computations. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 599–609. https://doi.org/10.1109/IPDPS.2018.00069

[30] Niall Emmart, Fangyu Zheng, and Charles Weems. 2018. Faster modular exponentiation using double precision floating point arithmetic on the GPU. In *2018 IEEE 25th Symposium on Computer Arithmetic (ARITH)*. IEEE, 130–137. https://doi.org/10.1109/ARITH.2018.8464792

[31] Ariel Gabizon, Zachary J Williamson, and Oana Ciobotaru. 2019. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *Cryptology ePrint Archive* (2019). https://eprint.iacr.org/2019/953

[32] Hisham S. Galal and Amr M. Youssef. 2018. Verifiable Sealed-Bid Auction on the Ethereum Blockchain. In *Financial Cryptography and Data Security: FC 2018 International Workshops, BITCOIN, VOTING, and WTSC, Nieuwpoort, Curaçao, March 2, 2018, Revised Selected Papers* (Nieuwpoort, Curaçao). Springer-Verlag, Berlin, Heidelberg, 265–278. https://doi.org/10.1007/978-3-662-58820-8_18

[33] Lili Gao, Fangyu Zheng, Niall Emmart, Jiankuo Dong, Jingqiang Lin, and Charles Weems. 2020. DPF-ECC: Accelerating Elliptic Curve Cryptography with Floating-Point Computing Power of GPUs. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 494–504. https://doi.org/10.1109/IPDPS47924.2020.00058

[34] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. 2013. Quadratic Span Programs and Succinct NIZKs without PCPs. In *Advances in Cryptology – EUROCRYPT 2013*, Thomas Johansson and Phong Q. Nguyen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 626–645. https://doi.org/10.1007/978-3-642-38348-9_37

[35] Craig Gentry. 2009. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*. Association for Computing Machinery, New York, NY, USA, 169–178. https://doi.org/10.1145/1536414.1536440

[36] Jia-Zheng Goey, Wai-Kong Lee, Bok-Min Goi, and Wun-She Yap. 2021. Accelerating number theoretic transform in GPU platform for fully homomorphic encryption. *The Journal of Supercomputing* 77 (2021), 1455–1474. https://doi.org/10.1007/s11227-020-03156-7

[37] Shafi Goldwasser, Silvio Micali, and Chales Rackoff. 2019. The Knowledge Complexity of Interactive Proof-Systems. (2019), 203–225. https://doi.org/10.1145/3335741.3335750

[38] Naga K Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith, and John Manferdelli. 2008. High performance discrete Fourier transforms on graphics processors. In *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE, 1–12. https://doi.org/10.1109/SC.2008.5213922

[39] Jens Groth. 2010. Short pairing-based non-interactive zero-knowledge arguments. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 321–340. https://doi.org/10.1007/978-3-642-17373-8_19

[40] Jens Groth. 2016. On the Size of Pairing-Based Non-interactive Arguments. In *Advances in Cryptology – EUROCRYPT 2016*, Marc Fischlin and Jean-Sébastien Coron (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 305–326. https://doi.org/10.1007/978-3-662-49896-5_11

[41] Darrel Hankerson, Alfred J Menezes, and Scott Vanstone. 2006. *Guide to elliptic curve cryptography*. Springer Science & Business Media.

[42] Mark Harris, Shubhabrata Sengupta, and John D Owens. 2007. Parallel prefix sum (scan) with CUDA. *GPU gems* 3, 39 (2007), 851–876.

[43] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. 2022. Zcash Protocol Specification. (2022). https://zips.z.cash/protocol/protocol.pdf

[44] QED it Corp. 2017. https://qed-it.com/.

[45] Wonkyung Jung, Sangpyo Kim, Jung Ho Ahn, Jung Hee Cheon, and Younho Lee. 2021. Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with GPUs. *IACR Transactions on Cryptographic*

Hardware and Embedded Systems (2021), 114–148. https://doi.org/10.46586/tches.v2021.i4.114-148

[46] Sangpyo Kim, Wonkyung Jung, Jaiyoung Park, and Jung Ho Ahn. 2020. Accelerating Number Theoretic Transformations for Bootstrappable Homomorphic Encryption on GPUs. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 264–275. https://doi.org/10.1109/IISWC50251.2020.00033

[47] Ahmed Kosba. 2021. jsnark: a Java library for building circuits for preprocessing zk-SNARKs. https://github.com/akosba/jsnark.

[48] Ahmed Kosba, Dimitrios Papadopoulos, Charalampos Papamanthou, and Dawn Song. 2020. MIRAGE: Succinct Arguments for Randomized Algorithms with Applications to Universal Zk-SNARKs. In *Proceedings of the 29th USENIX Conference on Security Symposium*. USENIX Association, USA, Article 120, 18 pages. https://dl.acm.org/doi/10.5555/3489212.3489332

[49] Ahmed Kosba, Charalampos Papamanthou, and Elaine Shi. 2018. xJsnark: A framework for efficient verifiable computation. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 944–961. https://doi.org/10.1109/SP.2018.00018

[50] SCIPR LAB. 2022. libsnark: a C++ library for zkSNARK proofs. https://github.com/scipr-lab/libsnark.

[51] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. 2019. Sonic: Zero-Knowledge SNARKs from Linear-Size Universal and Updatable Structured Reference Strings. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 2111–2128. https://doi.org/10.1145/3319535.3339817

[52] Peter L Montgomery. 1985. Modular multiplication without trial division. *Mathematics of computation* 44, 170 (1985), 519–521.

[53] Wuqiong Pan, Fangyu Zheng, Yuan Zhao, Wen-Tao Zhu, and Jiwu Jing. 2016. An efficient elliptic curve cryptography signature server with GPU acceleration. *IEEE Transactions on Information Forensics and Security* 12, 1 (2016), 111–122. https://doi.org/10.1109/TIFS.2016.2603974

[54] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. 2013. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 238–252. https://doi.org/10.1109/SP.2013.47

[55] Nicholas Pippenger. 1976. On the evaluation of powers and related problems. In *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*. IEEE Computer Society, 258–263. https://doi.org/10.1109/SFCS.1976.21

[56] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. 2014. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 459–474. https://doi.org/10.1109/SP.2014.36

[57] Srinath Setty. 2020. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In *Annual International Cryptology Conference*. Springer, 704–737. https://doi.org/10.1007/978-3-030-56877-1_25

[58] Ernst G Straus. 1964. Addition chains of vectors (problem 5125). *Amer. Math. Monthly* 70, 806-808 (1964), 16.

[59] Robert Szerwinski and Tim Güneysu. 2008. Exploiting the power of GPUs for asymmetric cryptography. In *International Workshop on Cryptographic hardware and embedded systems*. Springer, 79–99. https://doi.org/10.1007/978-3-540-85053-3_6

[60] Howard Wu, Wenting Zheng, Alessandro Chiesa, Raluca Ada Popa, and Ion Stoica. 2018. DIZK: A Distributed Zero Knowledge Proof System. Cryptology ePrint Archive, Paper 2018/691. (2018). https://eprint.iacr.org/2018/691 https://eprint.iacr.org/2018/691.

[61] Jiaheng Zhang, Zhiyong Fang, Yupeng Zhang, and Dawn Song. 2020. Zero knowledge proofs for decision tree predictions and accuracy. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 2039–2053. https://doi.org/10.1145/3372297.3417278

[62] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2017. vSQL: Verifying arbitrary SQL queries over dynamic outsourced databases. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 863–880. https://doi.org/10.1109/SP.2017.43

[63] Ye Zhang, Shuo Wang, Xian Zhang, Jiangbin Dong, Xingzhong Mao, Fan Long, Cong Wang, Dong Zhou, Mingyu Gao, and Guangyu Sun. 2021. PipeZK: Accelerating Zero-Knowledge Proof with a Pipelined Architecture. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 416–428. https://doi.org/10.1109/ISCA52012.2021.00040

[64] Zhichao Zhao and T-H Hubert Chan. 2015. How to vote privately using bitcoin. In *International Conference on Information and Communications Security*. Springer, 82–96. https://doi.org/10.1007/978-3-319-29814-6_8

[65] zkSync Corp. 2022. https://zksync.io/.

Received 2022-07-07; accepted 2022-09-22