To appear on ISCA 2021
This is a draft version. Camera-ready version is coming soon.

PipeZK: Accelerating Zero-Knowledge Proof with a Pipelined Architecture

***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 support ZKP and make it more practical to use in real-world applications, we propose PipeZK, an efficient pipelined accelerator consisting of two subsystems to handle the aforementioned two intensive compute tasks. 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 28nm, 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 attention 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 that she knows the secret w which satisfies $P(x, w) = y$ without revealing the value of w.

As one of the fundamental primitives 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 [50], online auction [25], anonymous credentials [22], verifiable database outsourcing [48], verifiable machine learning [47], privacy-preserving cryptocurrencies [21], [43], and various smart contracts [35]. More specifically, verifiable outsourcing, as a promising example use case of ZKP, allows a weak client to outsource computations to the powerful cloud and efficiently verify the correctness of the returned results [48], [49]. Another widely deployed application of ZKP is blockchains and cryptocurrencies. The intensive computation tasks can be moved off-chain and each node only needs to efficiently verify the integrity of a more lightweight proof on the critical path [1], [21], [43].

Since its birth [28], 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-SNARKs generate 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-SNARKs in real-world applications, especially in the blockchain community. [1], [4], [7], [10], [12], [36].

Although zk-SNARK proofs are succinct and fast to verify, their generation remains an obstacle in large-scale zk-SNARKs 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 in the system, therefore it is 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 longer, and could be up to a few minutes just for a *single* payment transaction [43].

In this paper, we present PipeZK, an efficient pipelined architecture for accelerating zk-SNARK. PipeZK mainly involves two subsystems, for the polynomial computation with large-sized 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 compute-intensive parts in zk-SNARK. 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-sized NTT computations (up to a million elements) results in sigificant challenges for both off-chip memory accesses and on-chip compute resources, due to the irregular strided access patterns similar to FFT computations, and the large bitwidth (up to 768 bits) of each element. We propose a novel high-level dataflow that decomposes the large NTT kernels into smaller ones recursively, 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 existing

in the vector inner products, and use Pippenger algorithm [40] 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 partitions a large-scale polynomial computation task into small tiles and processes them in a pipeline style. It can achieve high efficiency in both memory bandwidth and logic resource utilization.
- We designed a novel pipelined module for multi-scalar point multiplication on the elliptic curve. It leverages an optimized algorithm and pipelined dataflow to achieve high processing throughput. In addition, it can support various elliptic curves with different data bit-widths.
- We implemented a prototype of the proposed architecture in RTL and synthesized our design under a 28nm ASIC library, and evaluated it as used with CPUs in an end-to-end heterogeneous system. Compared to state-of-the-art approaches, the overall system can achieve 10x speedup for small-sized standard cryptographic benchmarks on average, and 5x for a real-world large-scale application, Zcash [43]. When individually executed, the two subsystems of PipeZK can achieve 20x to 77x speedup, respectively.

II. Background and Motivation

Zero-knowledge proof (ZKP) is a powerful cryptographic primitive that has recently been adopted to real-world applications [21], [22], [25], [35], [43], [47], [48], [50], and drawn a lot of attentions in both academia [16]–[19], [24], [27], [39], [44] and industry [1], [4], [7], [10], [12], [36]. 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 other that her result is valid without accidentally leaking any sensitive information. Many real-world applications can benefit from these properties, including electronic voting [50], online auction [25], anonymous credentials [22], verifiable database outsourcing [48], verifiable machine learning [47], privacy-preserving cryptocurrencies [21], [43], and various smart contracts [35].

A promising example application of ZKP is verifiable outsourcing [26], 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 [48] and machine learning jobs [47]. 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 efficiently check the integrity. The zero-knowledge property allows the prover to make arbitrary statements (i.e., compute functions) about the secret data without worrying about exposing the secret data, therefore naturally support theoretically general-purpose outsourcing computation.

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 computation to update the states, which brings a large overhead with long latency. ZKP enables private decentralized verifiable computation, such that the computation can be moved off-chain, and each node only needs to efficiently check the integrity of a more 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 [36]. 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 [43] and Pinocchio Coin [21] are such examples, where the transaction details including the amount of money and the user addresses are hidden. It offers privacy-preserving trust instead of trust with transparency.

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. [28], there have been significant improvements in the computation efficiency of ZKP to make it more practical. zk-SNARK [30], 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 w. Zero-knowledge means that the proof

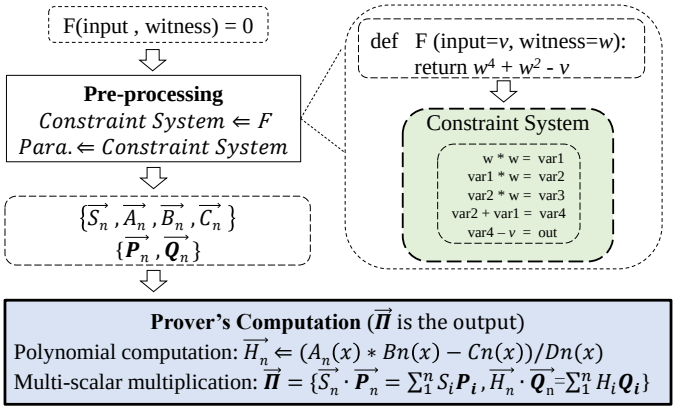

Fig. 1. The workflow of the prover. The illustrated F(v,w) has an constraint system size of five (i.e. $n = 5$).

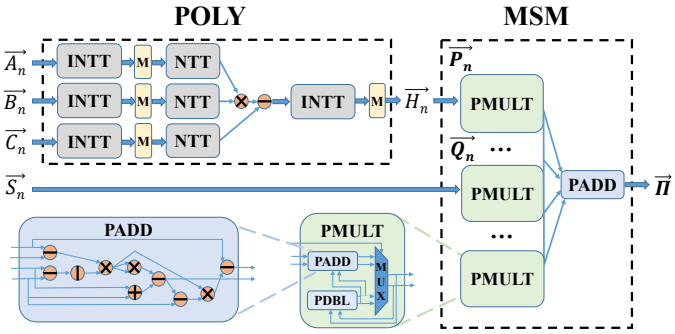

Fig. 2. POLY and MSM computations of the prover, which are our hardware acceleration target.

does not leak any knowledge of the secret witness w. And succinctness means that the size of the proof is small (e.g., of about 128 bytes) and it is also fast to verify (e.g., typically within 2 milliseconds), regardless of how complicated the function F might be.

Unfortunately, although the proof verification can be made fast, generating such 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 [30], which is our target for hardware acceleration.

For specific implementation of zk-SNARK, a security parameter λ is firstly decided to trade off the computation complexity and security strength, by specifying the data width used in the computation. A larger λ provides a stronger security guarantee 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 firstly 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 and the witness. The number of equations in the constraint system is determined by the complexity of the function F, which could be as many as up to millions for real-world applications. 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 would subsequently output a set of data, which will later be used in the computation phase. These involve two parts (Figure 1):

- **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 [33].
- **Point vectors** $\vec{\mathbf{P}_n}, \vec{\mathbf{Q}_n}$. Each vector includes n points on a pre-determined elliptic curve (EC) [31]. EC points are 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 operations on EC typically use projective coordinates to avoid modular inverse [13]. They also adopt Montgomery representations for basic arithmetic operations over the finite field [37].

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-sized 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 input 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

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

the point vector together. MSM is computation-intensive, because the cost of the inner products is proportional to n, and 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-processing, is used as the input for both POLY and MSM. The output of POLY will be included as the input of MSM. The final proof is the output of MSM.

C. Hardware Acceleration Opportunities

As we can see from the workflow in Section II-B, the prover's computation is particularly complicated and requires significant compute time. In Zcash [33], 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 the proof, which trades off privacy for better performance. In Filecoin [23], the function F is even larger. It contains over 128 million constraints and requires an hour to generate the proof. Actually, these blockchain applications usually use crypto-friendly functions that have well-crafted arithmetic computation flows, which are easier to be transferred 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-sized 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 works have been accelerating a single PMULT [14], [15], [34], [38], 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 an insufficient computation throughput and they cannot exploit the parallelism inside these operations well enough. GPUs, on the other hand, have a large computation throughput but mostly for floating 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 size of 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 the literature of circuit design. It's 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 computation in homomorphic encryption using customized hardware [41], [42]. Accelerating EC operations has also been well studied in the literature of circuit design [14], [15], [34], [38]. 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. 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. A detailed discussion is in Section III and Section IV.

A recent work called DIZK has proposed to leverage Spark for distributing the prover's computation to multiple machines [46]. 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-sized applications like anonymous payment and privacy-preserving smart contracts due to network latency and computation cost. Therefore DIZK can be regarded as complementary work to our design, while ours could achieve better efficiency for each distributed machine using our design.

Recently, a few approaches in industry try to accelerate the prover with the 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)

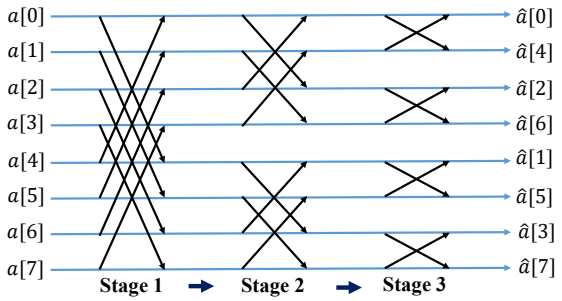

Fig. 3. The data access pattern of NTT (similar to FFT).

and the FPGA one does not contain a complete end to end implementation. 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 the large-sized 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 Computation

The NTT computation $\hat{\mathbf{a}} \stackrel{\text{def}}{=} \text{NTT}(\mathbf{a})$ is defined on two N-sized 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 factor 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} will perform a butterfly operation and output two elements to the next stage. The overall NTT computation completes in n stages. The different strides in different stages result in 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 array and generate the output elements in order [20]. 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 can be found in HEAX [41], which is specialized for homomorphic encryption workloads. However, the POLY computation in zk-SNARK has a substantially larger scale than that addressed in HEAX. It requires multiple NTTs of up to a few million elements, with the data width normally more than 256-bit. Such high scalability can hardly be satisfied by any previous NTT hardware design and poses 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-sized 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-sized NTT module, the accelerator has to support at least 3.2 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 module to minimize the bandwidth requirement and balance between computation and data transfers. In contrast, prior work like HEAX assumes all data can be buffered on-chip in most cases and does not specially design for off-chip data access [41].

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 optimized on-chip dataflow that uses a set of on-chip multiplexers before the computation units to choose the correct input elements for each butterfly operation [41]. If we naively 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 [20], [45] to recursively decompose a large NTT of arbitrary size (e.g., 1M-sized) into multiple smaller NTT kernels (e.g., 1024-sized). This allows us to only implement smaller NTT modules, which can fit into the on-chip computation 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 sizes of NTT kernels, 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 [20], [45]. In this example, the large NTT size is $N = I \times J$. We can then decompose an N-sized NTT into several I-sized and J-sized small NTTs. For convenience, we represent the original 1D input array $\mathbf{a}$ as a row-major $I \times J$

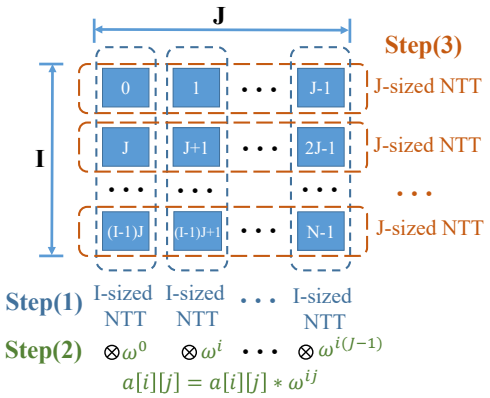

Fig. 4. The recursive NTT algorithm.

matrix in Figure 4. We first do an I-sized 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-sized 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 [41] implemented such NTT modules following the data access pattern in Figure 3, using a set of on-chip multiplexers to deliver each of the elements of the input array to the correct 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 for our NTT module. We choose a design similar to [32] as the basic building block. It is a fully pipelined architecture that reads one input element and outputs one element sequentially in each clock. Instead of using many multiplexers as in HEAX [41], we use FIFOs with different depths to deal with the different strides in each stage.

Figure 5 shows the simplified design for a 1024-sized 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 output 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 6.4 GB/s, much more practical to satisfy than before. Also, we reduce the superlinear multiplexer cost to linear memory cost. Not only the resource utilization is decreased, but also the type of resource changes from complex logic units to regular RAM.

The overall latency for a N-sized NTT module includes the $13 \log N$ 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 multiple such modules in parallel, it takes $13 \log N + N + \frac{NT}{t}$ cycles to compute T NTT kernels with t modules.

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 is the dominant component. 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.

Various-sized kernels. Moreover, our NTT module can easily support various-sized 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-sized NTT starts from the second stage. Thus the module can flexibly support different I-sized and J-sized NTTs after decomposition.

E. Overall NTT Dataflow

We follow the recursive algorithm in Figure 4 to process large-sized NTT kernels in a decomposed manner on 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 and 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-sized 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

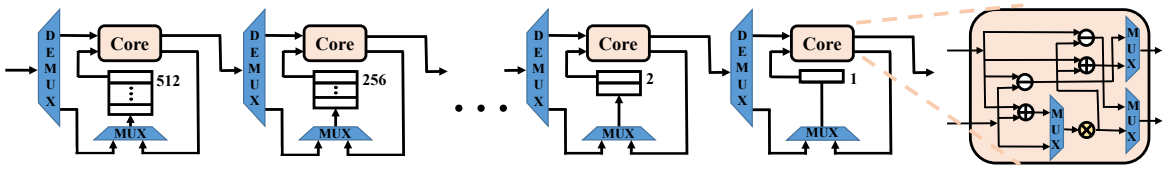

Fig. 5. The architecture of a 1024-sized bandwidth-efficient NTT module.

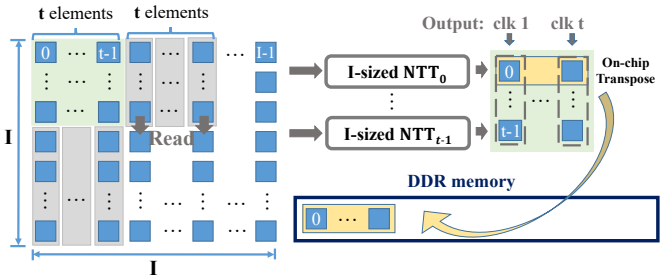

Fig. 6. The overall dataflow of NTT processing.

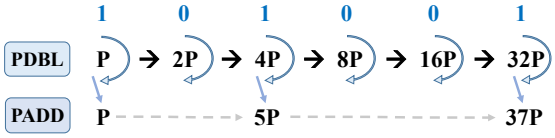

Fig. 7. An example of bit-serial PMUT computation.

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-sized 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. The data are still stored in the off-chip memory in a row-major order without changes. 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-sized 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. 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-sized 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-sized 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 (Figure 5). 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 Computation

As illustrated in Section II-B, the MSM computation is defined as $\mathbf{Q} = \sum_{i=1}^{n} k_i \mathbf{P_i}$, where all $\mathbf{P_i}$'s are points on a pre-determined EC and k_i's are λ-bit scalars 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) the resultant points of all PMULT operations to get one 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; only the scalar vectors change according to different witnesses in different applications.

As we can see, the most expensive operations in MSM are PMULT and PADD on the EC. Similar to the fast exponentiation algorithm [29], 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 $37\mathbf{P}$ where $\mathbf{P}$ is a point on EC. 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.

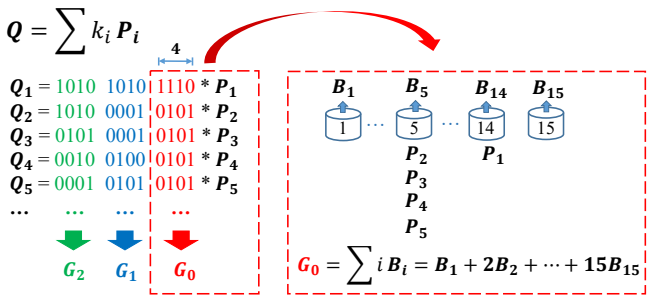

Fig. 8. Pippenger algorithm.

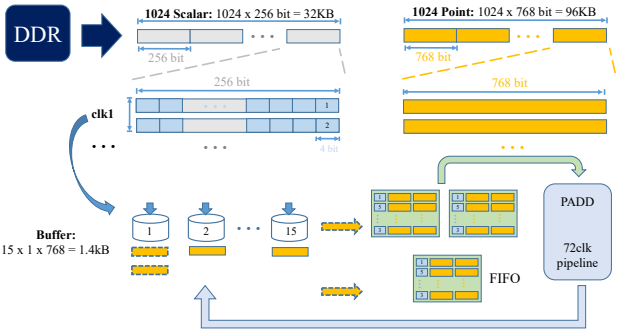

Fig. 9. Overall architecture of the Pippenger algorithm for MSM.

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 have specially designed for MSM, which involves a large number of PMULT operations whose results are finally accumulated with 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. Algorithm Optimization and Hardware Module Design

Instead of directly replicating inefficient PMULT modules, we adopt the Pippenger algorithm [40] to achieve high resource utilization and better load balancing. We firstly represent the scalar k_i under radix 2^s, where s is a chosen window size. This is equivalent to divide the λ-bit scalar k_i into $\frac{\lambda}{s}$ columns 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 column (s-bit wide each) 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 column by to the scalars, and put those $\mathbf{P_i}$ with the same scalar into the same bucket, as shown in Figure 8 right side. Since the bit width of the scalars 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 ($2^s - 1$), in this way we can convert the many expensive PMULT operations into more lightweight PADD within each bucket. The detailed maths is shown below, where $b_i[j]$ represents the j-th digit of a_i radix 2^s.

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

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

With the Pippenger algorithm, the MSM computation becomes PADD-intensive. We design efficient PADD modules. The PADD module is heavily pipelined due to the expensive arithmetic modular operations such as modular multiply. We result in a 72-stage pipeline. Since the datapath of PADD is deterministic, we alleviate the resource underutilization and load imbalance issues. There are still a few PDBL operations when summing up $i \times \mathbf{B_i}$ and $2^{i\times s}G_i$, but their cost is negligible (less than 0.1% in our evaluation).

D. Overall Architecture

While we convert the expensive PMULT into cheaper PADD operations, the overall architecture still faces a few design challenges. First, the group-by operation 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 be possibly imbalanced.

To solve the above problems, we propose a novel architecture for the Pippenger algorithm. Firstly, we divide a large MSM into smaller segments to fit in the on-chip memory. For example, as shown at the top of Figure 9, a MSM with 1M 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 global buffer. We put the points into different buckets according to the last four bits of the corresponding scalar. 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 number as their label, as the green area shown in Figure 9. Each entry of the FIFO contains a 4-bit label (bucket number) 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 FIFO are sent to a shared PADD module to be processed in the long pipeline. When the result sum is ready, it will be written back to the corresponding bucket according to the label. These output results also need another 15-entry FIFO to buffer in case of conflicting with the already existing data in the destination bucket. Basically, the newly obtained sum result can be immediately sent back to this FIFO together with the existing data in the bucket, for another round of PADD operation. Therefore, the PADD module can read from three FIFOs in total, two for newly loaded data and one for PADD results. After 512 cycles, the last 4-bit column of all 1024 scalars has been processed. We then move forward to the next lower 4 bits and repeat the above workflow again.

Overall, our architecture for the Pippenger algorithm uses a centralized and shared PADD module between all buckets, and dynamically dispatch work from the buckets to achieve load balance. Because the PADD module is the performance and area dominant components, sharing them would result 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 FIFOs and distribute the work from the FIFOs among them. However, this will result in a complicated synchronization control logic. Also, increasing the number of PADD modules may lead to more idle cycles when the FIFOs are empty, thus decreasing the utilization of resources.

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 bits with the same set of points. The control logic is greatly simplified in this way.

Considering the workload balance between 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 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, the load balance between 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 a uniform distribution since doing NTT will bring uncertainty to the input. 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 for computation. The above will bring 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. Thus, they are processed separately.

V. Overall System

The overall acceleration system of PipeZK is shown in Figure 10. The CPU first expands the witness and then transfers the data to the accelerator's DDR memory. Next, the accelerator reads from the DDR memory to do NTT/INTTs for the POLY phase. After the POLY is done, the MSM part processes the scalar vectors and point vectors from the DDR memory. In the end, 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 real implementations of zk-SNARK [9]. Both G1 and G2 have exactly the same high-level algorithm thus they can use 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 CPU considering the trade-off between resources and speed. Now, when the accelerator is computing MSM and NTT/INTTs, the CPU is computing the G2 part for MSM at the same time. This is a heterogeneous architecture with few interactions. In summary, the CPU generates the witness and processes the MSM for G2, and the accelerator processes the POLY and the MSM for G1.

VI. Evaluation

The evaluation consists of two parts. First, we present the microbenchmark results with various input sizes (i.e. constraint system sizes) for 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

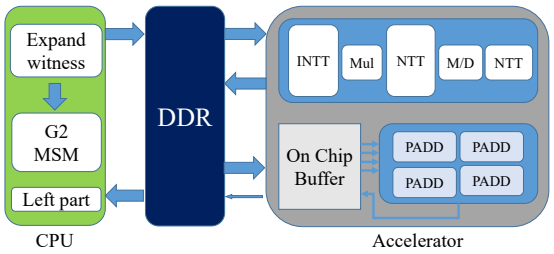

Fig. 10. The overall architecture of PipeZK.

TABLE I
CONFIGURATIONS AND SUPPORTED CURVES OF PLATFORMS

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

evaluated workloads to demonstrate the practicality of our design and implementation.

A. Evaluation Setup

For NTT and MSM, we have a full-stack Verilog implementation, which includes the low-level operations such as PADD, PDBL, and PMULT (with Montgomery optimizations [37] and projective coordinates [13]). We synthesized our design using Synopsys Design Compiler under UMC 28nm library (details in Table I), and ramulator is used to simulate the performance of DDR memory. We integrate our core NTT and MSM modules on ASIC along with other modules (such as trusted setup and witness generation) from libsnark [9] to derive an end-to-end prototype, as Figure 10 illustrates.

We compare our designs (denoted as "ASIC") with 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] (denoted as "CPU") and bellman [2] on a CPU server, respectively. Note that due to the limitations of baseline implementations, in the rest of the paper, we only show corresponding results for supported curves (details in Table I).

TABLE II
LATENCY (IN SECONDS) AND CORRESPONDING SPEEDUP FOR NTT MODULE WITH DIFFERENT INPUT SIZES

Size	λ= 768bit		λ = 256bit	
	CPU	ASIC	CPU	ASIC
2^{14}	0.050	0.004 (12.7X)	0.008	0.329ms (**24.3X**)
2^{15}	0.062	0.008 (7.91X)	0.015	0.667ms (22.8X)
2^{16}	0.151	0.016 (9.66X)	0.030	0.002 (22.8X)
2^{17}	0.284	0.031 (9.08X)	0.056	0.003 (21.45X)
2^{18}	0.471	0.063 (7.54X)	0.104	0.005 (19.65X)
2^{19}	0.845	0.125 (6.76X)	0.195	0.011 (18.5X)
2^{20}	1.368	0.250 (5.47X)	0.333	0.022 (15.75X)

TABLE III
LATENCY (IN SECONDS) AND CORRESPONDING SPEEDUP FOR MSM MODULE WITH DIFFERENT INPUT SIZES

Size	λ = 768bit		λ = 384bit		λ = 256bit	
	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	2.206 (15.42X)	0.749	0.184 (4.08X)	0.485	0.061 (7.92X)

TABLE IV
RESOURCE UTILIZATION AND POWER OF ASIC

Curve	Modules	Frequency	Area(mm2)	DP(W)	LP(mW)
BN128 (256)	POLY	300MHz	15.04 (29.63%)	1.36	0.68
	MSM	300MHz	35.34 (69.64%)	5.05	0.33
	Interface	600MHz	0.37 (0.73%)	0.03	0.01
	Overall	-	50.75	6.45	1.02
BLS381 (384)	POLY	300MHz	15.04 (30.51%)	1.36	0.68
	MSM	300MHz	33.72 (68.40%)	4.75	0.31
	Interface	600MHz	0.54 (1.10%)	0.04	0.01
	Overall	-	49.30	6.15	1.00
MNT4753 (768)	POLY	300MHz	9.69 (18.31%)	0.88	0.43
	MSM	300MHz	42.95 (81.18%)	6.14	0.40
	Interface	600MHz	0.27 (0.51%)	0.02	500uW
	Overall	-	52.91	7.04	0.84

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, BLS-381, and MNT4753, where security parameter $\lambda = 256, 384$ and 768, respectively. For BN-128 and MNT4753, we use libsnark [9] on CPUs while bellperson [3] for BLS-381 on GPUs.[2]

The results for NTT and MSM are shown[3] in Table II and Table III. The speedup number w.r.t. the baseline is also attached in each latency number of the ASIC, which equals the latency ratio between the baseline and the ASIC. To conclude, compared to the CPU/GPU implementation, our ASIC implementation demonstrates a speedup up to 24.3x and 77.7x for the NTT and MSM, respectively. And with the increasing input size, our implementation still shows superiority.

We achieve the superiority by tailoring the trade-off between ASIC resources and speed. For the 256bit curve BN-128, we implement 4 PEs for MSM and 4 NTT pipelines while only 1 PE for MSM/NTT in the MNT4753 (768bit curve). For BLS-381, we implement 2 PEs for MSM (384bit) and 4 NTT pipelines (256bit). This is determined by the resource utilization of different sizes of curves: the basic module for

[2]Since the eight-GPU implementation [3] on BLS-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 CPU server (80-core). Thus, we only list the CPU results for BN-128 and MNT4753 in Table II and Table III.

[3]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 V
EVALUATION FOR DIFFERENT WORKLOADS

		CPU			1GPU	ASIC					Acceleration Rate	
Application	Size	POLY	MSM	Proof	Proof	POLY	MSM (w/o G2)	Proof (w/o G2)	MSM G2	Proof	ASIC/CPU	ASIC/GPU
AES	16384	0.301	0.835	1.137	1.393	0.018	0.021	0.039	0.097	0.097	11.768	14.420
SHA	32768	0.545	0.984	1.529	1.983	0.036	0.027	0.063	0.102	0.102	14.935	19.365
RSA-Enc	98304	1.882	3.403	5.290	5.157	0.146	0.080	0.229	1.230	1.230	4.302	4.193
RSA-SHA	131072	1.935	3.578	5.514	5.958	0.146	0.105	0.250	0.822	0.822	6.705	7.246
Merkle Tree	294912	6.623	8.071	14.695	16.287	0.584	0.226	0.810	2.697	2.697	5.449	6.040
Auction	557056	13.875	10.817	24.692	30.573	1.167	0.445	1.612	2.053	2.053	12.025	14.890

TABLE VI
EVALUATION FOR ZCASH

		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	0.295	0.136	0.431	1.687	5.815	6.809
Zcash_Sapling_Spend	98646	0.187	0.441	0.766	1.393	0.167	0.018	0.014	0.032	0.354	3.937	6.368
Zcash_Sapling_Output	7827	0.043	0.107	0.115	0.266	0.034	0.001	0.001	0.003	0.077	3.480	5.863

768bit takes more resources than that of the 256bit curve, especially for the integer modular multiplications (details in Table IV). Large integer modular multiplication plays a dominant part in the resource utilization, the performance will be largely improved with careful resource-efficient design for the modular multiplications.

C. Evaluating POLY and MSM with Workloads

We also evaluate POLY and MSM of zk-SNARK[4] over typical workloads [8], which is shown in Table V. We also list the proof time, which sums up the MSM and the POLY latency. These workloads are compiled with jsnark [8] and executed with libsnark as our backend. Both CPU and GPU baseline [6] are evaluated with the curve MNT4753 where $\lambda = 768$. And as addressed in Section V, MSM G2 is offloaded to CPU either in the GPU baseline and our design. We list the time for POLY, MSM, and MSM G2, respectively. We only provide the proof time for [6] due to the intertwined timings of MSM/POLY on GPU/CPU and the lack of detailed information with GPU IDE tools.

For our implementation, the latency for proof without G2 (which runs on ASIC) and the latency for MSM G2 (which runs on CPU) are both presented. We can see that in most workloads, the latencies of the two parts can be almost overlapped. The final proof time for our design is determined by the maximum latency of the two parts since the two parts can execute simultaneously. We can find that Table V shows a considerable acceleration of our implementation over the baselines.

D. Evaluation for Zcash

Last, we evaluate a real-world application in industry, 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.

[4]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.

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 is adding up the proving time for different types of proofs. Other latency in a transaction such as generating signatures occupies 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 also find that the accelerating rate is much lower compared to the acceleration rate of single module (NTT, MSM) because the latency for G2 typed elliptic curve and generating witness on CPU dominate after our acceleration for other parts ("MSM G2" and "Gen_Witness"). As we mentioned in the previous section, G2 typed elliptic curve 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. CONCLUSION

Zero-knowledge proof has been introduced for decades and widely considered as one of the most useful weapons for trust/privacy. However, the limited performance of ZKP 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 ZKP protocol. We introduce and implement various techniques to efficiently streamline key operations (NTTs, MSMs, etc.) in zk-SNARK. Our empirical results demonstrate considerable speedup compared to state-of-the-art work.

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/zkcrypto/bellman.

[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] 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.

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

[21] 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.

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

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

[30] 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.

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

[32] 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.

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

[34] 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.

[35] 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.

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

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

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

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

[45] 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.

[46] 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

[47] 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.

[48] 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.

[49] 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.

[50] 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.