PipeZK: Accelerating Zero-Knowledge Proof with a Pipelined Architecture

Ye Zhang[1,5] Shuo Wang[1] Xian Zhang[3] Jiangbin Dong[4,7] Xingzhong Mao[7] Fan Long[6]
Cong Wang[8] Dong Zhou[2] Mingyu Gao[2,7*] Guangyu Sun[1*]

Peking University[1] Tsinghua University[2] Microsoft Research[3] Xi'an Jiaotong University[4]
Shanghai Tree-Graph Blockchain Research Institute[5] University of Toronto[6]
Institute for Interdisciplinary Information Core Technology, Xi'an[7]
International Digital Economy Academy at Guangdong-Hong Kong-Macau Greater Bay Area[8]

{zhangye1998, gsun}@pku.edu.cn, gaomy@tsinghua.edu.cn

***Abstract*—Zero-knowledge proof (ZKP) is a promising cryptographic protocol for both computation integrity and privacy. It can be used in many privacy-preserving applications including verifiable cloud outsourcing and blockchains. The major obstacle of using ZKP in practice is its time-consuming step for proof generation, which consists of large-size polynomial computations and multi-scalar multiplications on elliptic curves. To efficiently and practically support ZKP in real-world applications, we propose PipeZK, a pipelined accelerator with two subsystems to handle the aforementioned two intensive compute tasks, respectively. The first subsystem uses a novel dataflow to decompose large kernels into smaller ones that execute on bandwidth-efficient hardware modules, with optimized off-chip memory accesses and on-chip compute resources. The second subsystem adopts a lightweight dynamic work dispatch mechanism to share the heavy processing units, with minimized resource underutilization and load imbalance. When evaluated in 28 nm, PipeZK can achieve 10x speedup on standard cryptographic benchmarks, and 5x on a widely-used cryptocurrency application, Zcash.**

I. Introduction

Zero-knowledge proof (ZKP) blossoms rapidly in recent years, drawing attentions from both researchers and practitioners. In short, it is a family of cryptographical protocols that allow one party (called the *prover*) to convince the others (called the *verifier*s) that a computational statement is true, without leaking any information. For example, if a program P outputs the result y on a public input x and a secret input w, using a ZKP protocol, the prover can assure the verifiers that she knows the secret w that satisfies $P(x, w) = y$ without revealing the value of w.

As a fundamental primitive in modern cryptography, ZKP has the potential to be widely used in many privacy-critical applications to enable secure and verifiable data processing, including electronic voting [54], online auction [26], anonymous credentials [23], verifiable database outsourcing [52], verifiable machine learning [51], privacy-preserving cryptocurrencies [22], [47], and various smart contracts on blockchains [37]. More specifically, verifiable outsourcing, as a promising use case of ZKP, allows a weak client to outsource computations to the powerful cloud, and also efficiently verify the correctness of the returned results [52], [53]. Another widely deployed application of ZKP is blockchains and cryptocurrencies. With ZKP, the intensive computations can be moved off-chain and each node only needs to verify the integrity of a much more lightweight proof on the critical path [1], [22], [47].

Since its birth [30], tremendous effort has been made by cryptography researchers to make ZKP more practical. Among newly invented ones, zk-SNARK, which stands for *Zero-Knowledge Succinct Non-Interactive Argument of Knowledge*, is widely considered as a promising candidate. As its name suggests, zk-SNARK generates succinct proofs — often within hundreds of bytes regardless of the complexity of the program, and these proofs are very fast to verify. Because of these two properties, we are seeing more and more deployments of zk-SNARK in real-world applications, especially in the blockchain community. [1], [4], [7], [10], [12], [39].

Although zk-SNARK proofs are succinct and fast to verify, their generation remains an obstacle in large-scale zk-SNARK adoption. To generate proofs for a program, it is typical to first translate the program into a constraint system, the size of which is usually several times larger than the initial program, and could be up to a few millions. The prover then performs a number of arithmetic operations over a large finite field. The actual number of operations required is protocol-specific, but is always super-linear comparing to the number of constraints, hence even larger. As a result, it takes much longer to generate the zk-SNARK proof of a program than verifying it, sometimes up to hundreds of times, and could be up to a few minutes just for a *single* payment transaction [47].

In this paper, we present PipeZK, an efficient pipelined architecture for accelerating zk-SNARK. PipeZK mainly involves two subsystems, for the polynomial computations with large-size number theoretic transforms (NTTs), and for the multi-scalar multiplications that execute vector inner products on elliptic curves (ECs). These two phases are the most

* Mingyu Gao and Guangyu Sun are the co-corresponding authors.

compute-intensive parts. We implement them as specialized hardware accelerators, and combine with the CPU to realize a heterogeneous end-to-end system for zk-SNARK.

For the polynomial computation subsystem, we notice that the large-size NTTs (up to a million elements) result in sigificant challenges for both off-chip memory accesses and on-chip compute resources, due to the irregular strided access patterns similar to classical FFTs, and the large bitwidth (up to 768 bits) of each element. We propose a novel high-level dataflow that recursively decomposes the large NTT kernels into smaller ones, which can then be efficiently executed on a bandwidth-efficient NTT hardware module that uses lightweight FIFOs internally to realize the strided accesses. We also leverage data tiling and on-chip matrix transpose to improve off-chip bandwidth utilization.

For the multi-scalar multiplication subsystem, rather than simply replicating multiple processing units for EC operations, we exploit the large numbers of EC multiplications in the vector inner products, and use Pippenger algorithm [43] to share the dominant EC processing units with a lightweight dynamic work dispatch mechanism. This alleviates the resource underutilization and load imbalance issues when the input data have unpredictable value distributions. Furthermore, we scale the system in a coarse-grained manner to allow each processing unit to work independently from each other, while guaranteeing that there are no stragglers even when data distributions are highly pathological.

In summary, our contributions in this paper include:

- We designed a novel module, which decomposes a large-scale polynomial computation into small tiles and processes them in a pipeline style. It achieves high efficiency in both off-chip memory bandwidth and on-chip logic resource utilization.
- We designed a novel module for multi-scalar point multiplications on elliptic curves. It leverages an optimized algorithm and a pipelined dataflow to achieve high processing throughput.
- We implemented a prototype of the proposed architecture in RTL and synthesized our design as a 28 nm ASIC, and evaluated it in an end-to-end heterogeneous system with a host CPU. Compared to state-of-the-art approaches, the overall system can achieve 10x speedup for small-size standard cryptographic benchmarks on average, and 5x for a real-world large-scale application, Zcash [47]. When individually executed, the two subsystems of PipeZK can achieve 197x to 77x speedup, respectively.

Beyond our accelerator design, both subsystems in PipeZK could be of independent interest to a wider range of applications. The NTT module is the key building block in homomorphic encryption (HE) [29] and modern public-key encryption schemes [38] based on Ring Learning With Errors (R-LWE) problems [44]. The multi-scalar multiplication module is commonly used in vector commitments [19] and many pairing-based proof systems [32], [42]. We expect our architecture insights would inspire more opportunities in making modern cryptographic algorithms more practical to use towards general-purpose secure computation.

II. Background and Motivation

Zero-knowledge proof (ZKP) is a powerful cryptographic primitive that has recently been adopted to many real-world applications [22], [23], [26], [37], [47], [51], [52], [54], and drawn a lot of attentions in both academia [16]–[18], [20], [25], [28], [42], [48] and industry [1], [4], [7], [10], [12], [39]. ZKP allows the *prover* to prove to the *verifier* that a given statement of the following form is true: "given a function F and an input x, I know a secret witness w that makes $F(x, w) = 0$." More specifically, the prover can generate a proof, whose validity can be checked by the verifier. However, even though the verifier gets the proof and is able to verify its validity, she cannot obtain any information about w itself. The prover's secret remains secure after the proving process. As a result, the zero-knowledge property of ZKP provides a strong guarantee for the prover's privacy, as she can prove to others that she knows some private information (i.e., w) without leaking it.

A. Applications of Zero-Knowledge Proof

As one of the fundamental primitives in modern cryptography, ZKP can be widely used in many security applications as a basic building block to enable real-world secure and verifiable data processing. Generally speaking, ZKP allows two or multiple parties to perform compute tasks in a cooperative but secure manner, in the sense that one party can convince the others that her result is valid without accidentally leaking any sensitive information. Many real-world applications can benefit from these properties, including electronic voting [54], online auction [26], anonymous credentials [23], verifiable database outsourcing [52], verifiable machine learning [51], privacy-preserving cryptocurrencies [22], [47], and various smart contracts on blockchains [37].

A promising example application of ZKP is verifiable outsourcing [27], in which case a client with only weak compute power outsources a compute task to a powerful server, e.g., a cloud datacenter, who computes on potentially sensitive data to generate a result that is returned to the client. Examples include database SQL queries [52] and machine learning jobs [51]. In such a scenario, the client would like to ensure the result is indeed correct, while the server is not willing to expose any sensitive data. ZKP allows the server to also provide a proof associated with the result, which the client can use to efficiently check the integrity. The zero-knowledge property allows the prover to make arbitrary statements about (i.e., to compute functions on) the sensitive data without worrying about exposing them, therefore naturally supporting theoretically general-purpose outsourcing computations.

Another widely deployed application of ZKP is blockchains and cryptocurrencies. Conventional blockchain-based applications require every node in the system to execute the same on-chain computations to update the states, which brings a large overhead with long latency. ZKP enables private decentralized verifiable computations which are moved off-chain, and each node only checks the integrity of a lightweight proof to discover illegal state transitions. For instance, zk-Rollup [1] packs many transactions in one proof and allows the nodes to check their integrity by efficiently verifying the proof. Other work even enables verifying the integrity of the whole blockchain using one succinct proof [39]. This feature greatly increases the blockchain scalability. Furthermore, the zero-knowledge property allows users to make confidential transactions while still being able to prove the validity of each transaction. Zcash [47] and Pinocchio Coin [22] are such examples, where the transaction details including the amount of money and the user addresses are hidden.

B. Computation Requirements of Zero-Knowledge Proof

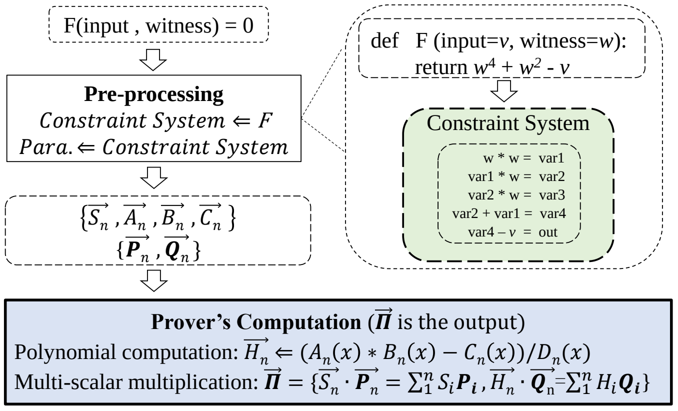
It is natural to imagine that realizing such a counter-intuitive ZKP functionality would require huge computation and communication costs. Since its first introduction by Goldwasser et al. [30], there have been significant improvements in the computation efficiency of ZKP to make it more practical. zk-SNARK [32], as the state-of-the-art ZKP protocol, allows the prover to generate a *succinct* proof, which greatly reduces the verification cost. Formally speaking, the proof of zk-SNARK has three important properties: *correctness*, *zero-knowledge*, and *succinctness*. Correctness means that if the verification passes, then the prover's statement is true, i.e., the prover does know the secret witness w. Zero-knowledge means that the proof does not leak any information about w. And succinctness means that the size of the proof is small (e.g., 128 bytes) and it is also fast to verify (e.g., within 2 milliseconds), regardless of how complicated the original statement might be.

Unfortunately, although the proof verification is fast, generating such a proof at the prover side with zk-SNARK has considerable computation overheads and can take a great amount of time, which hinders zk-SNARK from wide adoption in real-world applications. Therefore, this work focuses on the workflow and the key components of the prover's computation [32], which is our target for hardware acceleration.

For a specific implementation of zk-SNARK, a security parameter λ is first decided to trade off the computation complexity and the security strength, by specifying the data width used. A larger λ provides stronger security guarantees but also introduces significantly higher computation cost. Typically, λ ranges from 256-bit to 768-bit.[1]

As illustrated in Figure 1, the prover first goes through a pre-processing phase, during which the function F, typically written in some high-level programming languages, is first compiled into a set of arithmetic constraints, called "rank-1 constraint system (R1CS)". The constraint system contains a number of linear or polynomial equations of the input x and the witness w. Determined by the complexity of the function F, the number of equations in the constraint system could be as many as up to millions for real-world applications.

[1]Here we abuse the notion of security parameter for simplicity, since it is usually directly related to the bit width of parameters and the underlying elliptic curve.



F(input , witness) = 0
Pre-processing
Constraint System ⇐ F
Para. ⇐ Constraint System
$\{\vec{S_n}, \vec{A_n}, \vec{B_n}, \vec{C_n}\}$
$\{\vec{P_n}, \vec{Q_n}\}$
def F (input=v, witness=w):
return $w^4 + w^2 - v$
Constraint System
w * w = var1
var1 * w = var2
var2 * w = var3
var2 + var1 = var4
var4 – v = out
Prover's Computation ($\vec{\Pi}$ is the output)
Polynomial computation: $\vec{H_n} \Leftarrow (A_n(x) * B_n(x) - C_n(x))/D_n(x)$
Multi-scalar multiplication: $\vec{\Pi} = \{\vec{S_n} \cdot \vec{P_n} = \sum_1^n S_i P_i, \vec{H_n} \cdot \vec{Q_n} = \sum_1^n H_i Q_i\}$

Fig. 1. The workflow of the prover. The illustrated $F(x, w)$ has a constraint system size of five (i.e. $n = 5$).

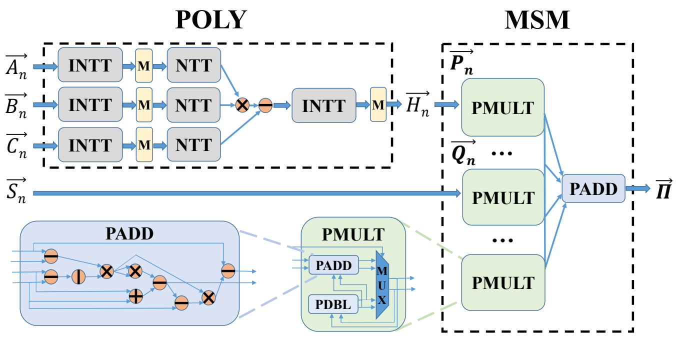


Fig. 2. Prover's POLY and MSM computations for hardware acceleration.

Meanwhile, various random parameters are set up, including the proving keys. With the prover's secret witness, the constraint system, the proving keys, and other parameters, the pre-processing phase subsequently outputs two sets of data (Figure 1), which are later used in the computation phase.

- **Scalar vectors** $\vec{S_n}, \vec{A_n}, \vec{B_n}, \vec{C_n}$. Each vector includes n λ-bit numbers. The dimension n is determined by the size of the constraint system. Note that n could be extremely large for real-world applications. For example, Zcash has n as large as a few millions [35].
- **Point vectors** $\vec{\mathbf{P}}_n, \vec{\mathbf{Q}}_n$. Each vector includes n points on a pre-determined elliptic curve (EC) [33]. EC is commonly used in cryptographic primitives. It supports several basic operations including *point addition* (PADD), *point double* (PDBL) and *point scalar multiplication* (PMULT). By leveraging the binary representation of the scalar, PMULT can be broken down into a series of PADD and PDBL in the scalar's bit-serial order. Both PADD and PDBL operations contain a bunch of arithmetic operations over a large finite field, as shown in Figure 2. Fast algorithms for EC operations typically use projective coordinates to avoid modular inverse [13]. They also adopt Montgomery representations for basic arithmetic operations over the finite field [40].

With these data, the prover can now generate the proof $\vec{\Pi}$.

This is the most computation-heavy phase, and therefore is our main target for hardware acceleration. It involves large-size number theoretic transforms (NTTs) and complicated EC operations, as illustrated in Figure 2. More specifically, the computation phase mainly includes the following two tasks:

- **Polynomial computation (POLY).** It takes $\vec{A_n}, \vec{B_n}, \vec{C_n}$ as inputs and calculates a resultant scalar vector $\vec{H_n}$, whose elements represent the coefficients of a degree-n polynomial. The state-of-the-art implementations for this part use NTTs and inverse NTTs (INTTs), which are similar to Fast Fourier Transforms (FFTs) but instead on a finite field. It can reduce the complexity of POLY from $O(n^2)$ to $O(n \log n)$. Nevertheless, POLY still needs to do NTTs/INTTs for many times, as shown in Figure 2. And each NTT/INTT also has considerable computation cost, given that n could be quite large (up to millions) and each coefficient is a very wide integer number (e.g., $\lambda = 768$ bits).
- **Multi-scalar multiplication (MSM).** This part includes the calculation of the "vector inner products" between $\vec{S_n}$ and $\vec{\mathbf{Q_n}}$, and between $\vec{H_n}$ (the output of POLY) and $\vec{\mathbf{P_n}}$, respectively. Note that the inner products are performed on EC, i.e., using the PADD and PMULT operations defined above to multiply the scalar vector and the point vector together. MSM is computation-intensive, because the cost of the inner products is proportional to n, and the PADD/PMULT operations on EC are also quite expensive, with arithmetic operations between wide integer numbers on a large finite field.

As Figure 1 shows, the prover's witness, after pre-processed, is used as the input for both POLY and MSM. The output of POLY will be included in the input of MSM. The final proof is the output of MSM composed of several EC points. The proof can be verified by the verifier within a few milliseconds through pairing, a special operation on the EC.

C. Hardware Acceleration Opportunities

As we can see from the workflow in Section II-B, the prover's computations are particularly complicated and require significant compute time. In Zcash [35], the size n of the constraint system is about two million. It takes over 30 seconds to generate a proof for each anonymous transaction. As a result, ordinary users sometimes prefer sending transparent transactions instead to avoid the high cost of generating proofs, which trades off privacy for better performance. In Filecoin [24], the function F is even larger. It contains over 128 million constraints and requires an hour to generate a proof. Actually, these blockchain applications usually use crypto-friendly functions that have well-crafted arithmetic computation flows, which are easier to transfer into smaller constraint systems. For real-world, general-purpose applications such as those in Section II-A, the problem sizes will be even larger, with extremely high computation overheads. This is the primary reason that hinders the wide adoption of ZKP. It is therefore necessary to consider hardware acceleration for ZKP workloads, especially on the prover side.

In the proving process, the pre-processing typically takes less than 5% time [8]. We hence focus mostly on the POLY and MSM computations. The POLY part takes about 30% of the proving time. As shown in Figure 2, it mostly invokes the NTT/INTT modules for seven times. Other computations like multiplications and subtractions only contribute less than 2% time. These large-size NTTs are extremely expensive. Similar to FFTs, NTTs have complicated memory access patterns with different strides in each stage. Moreover, all the arithmetic operations (multiplications, exponentiations, etc.) inside NTTs are performed over a large finite field, making them also compute-intensive. Thus, the main focus of hardware acceleration in POLY is the large-size NTTs/INTTs (Section III).

The MSM part takes about 70% of the proving time, which makes it the most computation-intensive part in proving. It requires many expensive PMULT operations on EC. Though several previous proposals have accelerated a single PMULT [14], [15], [36], [41], MSM additionally requires adding up the PMULT result points, i.e., an inner product. This brings the opportunity to use more efficient algorithms rather than simply duplicating multiple PMULT units. Also, in zk-SNARK, the scalar vectors exhibit certain distributions that we can take advantage of to improve performance. We propose a new hardware framework for MSM which can make full use of the hardware resources (Section IV).

Why not just CPUs/GPUs? The basic operations of both POLY and MSM are arithmetics over large finite fields, which are not friendly to traditional general-purpose computing platforms like CPUs and GPUs. CPUs have insufficient computation throughput and they cannot exploit the parallelism inside these operations well enough. GPUs, on the other hand, have high computation throughput but mostly for floating-point numbers. Moreover, the memory architecture of modern GPUs is also not efficient for POLY and MSM operations. Each thread can only access a very limited software cache (i.e., shared memory) and the irregular global memory access patterns in each component will slow down the operations in GPUs significantly. In contrast, large integer arithmetic operations have been well studied in specialized circuit design. It is also more flexible to generate customized designs for different memory access patterns. Thus, a domain-specific accelerator is more promising to achieve better performance and energy efficiency.

D. Prior Work

Prior work has achieved significant performance improvement for polynomial computations in homomorphic encryption using customized hardware [45], [46]. Accelerating EC operations has also been well studied in the literature of circuit design [14], [15], [36], [41]. However, it is inefficient to directly employ the prior designs for zk-SNARK due to two issues. First, the scale of polynomial computations in zk-SNARK is much larger than those needed in homomorphic encryption. Thus, it induces intensive off-chip memory accesses, which cannot be satisfied in prior design. In addition, the data bitwidth in zk-SNARK is much larger, thus it is inefficient to use large-scale multiplexers to select proper input elements for different butterfly operations like before [45]. Second, directly duplicating EC hardware cannot leverage state-of-the-art algorithm optimizations for zk-SNARK. Besides, the sparsity in scalars may cause a lot of resource underutilization in the pipelines that compute MSM. Detailed discussions are in Section III and Section IV.

A recent work called DIZK has proposed to leverage Apache Spark for distributing the prover's computation to multiple machines [50]. Though it can reduce the latency for the proving process, the primary goal for DIZK is supporting zk-SNARK for super large-scale applications, such as machine learning models. Large cloud computing is inefficient for ordinary-size applications like anonymous payment and privacy-preserving smart contracts due to network latency and computation cost. Therefore DIZK can be regarded as a complementary work to ours, while our design achieves better efficiency for each distributed machine.

Recently, a few approaches in industry try to accelerate the prover with dedicated hardware (GPU [11] or FPGA [5]) by leveraging the parallelism inside zk-SNARK. For example, Coda held a global competition for accelerating the proving process using GPU with high rewards ($100k) [11]. However, the final acceleration result of the competition is even worse than our CPU benchmark (See Section VI for more details). And the FPGA one does not contain a complete end-to-end implementation [5]. In summary, there is still a considerable gap between the existing performance and the requirement in practical usage.

III. Accelerating Polynomial Computation

The POLY part of zk-SNARK mainly consists of multiple NTTs and INTTs. To overcome the design challenges of large-size NTTs, we introduce a recursive NTT algorithm with an optimized overall dataflow. We also design efficient hardware NTT modules to alleviate the off-chip bandwidth and on-chip resource requirements.

A. NTT Computations

The NTT computation $\hat{\mathbf{a}} \overset{\text{def}}{=} \text{NTT}(\mathbf{a})$ is defined on two N-size arrays $\mathbf{a}$ and $\hat{\mathbf{a}}$, with their elements $\hat{a}[i] = \sum_{j=0}^{N-1} a[j]\omega_N^{ij}$. Here $a[j]$ and $\hat{a}[i]$ are λ-bit scalars in a finite field. And ω_N is the Nth root of unity in the same field. All possible exponents of ω_N are called *twiddle factors*, which are constant values for a specific size of N. Since we use off-chip memory to store them, we assume all twiddle factors for all possible Ns are pre-computed. This may only introduce tens of MB storage for N up to several millions. Typical implementations of NTT utilize the property of the twiddle factors to compute the results recursively. The access patterns are similar to the standard FFT algorithms, as shown in Figure 3. In this example, the NTT size is $N = 2^n$. In stage i, two elements with a fixed stride 2^{n-i} perform a butterfly operation and output two elements to the next stage. The overall NTT computations complete in n stages. The different strides in different stages result in

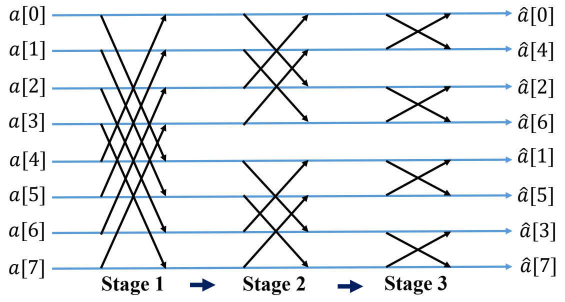


Fig. 3. The data access pattern of NTT (similar to FFT) with size $2^3 = 8$.

complicated data access patterns, which makes it challenging to design an efficient hardware accelerator.

As shown in Figure 3, the output elements on the right side are out-of-order and need to be reordered through an operation called bit-reverse. Alternatively, we can reorder the input elements and generate the outputs in order [21]. If we need to perform multiple NTTs in a sequence, it is possible to properly chain the two styles alternately and eliminate the need for the bit-reverse operations in between.

B. Design Challenges

NTT is an important kernel commonly used in cryptography. As a result, there exist many hardware accelerator designs for NTT. One state-of-the-art NTT hardware design is HEAX [45], which is specialized for homomorphic encryption. However, the POLY computations in zk-SNARK have substantially larger scales than those addressed in HEAX. It requires multiple NTTs of up to a few million elements, with the data width normally more than 256-bit. Such large sizes can hardly be satisfied by any previous NTT hardware design and pose new challenges that must be properly addressed.

First, the total size of zk-SNARK NTT data can be too large to keep on-chip and should be stored in off-chip memory. For example, a million-size NTT with 256-bit data width will need over 64 MB data storage for the input data and the twiddle factors. If we need to access 1024 elements in each cycle from the off-chip memory to feed a 1024-size NTT module, the accelerator has to support at least 2.98 TB/s bandwidth, even with a relatively low 100 MHz frequency. This is unrealistically high in existing systems, let alone that the complicated stride accesses may further reduce the effective bandwidth. Therefore, it is critical to optimize the off-chip data access patterns of the NTT hardware modules to minimize bandwidth requirements and balance between computations and data transfers. In contrast, prior work like HEAX assumes data can be buffered on-chip in most cases and does not specially design for off-chip data accesses [45].

Second, the large bitwidth of NTT elements also requires significant on-chip resources on the computation side. The original HEAX design only works with data no wider than 54-bit. It, therefore, adopts an approach that uses a set of on-chip multiplexers before the computation units to choose the correct input elements for each butterfly operation [45]. If we naively

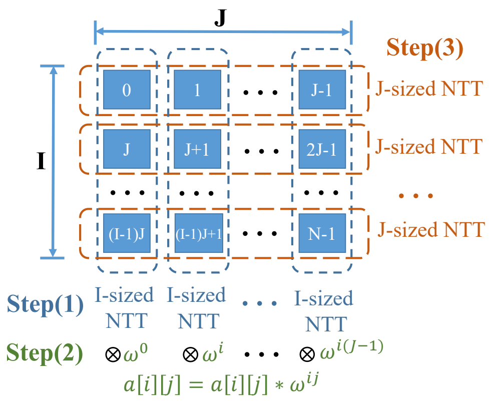


Fig. 4. The recursive NTT algorithm.

scale up the bitwidth beyond 256 as required in zk-SNARK, the area and energy overheads of such multiplexers will increase significantly. Furthermore, the required computation resources for the butterfly operation itself in the NTT module also scale in a super-linear fashion. Both make it inefficient to support large NTTs with high throughput.

C. Recursive NTT Algorithm

To overcome the above challenges, we adopt a parallel NTT algorithm from [21], [49] to recursively decompose a large NTT of arbitrary size (e.g., 1 million) into multiple smaller NTT kernels (e.g., 1024). This allows us to only implement smaller NTT modules, which can fit into the on-chip compute resources and also satisfy the off-chip bandwidth limitation. We then iteratively use the smaller NTT modules to calculate the original large NTT. The hardware NTT module in Section III-D can work with different NTT kernel sizes, therefore supporting flexible decomposition.

We give a high-level overview of the algorithm as shown in Figure 4. A more precise description can refer to the literature [21], [49]. In this example, the large NTT size is $N = I \times J$. We can then decompose the N-size NTT into several I-size and J-size smaller NTTs. For convenience, we represent the original 1D input array $\mathbf{a}$ as a row-major $I \times J$ matrix in Figure 4. We first do an I-size NTT for each of the J columns (step 1). Then we multiply the output with the corresponding twiddle factors (step 2). Next, we do a J-size NTT for each of the I rows (step 3). Finally, we output each element in the column-major order, as the final output 1D array $\hat{\mathbf{a}}$.

D. Bandwidth-Efficient NTT Hardware Module

With the above decomposition, we only need to design a relatively small-size NTT hardware module that works on I and J array elements. Previous work like HEAX [45] implemented such NTT modules following the data access pattern in Figure 3, using a set of on-chip multiplexers to deliver each input element to the corresponding multiplier. However, recall that I and J could still be large (e.g., 1024). Directly fetching these data from off-chip memory in every cycle would result in significant bandwidth consumption, as described in Section III-B. Therefore, we adopt a bandwidth-efficient pipelined architecture. We choose a design similar to [34] as the basic building block. It is a fully pipelined design that reads one input element and produces one output element sequentially in each clock. Instead of using many multiplexers, we use FIFOs with different depths to deal with the different strides in each stage.

Figure 5 shows the simplified design for a 1024-size NTT pipeline module. It contains 10 stages. Each stage has an NTT core that does the butterfly operation between two elements with a certain stride, as in Figure 3, and generates two new elements for the next stage. The core has a 13-cycle latency for the arithmetic operations inside. The depth of the FIFO in each stage matches the stride needed, i.e., 512 for the first stage, 256 for the second stage, and so on. The pipeline keeps reading one element per cycle from the memory. In the first 512 cycles, the 512 elements are stored in the FIFO in the first stage. In the next 512 cycles, we enable the NTT core, which uses the newly read element and pops the head of the FIFO as its two inputs, with the desired stride 512. In this way, the stride is correctly enforced with a FIFO instead of multiplexers. The NTT core generates two output elements in each cycle, one of which is directly sent to the next stage. The other output needs to be buffered and sent to the next stage at a later point (see the orders in Figure 3). We reuse the FIFO in the first stage for this purpose, as the input elements in the FIFO can be discarded after use. The next stage follows the same behavior but with a different FIFO depth to realize a different stride. The last stage writes the output back to the memory.

With the above design, we reduce the bandwidth needed to only one element read and one element write per cycle. With 256-bit elements and 100 MHz, this is just 5.96 GB/s, much more practical to satisfy than before. Also, we reduce the superlinear multiplexer cost to linear memory cost. Not only the resources scale better now, but also the resource type changes from complex logic units to regular RAM.

The total latency for an N-size NTT includes $13 \log N$ cycles for the $\log N$ stages, and N cycles for buffering the data across all stages. It requires another N cycles to fully process all elements, which can be overlapped with the next NTT kernel if any. If there are t modules, it takes $13 \log N + N + \frac{NT}{t}$ cycles to compute T NTT kernels in parallel.

Supporting INTT. We also need to support INTT in POLY. An INTT module is almost the same as NTT, except that (1) the execution order in the butterfly NTT core is different; (2) the control unit operates in the reversed stage order; and (3) the twiddle factors are inversed. We design one butterfly core for both NTT and INTT with different control logic, but shared computation resources such as the expensive multipliers which are the dominant components. In POLY, NTTs and INTTs are chained together as in Figure 2. Thus we can alternately adopt the two reordering styles of input and output arrays in our modules as described in Section III-A to eliminate the need for the bit-reverse operations.

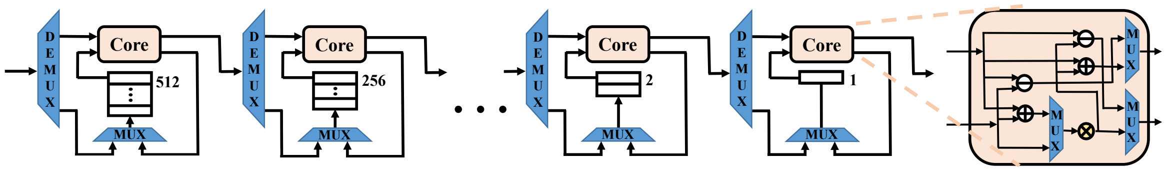


Fig. 5. The architecture of a 1024-size bandwidth-efficient NTT module.

Various-size kernels. Our NTT module can also easily support various-size NTT kernels that are smaller than N. The NTT kernels in POLY are always padded by software to power-of-two sizes. For a power-of-two size smaller than N, we can bypass the previous stages in the module and start from a later stage. For example, a 512-size NTT starts from the second stage. Thus the module can flexibly support different I-size and J-size NTTs after decomposition.

E. Overall NTT Dataflow

We follow the recursive algorithm in Figure 4 to process large-size NTT kernels in a decomposed manner on the small NTT hardware modules in Section III-D. However, the overall data access pattern in each of the steps does not match well with the data layout stored in the off-chip memory. This would result in inefficient large-stride accesses that poorly utilize the available bandwidth. To illustrate this issue, we consider the original input matrix in Figure 4, whose layout in memory is row-major, generated from the 1D array $\mathbf{a}$ (up to a million elements). In step 1, each I-size column NTT kernel needs to process one column of data. This would make J-strided accesses (up to 1024) on the row-major layout. The output data of this step naturally form a column-major matrix. Step 2 is a simple pass of element-wise multiplication. However, in step 3, each J-size row NTT kernel should access the data in a row, again resulting in large strides on the column-major layout. Finally, the output of step 3, which is in row-major after the row NTT kernels, should go through another transpose to be read out in the column order, leading to another round of strided accesses.

To alleviate the problem and make better use of the bandwidth, we effectively block the data to balance between the two choices of layouts (row-major and column-major) and initiate on-chip SRAM buffers to improve input data reuse and aggregate output data before storing back. We also implement multiple NTT modules to process in parallel and to fully utilize the data fetched together from memory each time.

For simplicity, suppose $I = J$ and the original NTT size $N = I \times I$. We implement t NTT modules of size I as shown in Figure 6. Data are still stored in the off-chip memory in a row-major order, as resulted from the original 1D array. First, we fetch t columns together from the off-chip memory and process them in the t NTT modules. Each memory access reads a t-size range of elements, resulting in better sequential access bandwidth. Recall from Section III-D that each NTT module only reads one new input element at each cycle, and outputs one element per cycle after the initial pipeline filling.

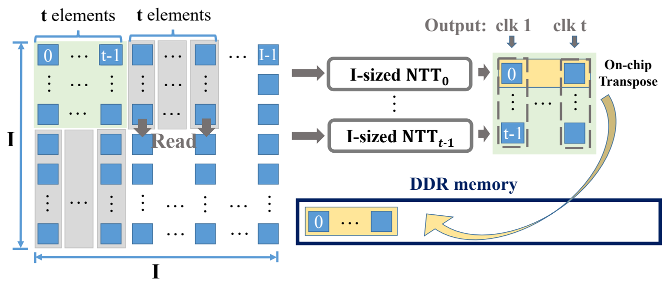


Fig. 6. The overall dataflow of NTT processing. In each cycle, we read t sequential elements from each row of the original 2D array as the input of the t I-size column NTTs. Or equivalently, we read from t columns simultaneously. The marked read happens in the $(I+t)$-th cycle to fetch that sub-row to the NTT modules. The green block shows the buffered output data on-chip to be transposed. The grey elements are currently being processed in the t NTT module pipelines.

We use an on-chip buffer of size $t \times t$ to resolve the data layout issue, by performing a small matrix transpose before writing data back to off-chip memory. In each cycle, the t modules output t elements and write a column in the on-chip buffer. When the buffer is filled up, we write back each row to off-chip memory, resulting in at least t-size access granularity. This allows us to always keep the data in the off-chip memory in row-major formats, while still achieving at least t-size access granularity for high effective bandwidth.

Figure 6 shows the details during the processing. The green block of $t \times t$ elements are already processed and the results are written to the on-chip buffer on the right side. They were pushed into the buffer by columns and popped out to the memory by rows. The gray elements, including the beginning of the second group of t columns, are being processed in the NTT module pipelines. In such a way, we see that the t NTT modules are fully pipelined and well utilized. The pressure on the off-chip bandwidth is also alleviated with our bandwidth-efficient NTT module design.

IV. Accelerating Multi-Scalar Multiplication

In this section, we first introduce the computation task and design challenges for MSM. Then, we present the algorithm and the corresponding architecture to accelerate it.

A. MSM Computations

As illustrated in Section II-B, the MSM computations are defined as $\mathbf{Q} = \sum_{i=1}^{n} k_i \mathbf{P_i}$, where each $\mathbf{P_i}$ is a point on a pre-determined EC and each k_i is a λ-bit scalar on a large finite field. Each pair $k_i\mathbf{P_i}$ is a point scalar multiplication (PMULT), and MSM needs to add up (PADD) these products to get one

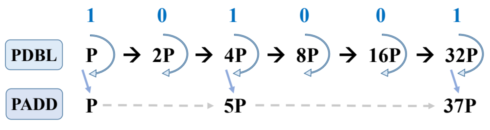


Fig. 7. An example of bit-serial PMUT computation.

final point. zk-SNARK requires several times of MSM with different scalar vectors. One is from the result of POLY (H_n) and the other is from the witness (S_n). Note that the point vectors are known ahead of time as fixed parameters for a certain application problem; only the scalar vectors change according to different witnesses.

As we can see, the most expensive operations in MSM are PMULT and PADD on the EC. Similar to the fast exponentiation algorithm [31], the more expensive PMULT can be decomposed into a series of PADD and PDBL in a bit-serial fashion. An example is shown in Figure 7, where we want to compute $37\mathbf{P}$. We represent 37 in its binary form $(100101)_2$. At each bit position, we execute a PDBL to double the point. If the bit is 1, we add it to the result using a PADD. We can find that PMULT invokes PADD and PDBL sequentially according to each bit of the scalar k_i. Thus, the sparsity of the scalar k_i impacts the overall latency. If the binary form of k_i contains more 1's, then the ith PMULT needs more PADD operations and thus more time.

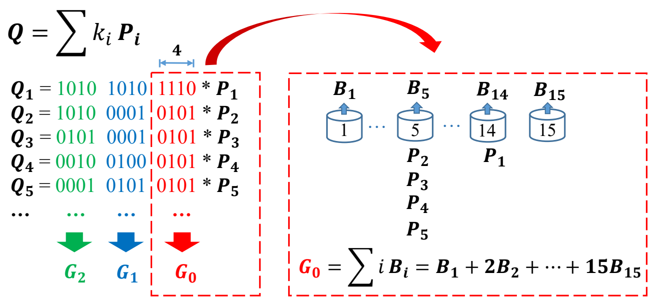
B. Design Challenges

While EC is a commonly used kernel in a wide range of cryptographic applications, most of them only need a single PMULT to encrypt values. Thus, none of the previous accelerators or ASICs has been specially designed for MSM, which involves a large number of PMULT operations whose results are finally accumulated with a PADD. For such a pattern, directly duplicating existing PMULT accelerators is inefficient. Because the computation demands of PADD and PDBL depend on each input scalar, not only the utilization of each PMULT module would be quite low for sparse scalars, but the multiple PMULT modules would also suffer from load imbalance issues, further decreasing the overall performance.

C. Optimized Algorithm and Hardware Module Design

Instead of directly replicating PMULT modules, we adopt the Pippenger algorithm [43] to achieve high resource utilization and better load balancing. We firstly represent the scalar k under radix 2^s, where s is a chosen window size. This is equivalent to dividing the λ-bit scalar k into $\frac{\lambda}{s}$ chunks with s bits each. An example is shown in Figure 8, where $\lambda = 12$ and $s = 4$. Computing Q can be done with the following steps: First, sum up the elements in each chunk i (s-bit wide) to get $\mathbf{G_i}$. Then, sum up $2^{i \times s}\mathbf{G_i}$ to get the final result, with $2^{i \times s}$ as the weights.

In this way, we convert the original computation to a set of smaller sub-tasks of computing $\mathbf{G_i}$. For each sub-task, the Pippenger algorithm groups the elements in the s-bit chunk by the scalars, and put those $\mathbf{P}$'s with the same corresponding



$Q = \sum k_i P_i$
4
Q_1 = 1010 1010 1110 * P_1
Q_2 = 1010 0001 0101 * P_2
Q_3 = 0101 0001 0101 * P_3
Q_4 = 0010 0100 0101 * P_4
Q_5 = 0001 0101 0101 * P_5
...
G_2
G_1
G_0
B_1
B_5
B_{14}
B_{15}
1
5
14
15
P_2 P_3 P_4 P_5
P_1
$G_0 = \sum i\, B_i = B_1 + 2B_2 + \cdots + 15B_{15}$

Fig. 8. Pippenger algorithm.

scalar value into the same bucket, as shown in Figure 8 right side. Since the scalar bitwidth is s, there are $2^s - 1$ different buckets in total. Note that if the scalar is zero, we can directly skip the corresponding points. Then we add up all the points assigned to the same bucket, to get one sum point $\mathbf{B_i}$ per each bucket. Then $\mathbf{G_i}$ can be computed by adding up $\mathbf{B_i}$ weighted by the corresponding scalar i to that bucket. As long as the number of original PMULT operations (i.e., the length of the point and scalar vectors) is much larger than the number of buckets ($2^s - 1$), by doing this we can convert the many expensive PMULT operations into the more lightweight PADD within each bucket. The detailed maths is shown below, where $b_i[j]$ represents the j-th radix-2^s chunk of a_i.

$$\sum_{i=1}^{n} a_i \mathbf{P_i} = \sum_{j=0}^{\frac{\lambda}{s}-1} [\sum_{i=1}^{n} (b_i[j] * \mathbf{P_i})] * 2^{js} = \sum_{j=0}^{\frac{\lambda}{s}-1} \mathbf{G_j} * 2^{js}$$

$$\mathbf{G_j} = \sum_{i=1}^{n} (b_i[j] * \mathbf{P_i}) = \sum_{k=0}^{2^s-1} k * [\sum_{i=1}^{n} (b_i[j] == k) * P_i]$$

With the Pippenger algorithm, MSM becomes PADD-intensive. We design efficient PADD modules. The PADD module is heavily pipelined with 74 stages for expensive arithmetic modular operations such as modular multiply. Since the datapath of PADD is deterministic, we alleviate resource underutilization and load imbalance issues. The remaining few PDBL operations when summing up $i\mathbf{B_i}$ and $2^{i \times s}\mathbf{G_i}$ have only negligible cost, less than 0.1% in our evaluation.

D. Overall Architecture

While we convert expensive PMULT into cheaper PADD operations, the overall architecture still faces a few challenges. First, the group-by phase requires an efficient implementation, especially considering that the size of the MSM (i.e., the length of the scalar and point vectors) could be very large, up to a few millions. The control logic is also non-trivial. Second, while each PADD operation is deterministic, the number of points assigned to each bucket, and hence the number of PADD operations needed, can be skewed. The workloads between buckets can therefore still be possibly imbalanced.

To solve the above problems, we propose a novel architecture for the Pippenger algorithm. First, we divide a large MSM into smaller segments to fit in the on-chip memory.

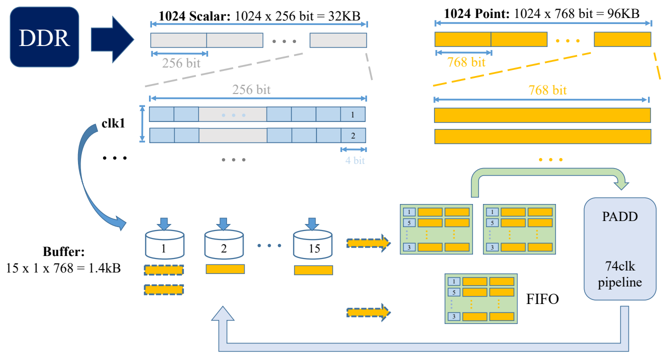


Fig. 9. Overall architecture of the Pippenger algorithm for MSM.

For example, as shown at the top of Figure 9, an MSM with 1 million scalars and points can be divided, and each time we load one segment of 1024 scalars (256-bit each) and points (768-bit each using projective coordinates) to the on-chip global buffer from the off-chip memory.

Then, in each cycle, we read two scalars and two corresponding points from the on-chip buffer. We put the points into different buckets according to the last four bits of the corresponding scalars. The depth for the bucket buffer is only one. Once there are two points that would appear in the same bucket, they will be transferred into a centralized FIFO together with the bucket index as their label, as the green area shown in Figure 9. Each entry of the FIFO contains a 4-bit label (bucket index) and two points from the same bucket waiting to be added together. There are two 15-entry FIFOs prepared for the two scalar-point pairs read in the same cycle.

The entries in the FIFOs are sent to a shared, pipelined PADD module to be processed. When the resultant sum is ready, it should be written back to the corresponding bucket according to the label for further accumulation. However, these results also need another 15-entry FIFO to buffer, in case of conflicts with the already existing data in the destination buckets. Basically, the newly obtained sum can be immediately sent to this FIFO together with the existing data in the bucket for another PADD operation. The PADD module hence can read from three FIFOs in total, two for newly loaded data and one for PADD results. After 512 cycles, the last 4-bit chunks of all 1024 scalars are processed. We then move forward to the next 4 bits and repeat the above workflow.

Overall, our architecture for the Pippenger algorithm uses a centralized and shared PADD module among all buckets, and dynamically dispatches work from these buckets to achieve load balance. Because the PADD module is the performance and area dominant part, sharing it results in a much better resource utilization than having separate private PADD modules in each bucket. Our work dispatch mechanism is also lightweight. We avoid physically sorting the points as typical group-by algorithms require. We mostly rely on a small number of buffers and FIFOs to stash the data to be accumulated. Carefully provisioning the buffer and FIFO sizes allows us to avoid most stalls and achieves high throughput.

E. Exploiting Parallelism and Balancing Loads

The PADD module in our architecture in Section IV-D is clearly a performance bottleneck. Now, we extend the design to use multiple PADD modules in parallel. A straightforward way to make use of those PADDs is to provision multiple PADD for the same set of FIFOs and distribute work from these FIFOs among them. However, this will result in complicated synchronization control logic. Also, increasing the number of PADD modules may lead to more idle cycles when the FIFOs are empty, thus decreasing resource utilization.

We use a different way to balance the workloads among different PADD modules. Notice that we only read 4 bits of a scalar in one round and then read the next 4 bits in the next round. Each round is independent of each other, and thus can be processed in parallel. Therefore, we replicate the entire design in Section IV-D as multiple processing elements (PEs), each with a separate set of buckets, the FIFOs, and a PADD module. For t PEs, we can read $4t$ bits of the scalar each time in one round. Each PE works exactly the same as previously described, and processes its own 4-bit chunk with the same set of points. The control logic is greatly simplified in this way.

We next consider the detailed workload balance among different PEs. The worst situation is that all points in one PE are put into a single bucket. Thus, it has the longest PADD dependency chain, with 1023 PADD operations to get the final result. The best situation is that all points in one round have a uniform distribution and they are put into the 15 buckets evenly, each with 64 or 65 points. This requires $1024 - 15 = 1009$ PADD operations. As the PADD module is shared across all buckets in a PE and is not aware of which bucket the pair of points is from, the end-to-end latency difference between these two cases with similar numbers of required PADD operations is negligible. Therefore, load balance among multiple PEs is well maintained.

As shown in Figure 2, one scalar H_n is from the polynomial computation, and the other S_n is from the expended witness directly. H_n is dense and can be regarded as approximately uniformly distributed, since doing NTT brings uncertainty to the data. Consequently, the possibility of the worst case is extremely low. S_n is very sparse. In fact, more than 99% of the scalars are 0 and 1. This is because the arithmetic circuit usually has a lot of bound checks and range constraints. It uses the binary form of values, and brings 0 and 1 to the expended witness vector. Note that the cases for 0 and 1 can be directly computed without sending into the pipelined acceleration hardware. We process those cases separately. [2]

V. Overall System

The overall architecture of PipeZK is shown in Figure 10. The CPU first expands the witness and transfers the data to the accelerator's DDR memory. Next, the accelerator reads from its memory to execute NTT/INTTs for the POLY phase. After the POLY is done, the MSM subsystem processes the scalar

[2]The cases of 0 and 1 can be filtered when fetching from the scalar and processed in parallel.

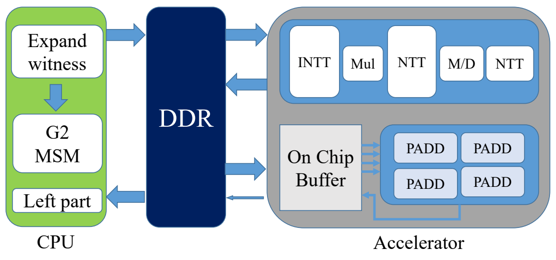


Fig. 10. The overall architecture of PipeZK.

TABLE I
CONFIGURATIONS AND SUPPORTED CURVES ON EACH PLATFORM.

Platforms	Detailed Configurations	Supported Curves
ASIC (ours)	Synopsys DC, UMC 28nm library, DDR4 @2400MHz (4 channels, 2 ranks)	BN-128, BLS12-381, MNT4753
CPU [9]	Intel(R) Xeon(R) Gold 6145 @2.00G Hz, 80 logical cores, 377G RAM	BN-128, MNT4753 [9] BLS12-381 [2]
8GPUs [3]	eight Nvidia GTX 1080 TI cards	BLS12-381
1GPU [6]	single Nvidia GTX 1080 TI card	MNT4753

and point vectors. It outputs the partial sums of B_i from each bucket (see Figure 9), and the CPU deals with the remaining additions, which is less than 0.1% of the execution time.

Note that there are two types of ECs (G1 and G2) in the actual MSM implementation of zk-SNARK [9]. Both G1 and G2 have exactly the same high-level algorithm, so they could have benefited from the same architecture as we introduced in Section IV. The difference is that G2 has different basic units, i.e., the multiplication on G2 needs four modular multiplications whereas G1 only needs one. It needs more resources to implement G2. However, G2 often takes less than 10% of the overall MSM time and the scalar vectors for this part are very sparse. Therefore we move the G2 part to the host CPU, to achieve better trade-off between resources and performance. In summary, the CPU generates the witness and processes the MSM for G2, and the accelerator processes the POLY and the MSM for G1. This results in a heterogeneous system with few interactions. And the computations on both sides can happen in parallel.

VI. EVALUATION

The evaluation consists of two parts. First, we present the microbenchmark results with various input sizes (i.e., constraint system sizes) for the NTT/MSM modules, along with the results of typical workloads shown in Table V. Second, a real-world application, Zcash, is showcased with three end-to-end workloads to demonstrate the practicality of our design and implementation.

A. Experimental Setup

For POLY and MSM, we have a full-stack Verilog implementation, which includes the low-level operations such as PADD, PDBL, and PMULT (with Montgomery optimizations [40] and projective coordinates [13]). We synthesize our design using Synopsys Design Compiler under UMC 28 nm library (details in Table I), and use Ramulator to simulate the performance of off-chip DDR memory. The ASIC-based POLY and MSM modules are integrated along with other modules (such as trusted setup and witness generation) from libsnark [9] running on the host CPU, to derive an end-to-end prototype, as Figure 10 illustrates.

We compare our design (denoted as "ASIC") against the state-of-the-arts, including a single GPU implementation [6] (denoted as "1GPU"), an 8-GPU implementation [3] (denoted as "8GPUs"), and libsnark [9] and bellman [2] on a CPU server (denoted as "CPU"), respectively. Note that due to the limitations of the baseline implementations, in the rest of the paper, we only show corresponding results for supported curves (details in Table I).

TABLE II
LATENCIES (IN SECONDS) AND CORRESPONDING SPEEDUPS FOR NTT MODULE WITH DIFFERENT INPUT SIZES.

Size	λ = 768-bit		λ = 256-bit	
	CPU	ASIC	CPU	ASIC
2^{14}	0.050	0.253 ms (**197.5x**)	0.008	0.076 ms (105.6x)
2^{15}	0.062	0.522 ms (118.8x)	0.015	0.151 ms (99.6x)
2^{16}	0.151	1.045 ms (144.5x)	0.030	0.281 ms (106.6x)
2^{17}	0.284	2.248 ms (126.3x)	0.056	0.604 ms (92.7x)
2^{18}	0.471	5.670 ms (83x)	0.104	1.489 ms (69.8x)
2^{19}	0.845	0.016 (54x)	0.195	4.052 ms (48.1x)
2^{20}	1.368	0.044 (30.4x)	0.333	0.011 (29.3x)

B. Evaluating NTT and MSM with Different Input Sizes

This section presents the microbenchmark results for our NTT and MSM implementations on ASICs. We vary the input size from 2^{14} to 2^{20} to demonstrate the scalability of our design. For both NTT and MSM, we evaluate them with different underlying elliptic curves: BN-128, BLS12-381, and MNT4753, where the bitwidth $\lambda = 256$, 384, and 768, respectively. For BN-128 and MNT4753, we use libsnark [9] on CPUs while bellperson [3] for BLS12-381 on GPUs.[3]

The results for NTT and MSM are shown[4] in Tables II and III. The speedup over the baseline is also attached to each latency number of the ASIC, which equals to the latency ratio between the baseline and the ASIC. Compared to the CPU/GPU implementations, our ASIC design demonstrates a speedup up to 197.5x and 77.7x for NTT and MSM, respectively. Even with increasing input sizes, our implementation still shows superiority.

We carefully tailor the tradeoffs between resource consumption and speed in our ASIC implementations. For the 256-bit curve BN-128, we implement 4 NTT pipelines and 4 PEs for MSM, while use only 1 PE for MSM/NTT in the 768-bit MNT4753 curve. For BLS12-381, we implement 4 NTT

[3]Since the eight-GPU implementation [3] on BLS12-381 is much faster than that of CPU [2], we omit corresponding latency results of CPU for simplicity. However, the one-GPU-card implementation [6] demonstrates weaker performance than that of our 80-core CPU server. Thus, we only list the CPU results for BN-128 and MNT4753 in Tables II and III.

[4]For BLS381 where $\lambda = 384$, the scalar field is still 256-bit. Thus we only compare the performance of 256 and 768-bit for NTT part in Table II.

TABLE III
LATENCIES (IN SECONDS) AND CORRESPONDING SPEEDUPS FOR MSM MODULE WITH DIFFERENT INPUT SIZES.

Size	λ = 768-bit		λ = 384-bit		λ = 256-bit	
	CPU	ASIC	8GPUs	ASIC	CPU	ASIC
2^{14}	0.449	0.012 (39.00x)	0.223	0.004 (**77.70x**)	0.018	0.001 (18.69x)
2^{15}	0.642	0.023 (27.93x)	0.233	0.006 (40.50x)	0.029	0.002 (15.24x)
2^{16}	1.094	0.046 (23.82x)	0.246	0.011 (21.42x)	0.047	0.004 (12.27x)
2^{17}	2.002	0.092 (21.78x)	0.265	0.023 (11.55x)	0.083	0.008 (10.86x)
2^{18}	3.253	0.184 (17.70x)	0.343	0.046 (7.47x)	0.180	0.016 (11.76x)
2^{19}	5.972	0.369 (16.26x)	0.412	0.092 (4.47x)	0.308	0.032 (10.05x)
2^{20}	11.334	0.735 (15.42x)	0.749	0.184 (4.08x)	0.485	0.061 (7.92x)

TABLE IV
RESOURCE UTILIZATION AND POWER CONSUMPTION.

Curve	Modules	Frequency	Area (mm^2)	Dyn Pwr	Lkg Pwr
BN128 (256)	POLY	300 MHz	15.04 (29.63%)	1.36 W	0.68 mW
	MSM	300 MHz	35.34 (69.64%)	5.05 W	0.33 mW
	Interface	600 MHz	0.37 (0.73%)	0.03 W	0.01 mW
	Overall	-	50.75	6.45 W	1.02 mW
BLS381 (384)	POLY	300 MHz	15.04 (30.51%)	1.36 W	0.68 mW
	MSM	300 MHz	33.72 (68.40%)	4.75 W	0.31 mW
	Interface	600 MHz	0.54 (1.10%)	0.04 W	0.01 mW
	Overall	-	49.30	6.15 W	1.00 mW
MNT4753 (768)	POLY	300 MHz	9.69 (18.31%)	0.88 W	0.43 mW
	MSM	300 MHz	42.95 (81.18%)	6.14 W	0.40 mW
	Interface	600 MHz	0.27 (0.51%)	0.02 W	0.01 mW
	Overall	-	52.91	7.04 W	0.84 mW

pipelines (256-bit) and 2 PEs for MSM (384-bit). These are determined by the resource utilization of different curves: the 768-bit modules take more resources than those of the 256-bit curve, especially for the integer modular multiplications (details in Table IV). Large integer modular multiplication plays a dominant role in the resource utilization. We expect the performance will be further improved with more careful resource-efficient design for modular multiplications.

C. Evaluating zk-SNARK Workloads

We also evaluate POLY and MSM of zk-SNARK[5] over typical workloads [8], as shown in Table V.

We present the end-to-end proof time, which includes the time of loading parameters through PCIe, computing POLY and MSM on chip, as well as other processing on CPU. These workloads are compiled with jsnark [8] and executed with libsnark as our backend. Both CPU and GPU baselines [6] are evaluated with the curve MNT4753 where $\lambda = 768$. And as described in Section V, MSM G2 is offloaded to CPU in the GPU baseline and our design. We list the time for POLY, MSM, and MSM G2, respectively. We only provide the overall proof time for "1GPU" without the breakdown due to their heterogeneous architecture with intertwined timings of MSM/POLY on GPU/CPU.

For our ASIC implementation, the latency for proof without G2 (which runs on ASIC) and the latency for MSM G2 (which runs on CPU) are both presented. The final proof time is determined by the maximum latency of the two parts, since they can execute in parallel. However, MSM G2 usually dominates in the overall latency. In summary, Table V shows significant acceleration rates of our implementation over the baselines (50x faster). If we could have additional support for MSM G2 part, the speedup would be even higher.

[5]Note that in the rest of the paper, MSM of zk-SNARK (or MSM for short) denotes the computations of four G1-type MSMs and one G2-type MSM, which differ from "MSM" in Section VI-B that consists of only one G1-type MSM.

D. Evaluating Zcash

Last, we evaluate a real-world industrial application, Zcash, and compare the end-to-end results with a CPU implementation (currently, there are no available GPU implementations for Zcash). The results are shown in Table VI.

There are three kinds of workloads (*sprout, sapling_spend, sapling_output*) in Zcash. To make a shielded transaction, a compound proof is required (i.e., a combination of those workloads). The time for the transaction adds up the proving time for different types of proofs. Other latencies in a transaction such as generating signatures occupy less than 0.5% portion. For the largest workload, *sprout*, we can accelerate the time to generate shielded transactions by 6x. For circuits *sapling_spend and sapling_output* , we can reduce the latency of making shielded transactions over 4x.

We can see that the overall acceleration rate is much lower compared to the acceleration rate of each single module (POLY, MSM). This is because the latencies for MSM G2 and generating witness on CPU ("MSM G2" and "Gen Witness") start to dominate after our acceleration for other parts. As we mentioned in the previous section, MSM G2 can use exactly the same architecture as G1 and get a similar acceleration rate if needed. In addition, generating witness is highly parallelizable with software optimizations, which takes 10% of the overall time and one only needs to accelerate this part for 3 or 4 times to match the overall speedup achieved by our implementation. Therefore, we expect the effort to be technically trivial for ASIC-based MSM G2 and software-optimized witness generating. We leave these for future work.

VII. CONCLUSIONS

Zero-knowledge proof has been introduced for decades and are widely considered as one of the most useful weapons for establishing trust and preserving privacy. However, its limited performance has impeded its wider applications in practice. In this paper, we propose PipeZK, the first architectural effort to significantly accelerate zk-SNARK, the state-of-the-art zero-knowledge proof protocol. We introduce and implement various techniques to efficiently streamline key operations (NTTs, MSMs, etc.) in zk-SNARK. Our empirical results demonstrate considerable speedups compared to state-of-the-art solutions.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their valuable comments. This work is partially supported by National Key R&D Program of China (2020AAA0105200), National Natural Science Foundation of China (62032001 and 62072262).

TABLE V

Results for different workloads (latencies in seconds).

		CPU			1GPU	ASIC					Acceleration Rate		Acceleration Rate (w/o G2)	
Application	Size	POLY	MSM	Proof	Proof	POLY	MSM w/o G2	Proof w/o G2	MSM G2	Proof	ASIC/CPU	ASIC/GPU	ASIC/CPU	ASIC/GPU
AES	16384	0.301	0.835	1.137	1.393	0.002	0.021	0.023	0.097	0.097	11.768	14.420	49.791	61.012
SHA	32768	0.545	0.984	1.529	1.983	0.003	0.027	0.030	0.102	0.102	14.935	19.365	50.330	65.261
RSA-Enc	98304	1.882	3.403	5.290	5.157	0.014	0.080	0.094	1.230	1.230	4.302	4.193	56.297	54.878
RSA-SHA	131072	1.935	3.578	5.514	5.958	0.014	0.105	0.119	0.822	0.822	6.705	7.246	46.481	50.228
Merkle Tree	294912	6.623	8.071	14.695	16.287	0.063	0.226	0.289	2.697	2.697	5.449	6.040	50.869	56.381
Auction	557056	13.875	10.817	24.692	30.573	0.139	0.445	0.585	2.053	2.053	12.025	14.890	42.243	52.306

TABLE VI

Results for Zcash (latencies in seconds).

		CPU				ASIC					Acceleration Rate	
Application	Size	Gen Witness	POLY	MSM	Proof	MSM G2	POLY	MSM w/o G2	Proof w/o G2	Proof	ASIC/CPU	ASIC/CPU w/o G2
Zcash_Sprout	1956950	1.010	3.652	5.147	9.809	0.677	0.076	0.136	0.211	1.687	5.815	8.031
Zcash_Sapling_Spend	98646	0.187	0.441	0.766	1.393	0.167	0.004	0.014	0.018	0.354	3.937	6.817
Zcash_Sapling_Output	7827	0.043	0.107	0.115	0.266	0.034	0.254ms	0.001	0.002	0.077	3.480	5.982

References

[1] "barrywhitehat. roll_up: Scale ethereum with snarks," https://github.com/barryWhiteHat/roll_up/.

[2] "bellman: a crate for building zk-snark circuits," https://github.com/zkcrypto/bellman.

[3] "bellperson: Gpu parallel acceleration for zk-snark," https://github.com/filecoin-project/bellperson.

[4] "Filecoin company," https://filecoin.io/.

[5] "Fpga snark prover targeting the bn128 curve," https://github.com/bsdevlin/fpga_snark_prover.

[6] "Gpu groth16 prover," https://github.com/CodaProtocol/gpu-groth16-prover-3x.

[7] "J.p. morgan quorum," https://www.goquorum.com/.

[8] "jsnark: A java library for building snarks," https://github.com/akosba/jsnark.

[9] "libsnark: a c++ library for zksnark proofs," https://github.com/scipr-lab/libsnark.

[10] "Qed-it," https://qed-it.com/.

[11] "The snark challenge: A global competition to speed up the snark prover," https://coinlist.co/build/coda.

[12] "Zcash company," https://z.cash/.

[13] "Ieee standard specifications for public-key cryptography," *IEEE Std 1363-2000*, pp. 1–228, 2000.

[14] H. Alrimeih and D. Rakhmatov, "Fast and flexible hardware support for ecc over multiple standard prime fields," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 12, pp. 2661–2674, 2014.

[15] B. Baldwin, R. R. Goundar, M. Hamilton, and W. P. Marnane, "Co-Z ECC scalar multiplications for hardware, software and hardware–software co-design on embedded systems," *Journal of Cryptographic Engineering*, vol. 2, no. 4, pp. 221–240, 2012.

[16] E. Ben-Sasson, I. Bentov, A. Chiesa, A. Gabizon, D. Genkin, M. Hamilis, E. Pergament, M. Riabzev, M. Silberstein, E. Tromer *et al.*, "Computational integrity with a public random string from quasi-linear pcps," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2017, pp. 551–579.

[17] E. Ben-Sasson, A. Chiesa, and N. Spooner, "Interactive oracle proofs," in *Theory of Cryptography Conference*. Springer, 2016, pp. 31–60.

[18] N. Bitansky, A. Chiesa, Y. Ishai, O. Paneth, and R. Ostrovsky, "Succinct non-interactive arguments via linear interactive proofs," in *Theory of Cryptography Conference*. Springer, 2013, pp. 315–333.

[19] D. Catalano and D. Fiore, "Vector commitments and their applications," in *International Workshop on Public Key Cryptography*. Springer, 2013, pp. 55–72.

[20] A. Chiesa, Y. Hu, M. Maller, P. Mishra, N. Vesely, and N. Ward, "Marlin: Preprocessing zksnarks with universal and updatable srs," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2020, pp. 738–768.

[21] E. Chu and A. George, *Inside the FFT black box: serial and parallel fast Fourier transform algorithms*. CRC press, 1999.

[22] G. Danezis, C. Fournet, M. Kohlweiss, and B. Parno, "Pinocchio coin: building zerocoin from a succinct pairing-based proof system," in *Proceedings of the First ACM workshop on Language support for privacy-enhancing technologies*. ACM, 2013, pp. 27–30.

[23] A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, and B. Parno, "Cinderella: Turning shabby x. 509 certificates into elegant anonymous credentials with the magic of verifiable computation," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 235–254.

[24] B. Fisch, J. Bonneau, N. Greco, and J. Benet, "Scaling proof-of-replication for filecoin mining," *Benet//Technical report, Stanford University*, 2018.

[25] A. Gabizon, Z. J. Williamson, and O. Ciobotaru, "Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge." *IACR Cryptol. ePrint Arch.*, vol. 2019, p. 953, 2019.

[26] H. S. Galal and A. M. Youssef, "Verifiable sealed-bid auction on the ethereum blockchain," in *International Conference on Financial Cryptography and Data Security*. Springer, 2018, pp. 265–278.

[27] R. Gennaro, C. Gentry, and B. Parno, "Non-interactive verifiable computing: Outsourcing computation to untrusted workers," in *Annual Cryptology Conference*. Springer, 2010, pp. 465–482.

[28] R. Gennaro, C. Gentry, B. Parno, and M. Raykova, "Quadratic span programs and succinct nizks without pcps," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2013, pp. 626–645.

[29] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the forty-first annual ACM symposium on Theory of computing*, 2009, pp. 169–178.

[30] S. Goldwasser, S. Micali, and C. Rackoff, "The knowledge complexity of interactive proof systems," *SIAM Journal on computing*, vol. 18, no. 1, pp. 186–208, 1989.

[31] D. M. Gordon, "A survey of fast exponentiation methods," *Journal of algorithms*, vol. 27, no. 1, pp. 129–146, 1998.

[32] J. Groth, "On the size of pairing-based non-interactive arguments," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2016, pp. 305–326.

[33] D. Hankerson and A. Menezes, *Elliptic curve cryptography*. Springer, 2011.

[34] S. He and M. Torkelson, "A new approach to pipeline fft processor," in *Proceedings of International Conference on Parallel Processing*. IEEE, 1996, pp. 766–770.

[35] D. Hopwood, S. Bowe, T. Hornby, and N. Wilcox, "Zcash protocol specification," *GitHub: San Francisco, CA, USA*, 2016.

[36] K. Javeed and X. Wang, "Low latency flexible fpga implementation of

point multiplication on elliptic curves over gf (p)," *International Journal of Circuit Theory and Applications*, vol. 45, no. 2, pp. 214–228, 2017.

[37] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, "Hawk: The blockchain model of cryptography and privacy-preserving smart contracts," in *2016 IEEE symposium on security and privacy (SP)*. IEEE, 2016, pp. 839–858.

[38] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2010, pp. 1–23.

[39] I. Meckler and E. Shapiro, "Coda: Decentralized cryptocurrency at scale," *O (1) Labs whitepaper. May*, vol. 10, p. 4, 2018.

[40] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of computation*, vol. 44, no. 170, pp. 519–521, 1985.

[41] G. Orlando and C. Paar, "A scalable gf (p) elliptic curve processor architecture for programmable hardware," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2001, pp. 348–363.

[42] B. Parno, J. Howell, C. Gentry, and M. Raykova, "Pinocchio: Nearly practical verifiable computation," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 238–252.

[43] N. Pippenger, "On the evaluation of powers and related problems," in *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*. IEEE, 1976, pp. 258–263.

[44] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," *Journal of the ACM (JACM)*, vol. 56, no. 6, pp. 1–40, 2009.

[45] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "HEAX: An Architecture for Computing on Encrypted Data," in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2020, pp. 1295–1309.

[46] S. S. Roy, F. Turan, K. Jarvinen, F. Vercauteren, and I. Verbauwhede, "Fpga-based high-performance parallel architecture for homomorphic computing on encrypted data," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 387–398.

[47] E. B. Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from bitcoin," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 459–474.

[48] S. Setty, "Spartan: Efficient and general-purpose zksnarks without trusted setup," in *Annual International Cryptology Conference*. Springer, 2020, pp. 704–737.

[49] T.-W. Sze, "Schönhage-strassen algorithm with mapreduce for multiplying terabit integers," in *Proceedings of the 2011 International Workshop on Symbolic-Numeric Computation*, 2012, pp. 54–62.

[50] H. Wu, W. Zheng, A. Chiesa, R. A. Popa, and I. Stoica, "DIZK: A distributed zero knowledge proof system," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 675–692. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/wu

[51] J. Zhang, Z. Fang, Y. Zhang, and D. Song, "Zero knowledge proofs for decision tree predictions and accuracy," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 2039–2053.

[52] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou, "vsql: Verifying arbitrary sql queries over dynamic outsourced databases," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 863–880.

[53] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou, "A zero-knowledge version of vsql." *IACR Cryptol. ePrint Arch.*, vol. 2017, p. 1146, 2017.

[54] Z. Zhao and T.-H. H. Chan, "How to vote privately using bitcoin," in *International Conference on Information and Communications Security*. Springer, 2015, pp. 82–96.