SZKP: A Scalable Accelerator Architecture for Zero-Knowledge Proofs

Alhad Daftardar
New York University
United States of America
ajd9396@nyu.edu

Brandon Reagen
New York University
United States of America
bjr5@nyu.edu

Siddharth Garg
New York University
United States of America
sg175@nyu.edu

Abstract

Zero-Knowledge Proofs (ZKPs) are an emergent paradigm in verifiable computing. In the context of applications like cloud computing, ZKPs can be used by a client (called the verifier) to verify the service provider (called the prover) is in fact performing the correct computation based on a public input. A recently prominent variant of ZKPs is zkSNARKs, generating succinct proofs that can be rapidly verified by the end user. However, proof generation itself is very time consuming per transaction. Two key primitives in proof generation are the Number Theoretic Transform (NTT) and Multi-scalar Multiplication (MSM). These primitives are prime candidates for hardware acceleration, and prior works have looked at GPU implementations and custom RTL. However, both algorithms involve complex dataflow patterns – standard NTTs have irregular memory accesses for butterfly computations from stage to stage, and MSMs using Pippenger's algorithm have data-dependent memory accesses for partial sum calculations. We present SZKP, a scalable accelerator framework that is the first ASIC to accelerate *an entire proof on-chip* by leveraging structured dataflows for both NTTs and MSMs. SZKP achieves conservative full-proof speedups of over 400×, 3×, and 12× over CPU, ASIC, and GPU implementations.

CCS Concepts

• **Computer systems organization → Special purpose systems**; • **Security and privacy → Privacy-preserving protocols**; • **Hardware → High-level and register-transfer level synthesis**.

Keywords

Zero-Knowledge Proofs, Cryptography, Hardware Acceleration

ACM Reference Format:
Alhad Daftardar, Brandon Reagen, and Siddharth Garg. 2024. SZKP: A Scalable Accelerator Architecture for Zero-Knowledge Proofs. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '24), October 14–16, 2024, Long Beach, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3656019.3676898

PACT '24, October 14–16, 2024, Long Beach, CA, USA

ACM ISBN 979-8-4007-0631-8/24/10
https://doi.org/10.1145/3656019.3676898

1 Introduction

Zero-knowledge proofs (ZKPs) [16] are cryptographic protocols that offer an enticing capability: a *prover* is able to convince a *verifier* of the truthfulness of a statement, without revealing any further knowledge of how or why that statement is true. For example, one can prove they qualify for a loan *without* revealing bank account details, that one purchased an authentic ticket to enter an event *without* revealing personal details, or that one's vote was counted *without* revealing which candidate was selected. ZKPs are seeing killer applications in Machine-Learning-as-a-service (MLaaS) [36] and crypto-currencies [10] (the latter with extensions to electronic voting [38]). MLaaS providers can use ZKPs to offer services (e.g. recommendation systems) while protecting proprietary model weights that would otherwise be necessary to verify correct operation on clients' inputs. Cryptocurrencies can use ZKPs to enable private transactions while maintaining regulatory compliance [6].

Like other cryptographic protocols, however, ZKPs incur massive computational overheads. For example, the runtime of a single AES ZKP, i.e., where the prover proves access to the secret key without revealing it, on a CPU is on the order of 100s of milliseconds. This is because of how programs (that are to be verified) have to be transformed into specific linear forms to even construct the proof in the first place, followed by intense computational demands of the underlying cryptographic algorithms. The impractically high runtimes of ZKPs have motivated research on both GPU and custom hardware acceleration [24, 25, 37]. Recent works accelerate two of the most computationally demanding kernels of a ZKP: the number theoretic transform (NTT), and the multi-scalar multiplication (MSM). NTTs are analogs of fast Fourier transforms (FFTs) over finite fields. MSMs compute a weighted sum of points on an elliptic curve; since multiplications in elliptical curve cryptography (ECC) are expensive, they leverage Pippenger's algorithm [31] (see Section 3.1.1) to restructure computation to reduce the number of ECC multiplications in favor of ECC additions.

Yet, prior work has stopped short of fully evaluating the benefits of hardware acceleration for ZKPs. First, these works choose to leave key parts of the ZKP algorithm, for instance, so-called "Sparse" MSMs, to software. However, these functions are by no means inexpensive or easy to accelerate. Prior work reveals that Sparse MSMs in software are nearly five times slower than the hardware accelerated components of the ZKP algorithm [37]. Second, prior work has presented single design point solutions, leaving a large design space of area-performance trade-offs unexplored. This applies as well to the individual MSM and NTT modules; some of these implementations have scalability bottlenecks, requiring, for example, on-chip buffers with a large number of read/write ports and/or complex control logic. And finally, prior work has stopped

short of "full-chip" simulations/evaluations, leaving, for example, key questions around off-chip bandwidth unaddressed.

SZKP addresses each of these limitations. First, we accelerate *all* steps in online ZKP proof generation. In the process, we uncover interesting and previously unexplored design choices in integrating modules such as Sparse MSMs on the chip. Second, we perform a detailed evaluation of the ZKP accelerator design space, enabled by new scalable designs for MSM and NTT modules with simple control logic and high-level synthesis (HLS) friendly implementations. Full-chip simulations reveal a rich design space of accelerators from large, highly-parallel 800 mm^2 chips to tiny, sub-50 mm^2 cores, yielding designs that achieve $12-86\times$ speedups over GPU works while using 50% less area and $3-12\times$ speedups over existing ASICs.

2 Background

2.1 Zero Knowledge Proofs

Zero-Knowledge Proofs (ZKPs) allow an entity called the *prover* to demonstrate knowledge of a computation $y = f(w, x)$, on a private witness w and public input x without revealing *anything* about the witness w. For instance, a prover could prove knowledge of a secret key that encrypts public input x to output y, without revealing the key itself. This ZKP, for example, would allow the prover to access a password-protected cloud service without revealing the password to the cloud. Several ZKP protocols have been proposed in literature with different properties. A prominent state-of-art protocol is a zkSNARK, or a *zero-knowledge Succinct Non-interactive ARgument of Knowledge* [9, 17]. zkSNARKs have three main properties: (i) zero-knowledge, meaning the proof reveals no information about the secret witness w; (ii) succinctness, meaning the proof is very small, on the order of 100s of bytes, and (iii) non-interactiveness, meaning only one round of communication is performed where the prover sends the proof to the verifier (as opposed to a protocol where multiple rounds of communication are performed). Given their wide applicability and usage, this paper focuses on hardware accelerators for zkSNARK proof generation.

2.2 zkSNARK Protocol Description

Groth16 [17], shown in Figure 1, is a state-of-art zkSNARK protocol. In this protocol, the prover first generates two keys: (i) a proving key, which is used subsequently to construct the actual proof, and (ii) a verification key, sent to the verifier. This step is time-consuming but is performed offline and only once. Then, for each new input x, the prover has to construct a proof *online*. This online step is also computationally expensive, hence a potential target for hardware acceleration [24, 25, 37]. Proof generation in Groth16 involves two basic operations: MSMs and NTTs. These are described next, followed by the overall dataflow of Groth16.

2.2.1 MSMs. MSMs compute dot products, i.e., $\sum_{i=0}^{n-1} a_i P_i$ where P_i are elements in a cyclic group G, typically, 3-dimensional points on an elliptical curve, and a_i are scalar integers. The individual scalars and coordinates for points can range from 254-753 bits wide depending on the elliptical curve [2]. Compared to a typical dot product operation, multiplication between a scalar and a point, e.g. $a_i P_i$, is computationally expensive and achieved via repeated additions of point P_i. Point additions are themselves costly; depending upon the elliptical curve, a point addition consists of anywhere between 16 and 80 modular multiplications alone.

Groth16 computes two types of MSMs: Sparse and Dense. Sparse MSMs differ from Dense MSMs primarily in the values of the scalars. As the name suggests, Sparse MSM scalars are typically 0 or 1, and usually account for 90%-99% of all scalars. This is because scalar values come from intermediate outputs of program execution, and 0s and 1s relate to the outcomes of branches, conditionals, and other nonlinearities in the program execution that is being verified. This means that roughly half the points can be ignored and half can be directly added once, while the remaining 1-10% of points require more complex processing. In contrast, scalars for Dense MSMs are typically uniformly distributed [2].

In the Groth16 protocol, there are four Sparse MSMs. Three of these are performed on points belonging to an elliptic curve group G1, as is the Dense MSM. The fourth MSM is performed on points from an elliptic curve group G2. The latter operation is distinct in that it involves computations on pairs of points within G2, which is necessary for the pairing-based operations that underpin the Groth16 proofs. Consequently, the G2 MSM is computationally more expensive than G1. Furthermore, one of the Sparse G1 MSMs and the Sparse G2 MSM also exhibit sparsity in their *points*. In ECC, this manifests as the "point at infinity" $\mathcal{O}$, which essentially has the same properties as the number 0 in normal addition.

2.2.2 NTTs and Polynomial Computation. The Number Theoretic Transform is an analog of the Fast Fourier Transform with elements that lie in a field, for instance, integers modulo a prime p. NTTs are used for performing polynomial multiplications, which are equivalent to the convolution of polynomial coefficients. Instead, it is faster to compute the NTT of the coefficients of each polynomial, element-wise multiply (EWM) the NTT outputs, and perform an inverse NTT (INTT). Groth16 involves multiplications, divisions and subtractions over input polynomials implemented using a sequence of NTTs, EWMs and INTTs. In hardware, typically the Cooley-Tukey NTT is used, based on the Cooley-Tukey FFT [12]. Several prior works [18–20, 25, 33–35, 37] propose techniques to accelerate the NTT, including two prior works on ZKPs.

2.2.3 Overall Dataflow of Groth16. While we cannot do full justice to Groth16 here, we will highlight some key concepts relevant to hardware acceleration. Groth16, as well as several other zkSNARK protocols, represents the computation f as a rank-1 constrained system (R1CS) that represents the inputs and outputs of each intermediate computation in f via three vectors, a, b and c. These vectors are transformed into polynomials $A(x)$, $B(x)$ and $C(x)$, using which Groth16 first computes:

$$H(x) = \frac{A(x)B(x) - C(x)}{x^N - 1}$$

where N is the NTT length. All three sequences start in the NTT domain, and each undergoes an INTT, an EWM, and then another NTT. Then, $H(x)$ is computed, with the division operation performed as a multiplication by a scalar. $H(x)$ is still in the NTT domain, so it is processed with an INTT and an element-wise multiply to yield the set of scalars h.

The elements of h form the scalars that are used to compute a Dense MSM with points derived from the proving key (computed

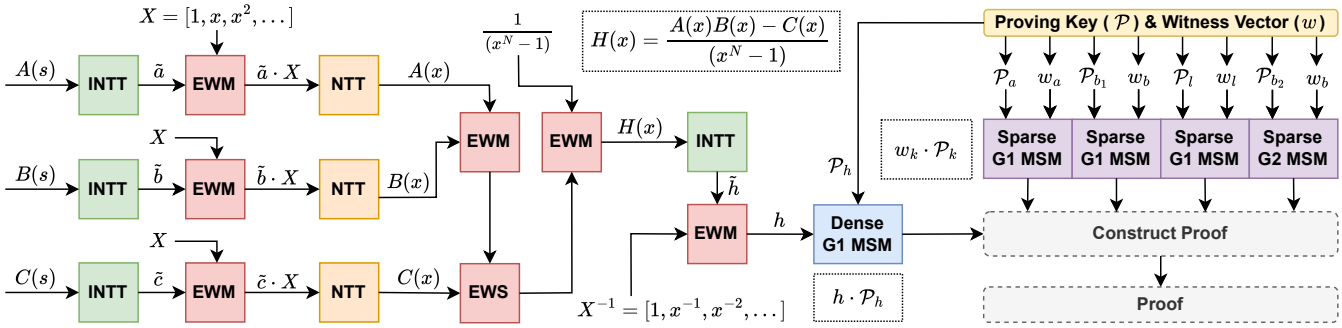

Figure 1: Groth16 dataflow. This protocol involves 7 (I)NTTs and 5 MSMs to construct a lightweight proof for the verifier. In our design, software provides us $A(s), B(s)$, and $C(s)$, as well as the witness vector w and ECC points derived from the offline-generated proving key.

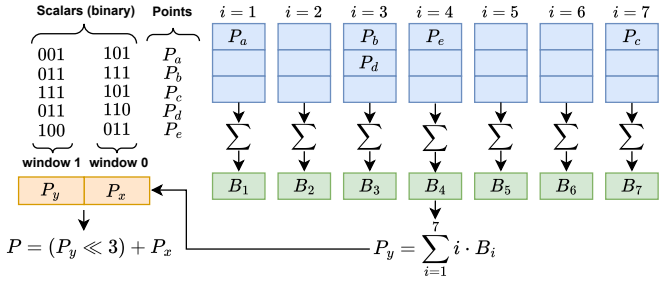

Figure 2: Pippenger's algorithm. This example demonstrates computation of window 1, with P_x already computed for window 0. The final result involves doubling P_y 3 times before adding it with P_x

offline). In parallel, the proving key and witness vectors are also used to compute the four sparse MSMs. Together, the MSM outputs are used to construct the final proof sent to the verifier.

3 The SZKP Architecture

3.1 Dense MSM Architecture

SZKP's Dense MSM architecture uses Pippenger's algorithm [31], commonly deployed in a range of ZKP implementations [24, 25, 37]. We review Pippenger's before describing our architecture.

3.1.1 Pippenger's algorithm for MSMs. MSM computations are prohibitively expensive for high bit-width scalars because of the potentially large number of point additions needed to compute each term in the MSM. The commonly used alternative is Pippenger's algorithm (see Figure 2), where λ-bit scalars are broken up into narrower W-bit *windows*. For instance, a 256-bit scalar might be broken up into 64 4-bit windows.

Then, each scalar within a window maps to one of $B = 2^W - 1$ buckets, omitting the 0^{th} bucket. For example, for 5-bit windows, there are 31 buckets. Points are accumulated into buckets corresponding to their 5-bit scalar values—equivalently, all points corresponding to the same 5-bit scalar are mapped to the same bucket. These points are then added together and multiplied with their corresponding scalar value at the very end. Finally, the results of each bucket are summed to yield a final value for the window.

Once all windows complete their bucket summations, a global reduction is performed by "bit-shifting" each i^{th} window's reduced sum b_i times, where b_i is the offset of window i in the scalar. For this operation, point doubling is used in lieu of point addition, i.e., without incurring point-equality checks inherent to point additions.

Pippenger's algorithm offers two opportunities for parallelism: (1) points in each bucket within a window can be simultaneously added; and (2) multiplications across windows are fully independent until the final global reduction. This implies that two different windows can compute their point additions entirely in parallel. However, both approaches assume unrestricted and content-based access to scalars and points which is challenging in practice.

Prior works have exploited these opportunities in both GPU and ASIC. However, they encounter heavy preprocessing overheads [25] or are not scalable because of how they handle data on-chip and because they offload Sparse MSM compute to CPU [37].

SZKP relies on a PE-array to exploit the high degree of parallelism inherent in Pippenger's algorithm. Each PE handles the computation for one W-bit window of scalars at a time, reading scalars and points from pre-loaded on-chip scalar and point buffers. We first describe the architecture of a single PE, and then how SZKP handles multiple PEs on the chip.

3.1.2 Single PE Design. Figure 3 shows a pipeline diagram of a single PE. As noted, in Pippenger's algorithm, points are fetched into $B = 2^W - 1$ buckets and each bucket independently accumulates points mapped to it. In theory, therefore, this work can be parallelized using B parallel point adders (PADDs). Unfortunately, point addition is *slow*; consistent with prior work, the PADDs we synthesized have latencies in the tens of clock cycles (see Table 1) and dominate chip area. Therefore, we allocate a single, fully-pipelined PADD to each PE. Prior work does the same but requires complex control logic to maximize PADD utilization. We show a much simpler scheme that can achieve equal or better utilization.

Storing Point Addresses: Each MSM PE has $B = 2^W - 1$ buckets, each with a queue of depth D implemented as a FIFO. Each bucket also has a *single* bucket accumulation register, which stores partially accumulated values of points mapped to that bucket. In each cycle, 2 scalars are fetched from the scalar memory to index the buckets, but instead of pushing *points*, we push *addresses* into the respective

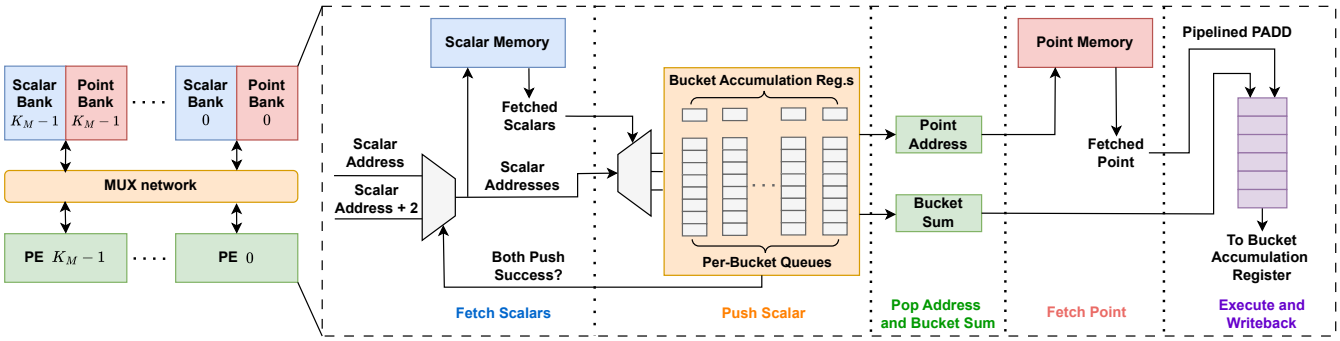

Figure 3: Pipeline architecture for a single Dense MSM. Buckets store point addresses. Queue selection policy can be RR, Max-*r*, or LQ. Writebacks always succeed. Each PE reads from one bank of scalars and points at a time, avoiding memory contention.

buckets' queues. Points are typically large, ranging from 762-2259 bits in total, depending upon the elliptical curve, but addresses to a typical 16K point buffer are only 14 bits. Thus, we can provision deep queues, thus ensuring the PADD always has a sufficient supply of operands to compute from and minimizing stalls to the PADD. Additionally, the control logic in the fetch stage is simple because we always fetch consecutive scalars unless we stall because a bucket's queue is full. Prior work [37] has a complex fetch mechanism that must fetch the points themselves into either a bucket register or one of two FIFOs, having to also face contention with writebacks to the bucket register after a sum is computed from the PADD.

Round-Robin Scheduling: How can we ensure the PADD is maximally utilized? We start with the simplest implementation and then iteratively refine it. As scalars are fetched, in parallel, the PADD iterates over buckets in *round-robin* order. If a bucket's queue is non-empty, it pops an address and uses it to fetch the corresponding point from the point buffer and add it to the value in the bucket's accumulation register. If the bucket's queue is empty, it inserts a bubble into the PADD and moves on to the next bucket. Interestingly, we find that as long as the number of buckets, B, is greater than the PADD latency, t_{add}, even this simple design has more than 85% utilization (see Figure 4). The reason is two-fold: (1) by the time the PADD returns to a bucket, the previous add issued from that bucket has completed, ensuring that a new add can be issued unless the bucket's queue is empty; and (2) scalar values in dense MSMs tend to be uniformly distributed; thus the probability of the bucket's queue being non-empty in $t_{add} > B$ clock cycles is high. We introduce two further optimizations to this architecture.

Longest Queue Policy: To further increase utilization, we replace round-robin with a longest available queue (LQ) dispatch policy. LQ uses simple tournament-style logic to schedule the next add operation from the deepest buffer with a valid accumulated value. As shown in Figure 4, LQ achieves 97 – 99% utilization for 5- to 8-bit windows. We also implement a simpler version of LQ that dispatches from the largest of r consecutive buffers (Max-r). For example, Max-8 achieves more than 95% utilization as shown in Figure 4. In contrast, prior work [37] feeds the PADD from one of two FIFOs, each holding two full points. While this keeps the PADD fed, they also require a spillover FIFO that store the results of add operations that could not be written back to a bucket register, necessitating further area for storing points. In this scheme, it is trivial to show that if the spillover FIFO is not prioritized for pushing points, then the system deadlocks, and forward progress cannot be made leading to functional correctness not being satisfied.

3.1.3 Scaling to Multiple PEs. SZKP allocates K_M PEs to exploit parallelism across different scalar windows that operate on the same set of points. In prior work [37], each PE reads from a single point buffer shared by all PEs. Unfortunately, since the PEs operate on different scalar windows, they operate out-of-sync since they stall at different points. To enable PEs to read from different locations in the point buffer, prior work assumes a multi-ported point buffer with two ports per PE, presenting a scalability bottleneck.

In SZKP, we partition the scalar and point buffers into K_M banks, each with 2 read/write ports. Each PE operates on a *different* bank; when each PE has completed operating on the points in its bank, it switches to the next bank and so on. Specifically, in round j, PE i operates on bank $(i + j) \% K_M$. We note that this scheme does introduce synchronization delays since each PE waits for others to finish. However, although the exact execution traces of the PEs differ, their completion times differ negligibly because scalar values for Dense MSMs are uniformly distributed and hence each PE has the same stall probability. In other words, the PEs share the same statistical behavior which averages out over sufficiently many points. The logic for each PE switching to a different memory bank is simple and can be handled by a simple crossbar, thereby avoiding memory contention that would otherwise occur for the same set of points. Additionally, MSM sequences are long, so to minimize the number of off-chip accesses, the PEs collectively compute the MSM on *all* windows (storing per-window partial bucket sums in local on-chip memories) before fetching more points and scalars into the point and scalar memories. This enables us to have relatively low MSM bandwidth costs in spite of the large bit-widths.

Algorithm 1: BucketReduction

```
1 s_r ← O // running sum
2 s_t ← O // total running sum
3 for i = 2^W − 1 to 1; i = i − 1 do
4 |   s_r ← s_r + B_i
5 |   s_t ← s_t + s_r
6 return s_t
```

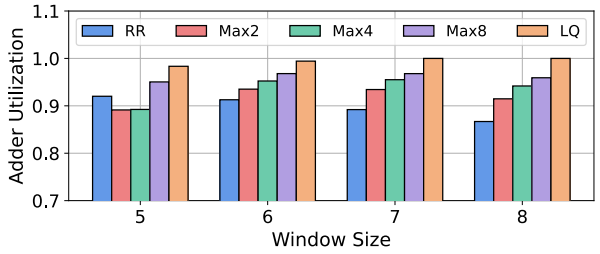

Figure 4: PADD utilization across varying window sizes for round robin (RR), longest queue (LQ) and Max-r. Max-8 and LQ consistently have more than 90% utilization.

Bucket Reduction: After a PE processes all points for a window, the window must compute a bucket reduction, $\sum_{i=1}^{2^W-1} i \cdot B_i$, where B_i is the accumulated value in bucket i. Prior work [37] performs bucket (and window) reductions in software because these typically constitute less than 0.1% of the CPU runtime. However, with the rest of the MSM computation accelerated on-chip, reductions become a larger portion of the runtime if offloaded to CPU. In SZKP, we observe that the recursive Algorithm 1 from [11] admits a hardware-friendly implementation with simple control logic. This algorithm iteratively computes a running sum in one pass with two serial additions per bucket, with a total latency of $(2t_{add})(2^W - 1)$ cycles.

Window Reduction: The final step is to take the sum in each window and compute its offset based on its location in the overall scalar. This is analogous to bit-shifting with integers, but because we are computing on elliptical curve points, we instead perform point-doubling. Note that the most-significant bits (in the most-significant window) have to be doubled the most times. For example, in a 254-bit scalar with 5-bit windows, the highest window has to be scaled by roughly 2^{250}, meaning 250 *serial* point doubling operations. To account for this, we adopt the approach in the CPU implementation [2] where we start computing the accumulated sum with the MSB windows and iteratively compute point doubling towards the LSB windows. With multiple PEs, each PE can handle the doubling responsibilities for 1 window. With K_M PEs, W-bit windows, the latency of window reduction is roughly

$$(K_M \cdot W \cdot t_{dbl} + t_{add})\lambda/W$$

The critical path for each group of K_M PEs is the number of point-doubling operations needed by the PE handling the accumulated result (from the prior group of windows). The maximum number of point doubles done in a group is $W \cdot K_M$ (the bit-width spanned by all PEs). Since the LSB window doesn't need to be doubled within a group of PEs, all PADD units (which support doubling) are utilized.

3.2 Supporting Sparse MSMs

For the majority of the Sparse MSM computation, Sparse MSMs can be viewed as Dense MSMs with only one bucket that corresponds to a scalar value of 1. The Dense MSM architecture loads a point from point memory and adds it to an accumulated value. Instead, in a Sparse MSM core, we first load all points with a scalar of 1 into the point memory, fetch two points at a time, feed them to the PADD, and write back the result to point memory, resulting in near perfect utilization. This process continues until the desired sum is computed. Afterwards, we compute the scalar multiplications for the remaining 1% of scalars using the same Pippenger approach as for Dense MSMs. While prior work implements Sparse G1 MSMs (without any reductions), they offload Sparse G2 MSMs (and all G1 reductions) to CPU [37]; in contrast, SZKP accelerates Sparse G2 MSMs with a dedicated G2 MSM core.

3.2.1 Separate Vs. Shared Hardware. Since both Sparse and Dense MSMs in group G1 use the same PADDs, we explore PADD sharing between the two operations. We note in the overall Groth16 dataflow that the Sparse and Dense MSMs operate in parallel; thus, sharing hardware does cause an artificial dependency between these operations that can increase overall latency. On the other hand, separate hardware for Sparse and Dense MSMs increases chip area. We compare these alternatives in our empirical evaluations.

3.3 Sparse G2 MSM Optimizations

A fully pipelined G1 adder by itself is large, requiring 16+ modular multipliers. But a G2 adder is much larger, requiring 80+ modular multipliers. Because Sparse G2 MSMs are executed independently of other kernels and exhibit sparsity in their *points*, they are not on the critical path. We can significantly reduce modular multiplier counts of Sparse G2 MSMs by pipelining them less. We investigate this by building G2 PADD units with initiation intervals (II) from II=1 to II=4[1]. We also implement this for G1 PADD units and tabulate the multiplier counts in Table 1.

Table 1: Modular multiplier counts for G1 and G2 PADDs for different initiation intervals (IIs) and ECC bit-widths.

PADD	Cycles	II=1	II=2	II=3	II=4
G1 (254b)	30	16	8	7	5
G2 (254b)	55	80	43	36	22
G1 (753b)	38	23	15	9	7
G2 (753b)	67	85	45	38	25

3.4 NTT Architecture

Four-Step NTT: In Groth16 protocols, the NTT sequences are long, ranging from $2^{14} - 2^{20}$ and potentially longer sequences. Because large NTT datapaths are prohibitive in cost, we use the Four-Step algorithm [8] used by prior works [19, 33, 34, 37] to decompose an N-point NTT into a series of $\sqrt{N}$ $\sqrt{N}$-point computations. Thus, we only need to construct an NTT core that computes on up to 2^{10} points to support power-of-2 NTT evaluations up to 2^{20}-point NTTs. Each Four-Step (I)NTT involves transforming the input polynomial into a two-dimensional array, computing the (I)NTT first on the columns, scaling by twiddle factors (different from those used within the (I)NTT), tranposing the matrix, and then performing (I)NTT on the rows. For each of these steps, we fetch inputs from off-chip memory, compute on our NTT core, and write back results to off-chip memory in either a transposed or normal manner, depending on the step being computed.

[1] In HLS terminology, II represents a module's throughput; smaller IIs represent more deeply pipelined modules.

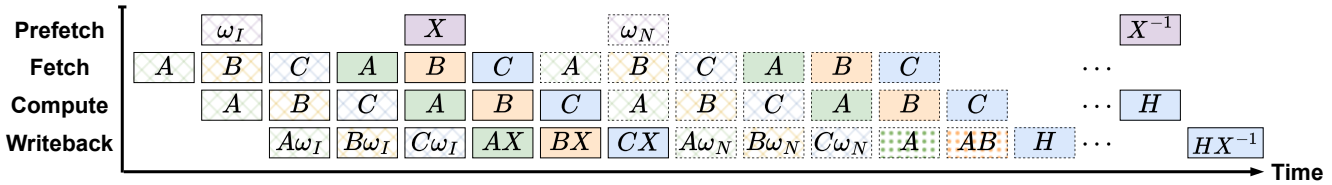

Figure 5: Example Polynomial Computation Pipeline Schedule. This simple schedule assumes an $M \times M$ matrix and M NTT PEs. Slots with solid borders represent an INTT phase, while dashed borders represent an NTT phase. Slots with cross hatches represent column-wise operations, while slots with solid fill represent row-wise operations. Slots with dots represent values that used operands stored on-chip instead of being prefetched. ω_I are INTT twiddles, ω_N are NTT twiddles, and X are generators.

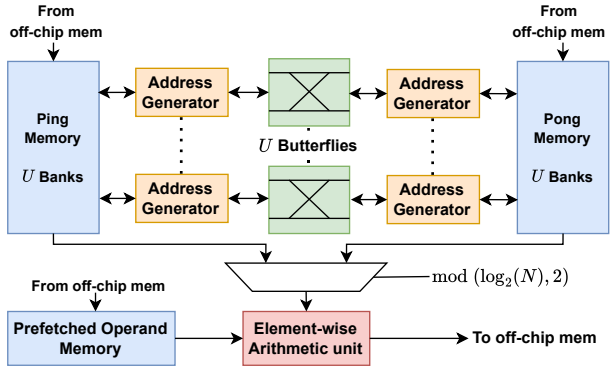

Figure 6: Single NTT PE with simple, static control logic for address generation

Constant-Geometry Topology: Most prior NTT accelerators are based on the Cooley-Tukey algorithm [12]. However, Cooley-Tukey exhibits irregular data flow from stage to stage, resulting in irregular memory access patterns [27, 32]. For this reason, HLS tools struggle to synthesize high-performance memory-based Cooley-Tukey NTT implementations. Instead, we find that a constant-geometry implementation based on the Pease and Korn-Lambiotte algorithms [21, 30] synthesizes high-utilization NTT hardware and much faster than Cooley-Tukey implementations.

In constant-geometry NTTs, the read and write addresses for each butterfly unit in a stage remain fixed, allowing for simple muxing between butterflies and memory buffers. We implement our NTT network using U parallel butterflies and a double-buffered, dual-ported ping-pong on-chip memory with U banks. True dual-porting enables full pipelining of reads from ping (pong) buffers, butterfly compute, and writes to pong (ping) buffers in each stage. Our constant-geometry NTT implementations are competitive with state-of-art hand-designed NTT accelerators [20, 27, 33, 37].

Note that manually-coded Cooley-Tukey NTTs (using a memory-based architecture as we do) would be equally performant to our HLS constant-geometry NTT because the irregular memory accesses per stage could be managed by a dedicated controller, thereby allowing high butterfly utilization. However, the logic for this is complex (adding area overheads) and requires special care to avoid memory conflicts as we scale the number of butterflies [32]. Manually-coded NTTs using pipelined architectures [15, 37] for large lengths require dedicated SRAM delay buffers (and associated controllers) for each stage. These too add non-trivial overheads in area and interconnect. N-point constant-geometry NTTs have simpler control logic, and in turn, are faster to design and prototype with HLS tools. However, they are out-of-place and require $2N$ words for storing intermediate computations. As we show later, MSMs are typically area-dominant in SZKP, so the out-of-place memory costs are worth the savings in design complexity.

Operands: In the Four-Step (I)NTT, the output of column-wise (I)NTTs are multiplied by twiddle factors before being written back to memory. INTT outputs are scaled by a series of multiplicative generators on the finite field (the vector X in Figure 1), $B(x)$ is multiplied with $A(x)$, $C(x)$ is subtracted from $A(x)B(x)$ and $H(x)$ is scaled by inverse generators (X^{-1}) on the finite field. For post-(I)NTT scaling/subtraction, we call these twiddle factors, generators, and intermediates *operands*. Operands are the same length as the (I)NTT, and we prefetch them from off-chip memory while (I)NTTs are computed by the butterflies. We then pipeline the element-wise operand arithmetic with writeback to off-chip memory.

We can significantly reduce the prefetch bandwidth for the twiddle factors and generators because they can be computed on-the-fly given initial columns of the twiddle and generator matrices. Then, by interleaving the execution of the A, B, and C (I)NTTs, we maximize operand reuse and generate them on-the-fly in our element-wise arithmetic unit, eliminating most of the prefetch bandwidth and masking most of the matrix transpose latencies. The operands that cannot be generated on-the-fly are $A(x)$ for multiplication with $B(x)$ and $A(x)B(x)$ from which $C(x)$ is subtracted; these operands can directly be loaded into the operand memory buffer instead of writing back to off-chip memory.

Finally, the four-step (I)NTT's column/row-wise operations can be performed independently prior to the matrix transpose. This allows for a multi-PE (I)NTT where each PE handles one column/row (I)NTT without the need for complex interconnect for transferring data. This enables easy scaling of the number of PEs.

Figure 5 shows a simplified example of the polynomial computation schedule to highlight the kernel-level pipelining, with each on-chip compute and fetch, prefetch, and writeback from off-chip memory (except where noted) running concurrently. For simplicity, we display the same matrix names (A, B, C, or H) for all intermediate and final results. Here, we can see how interleaved scheduling allows us to complete all column-wise computations before reading in data row-wise. This helps mask serial matrix transpose latencies.

Bit-reversals: Since our NTT unit supports both NTT and INTT, it needs to handle bit-reversals. Our NTT does this by utilizing the chaining property as noted by prior designs [23, 37] to avoid bit-reversals. However, if we assume the input polynomials are

in natural input order, the final INTT step will output scalars in bit-reverse order. Because the points used by the Dense MSM are generated by the proving key during one-time key generation, we assume the key generation software bit-reverses these points for us. Since point addition is commutative, the order in which we compute the MSM does not matter, thus, elimination of bit-reversal overhead still maintains functional correctness.

4 Evaluation Methodology

SZKP is, as far as we know, the first ASIC to accelerate entire zk-SNARK proofs on-chip, including Sparse G1/G2 MSMs and MSM reductions. Figure 7 details the full SZKP architecture. As part of our evaluation, we conduct a comprehensive design space exploration of both individual modules and the full chip. We additionally investigate resource sharing among G1 MSMs as well as the effect of off-chip bandwidth constraints on total proof-generation time.

4.1 Performance Modeling

We used Catapult HLS 2022 to generate the RTL for pipelined field adders and multipliers (we used Montgomery multipliers [29] as in prior work [33, 37]), constant-geometry NTT PEs, and G1 and G2 PADD units with different IIs to study area vs. runtime tradeoffs. We synthesized RTL using Synopsys DC Compiler with a TSMC 22nm technology library and a Synopsys 22nm Memory Compiler for area estimation with a 300 MHz clock to match prior ASICs [37]. We also separately synthesized the longest queue (LQ) dispatch policy and confirmed that it clocks at 1 GHz, much faster than the 300 MHz target.

We used HLS-generated modules to determine PADD and NTT latencies for performance modeling. We developed a cycle-accurate simulator to model the runtime of the Dense MSM pipeline using the LQ dispatch policy. We used this simulator along with analytical models for Sparse MSM computations and bucket/window reductions. For polynomial computation, we used single-NTT latencies to model the total latency and constructed traces of the off-chip memory accesses for inputs, operands, and outputs. We initially assumed unconstrained memory bandwidth to model peak theoretical performance, and later imposed bandwidth constraints for a range of memory technologies. We also constructed power traces to model thermal design power and power density.

4.2 Benchmarks

We evaluate our architecture using workloads from Jsnark [22], a Java based library with Libsnark [2] as its backend. Libsnark is a state-of-the-art CPU implementation of Groth16, supporting several elliptic curves used for zkSNARK computations.

We focus our analysis on the BN128 and MNT4753 curves which both provide 128 bits of security and are commonly used in prior work [24, 25, 37]. The former has an underlying bit-width of 254 for scalars and points, while the latter uses 753 bits. We compare SZKP with CPU using BN128, with a prior ASIC using BN128 and MNT4753, and with prior GPUs using MNT4753. This is done to reflect the range of areas we typically see as SZKP is scaled to higher bit-widths. We first examine the performance of individual modules and then use Jsnark workloads to measure the time to compute full proofs against CPU, ASIC, and GPU.

Table 2: Design Space of SZKP Architecture. We evaluate all combinations of these design knobs.

Module	Design Parameter	Values
MSM	PEs (K_M)	1, 2, 4, 8, 16
MSM	Window Size (W)	5, 6, 7, 8
MSM	Points/Window (PPW)	1K, 2K, 4K, 8K, 16K
MSM	PADD Pipelining (II)	1, 2, 3, 4
NTT	PEs (K_N)	1, 2, 4, 8
NTT	Butterflies (U)	1, 2, 4, 8, 16, 32

5 SZKP Evaluation

We now evaluate the area and performance of a range of different SZKP accelerators using the design space exploration parameters listed in Table 2. In our evaluations, we begin by reporting the area and power of individual components. We then use synthetic data to determine runtime and speedup of SZKP's Dense MSMs and NTTs over CPU, ASIC, and GPU designs.

Next, we report full-chip results. Since prior work did not accelerate Sparse MSMs in hardware, we highlight the impact of design choices including shared vs. separate hardware for G1 MSMs and the benefits of optimizing II for Sparse MSM PADDs. We then identify and analyze several representative zkSNARK designs at different area-performance points, show the impact of bandwidth on accelerator performance, and report full-chip speedups relative to CPU, ASIC, and GPU.

Table 3: Area and Power of Components

Component	Area (mm^2)		Power (W)	
	254b	**753b**	**254b**	**753b**
Modmul	0.30	2.89	0.12	0.78
NTT Butterfly	0.31	2.90	0.13	0.81
G1 PADD II = 1	5.05	67.33	2.14	18.50
G1 PADD II = 2	2.56	43.91	1.08	12.07
G1 PADD II = 3	2.52	26.35	1.06	7.24
G1 PADD II = 4	1.61	20.49	0.67	5.63
G2 PADD II = 4	8.72	73.74	2.72	20.20

5.1 Individual MSM and NTT Unit Evaluations

Table 3 shows the area and power for different components synthesized via HLS. For 254b, HLS generates 30-cycle G1 PADDs and 55-cycle G2 PADDs; for 753b, HLS generates 38-cycle G1 PADDs and 67-cycle G2 PADDs (see Table 1). We estimate power using pessimistic 50% switching activity. Figure 8 plots area vs. runtime Pareto plots for each individual module running the Auction workload on 254b datatypes. We note that although Sparse G2 MSMs consume the most area, they are much faster than and can be computed in parallel with other computations. Thus, in practice, full-chip Pareto designs only instantiate the smallest Sparse G2 units which are still well off the critical path. Of the remaining modules, Dense MSMs consume the most area, followed by NTTs and Sparse G1 MSMs.

5.1.1 SZKP vs. CPU. Table 4 details the speedup of our Dense MSM in isolation and NTT in isolation versus CPU. For fair comparison,

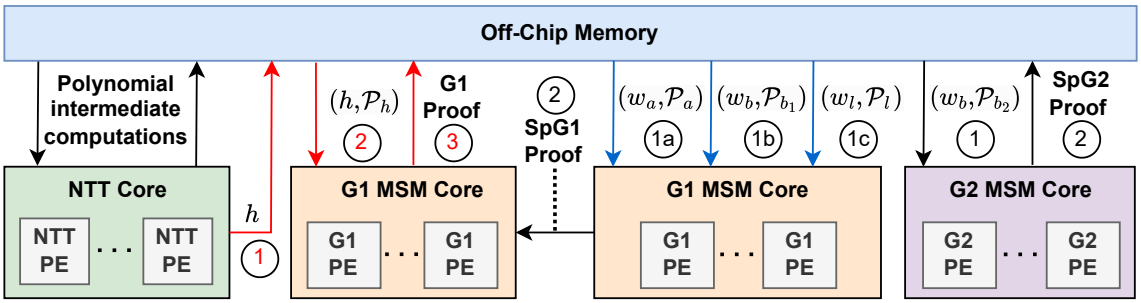

Figure 7: SZKP Chip Architecture. Numbers indicate order of operations. Red arrows and numbers indicate the critical path from polynomial computation through the Dense MSM. Blue arrows indicate serialization of the 3 Sparse MSMs through the G1 core dedicated for Sparse MSMs. The final proof is constructed in two parts, a G1 component and a G2 component, that get written back to off-chip memory

we pick an MSM and NTT design point that is roughly iso-area to a single core on our CPU, an AMD EPYC 7502 32-core processor running either a single Dense MSM or single polynomial computation. The CPU processor consists of four 8-core chiplets, each of die size 74 mm^2 fabricated in 7nm, with a memory bandwidth of 204.8 GB/s [3–5]. We estimate the core area to be 9.25 mm^2, and using a scale factor of 3.6 based on prior work [13, 28], we pick designs close to 30-33 mm^2 in 22nm and estimate a speedup factor of 1.7 to model scaling down to 7nm [13]. Under these constraints, we choose an MSM with 4 PEs, 8-bit windows, and 4096 points per window, and an NTT unit with 4 PEs and 16 butterflies/PE.

5.1.2 SZKP vs. PipeZK. We compare individual module speedups with PipeZK [37], the only known ASIC that accelerates full zk-SNARK proofs, on 254b data in Table 5. For this, we pick an MSM with 4 PEs, 8-bit windows, and 2048 points per window (26 mm^2 vs. PipeZK's 35 mm^2 MSM), and an NTT with 4 PEs and 8 butterflies/PE (18 mm^2 vs. PipeZK's 15 mm^2 NTT). We achieve MSM speedups of nearly 2× at large workload lengths. Our MSM runtimes include bucket reductions and window reductions performed on-chip, while PipeZK offloads reductions to CPU. We also achieve 3-8× NTT speedups. Note that for variable-length architectures, PipeZK's NTT suffers from low butterfly utilization. PipeZK uses a serially-pipelined NTT [15] which uses 1 butterfly per NTT stage. PipeZK's NTT handles up to $2^{10} = 1024$ elements to support a Four-step NTT of maximum length $N = 2^{20}$. However, for workloads of length $N \leq 2^{19}$, where $\sqrt{N} \leq 1024$, one or more stages of the NTT need to be bypassed during the row and/or column step; butterflies in those stages are then underutilized. In other words, except for workloads of maximal supported length, there are always some butterflies underutilized in *each* PE of PipeZK's NTT. This poses a challenge to scalability because supporting longer NTTs risks more unused butterflies on sub-maximal-length workloads. In contrast, SZKP's memory-based NTT uses *all* available butterflies in *each* stage ensuring near-perfect utilization during operation.

5.1.3 SZKP vs. GPU designs. Table 6 shows our speedups over cuZK and GZKP, which both use NVIDIA V100s, using 815 mm^2 in 12nm [1, 24, 25]. For data that both cuZK and GZKP provide, we show speedup over the faster design. We pick Dense MSM and NTT modules based on a roughly 800 mm^2 budget in 22nm. We use scale factors of 2× for area and 1.65× for delay [13] to scale to 12nm. We pick an MSM with 8 PEs, 6-bit windows, and 1024 points per window (278 mm^2 in 12nm), and an NTT with 2 PEs and 4 butterflies/PE (20 mm^2 in 12nm), with Sparse G1 and G2 combined area of 102 mm^2. We choose this design to show that even with 3× and 40× less area than a V100 (and at 300 MHz), we still achieve 15 – 35× MSM speedup and 2.5 – 4× NTT speedup, respectively.

Table 4: Runtime (ms) of kernels vs CPU on 254b

Size	CPU		SZKP-7nm	
	Poly	MSM	Poly	MSM
2^{14}	102	344	0.078 (**1315**×)	0.52 (**662**×)
2^{15}	211	625	0.137 (**1545**×)	0.78 (**804**×)
2^{16}	479	1131	0.235 (**2040**×)	1.30 (**873**×)
2^{17}	982	2204	0.442 (**2223**×)	2.33 (**946**×)
2^{18}	2151	3884	0.807 (**2665**×)	4.40 (**883**×)
2^{19}	4379	7554	1.567 (**2795**×)	8.53 (**885**×)
2^{20}	8865	14834	2.996 (**2959**×)	16.81 (**883**×)

Table 5: Runtime (ms) of kernels vs PipeZK on 254b

Size	PipeZK		SZKP	
	NTT	MSM	NTT	MSM
2^{14}	0.076	1	0.026 (**2.92**×)	0.93 (**1.07**×)
2^{15}	0.151	2	0.048 (**3.15**×)	1.42 (**1.41**×)
2^{16}	0.281	4	0.087 (**3.23**×)	2.40 (**1.66**×)
2^{17}	0.604	8	0.166 (**3.64**×)	4.37 (**1.83**×)
2^{18}	1.489	16	0.318 (**4.68**×)	8.30 (**1.93**×)
2^{19}	4.052	32	0.631 (**6.42**×)	16.17 (**1.98**×)
2^{20}	11	61	1.25 (**8.80**×)	31.90 (**1.91**×)

5.2 Full Chip Evaluations

5.2.1 Shared Vs. Separate G1. As mentioned in section 3.2.1, we can share the G1 MSM core between Sparse and Dense MSMs. We evaluate both topologies, Shared-G1 and Separate-G1, on Jsnark's Auction workload at 254b and plot Pareto curves in Figure 9. We would typically expect separate (shared) hardware to dominate in high-performance, high-area (low-performance, low-area) configurations.

Surprisingly, from these plots we see that the shared topology does not yield significant performance benefits. In fact, the shared-G1 topology's Pareto frontier essentially tracks the separate-G1

Table 6: Runtime (ms) of kernels vs GPU on 753b

Size	cuZK	GZKP		SZKP-12nm	
	MSM	NTT	MSM	NTT	MSM
2^{14}	–	0.15	20	0.04 (**4.1**×)	0.86 (**23.3**×)
2^{16}	–	0.49	50	0.18 (**2.8**×)	2.83 (**17.7**×)
2^{18}	–	1.91	160	0.74 (**2.6**×)	10.69 (**15.0**×)
2^{19}	732	–	–	1.49	21.18 (**34.6**×)
2^{20}	1163	7.46	600	2.99 (**2.5**×)	42.15 (**14.2**×)
2^{21}	1960	–	–	5.98	84.09 (**23.3**×)
2^{22}	3608	33.67	2660	11.97 (**2.8**×)	167.98 (**15.8**×)
2^{23}	6635	–	–	23.96	335.75 (**19.8**×)
2^{24}	–	141.4	11300	47.92 (**3.0**×)	671.29 (**16.8**×)
2^{26}	–	602.53	40700	191.73 (**3.1**×)	2684.56 (**15.2**×)

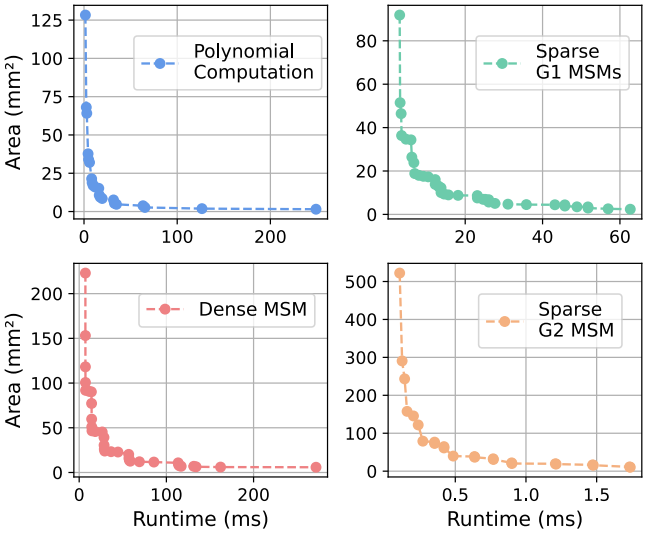

Figure 8: Pareto Curves for individual SZKP modules. Sparse G2 MSMs are the largest module by area but contribute to less than 0.5% of total proof generation time

topology's frontier, except for high-performance (and high-area) where only Separate-G1 yields valid designs. The reason for this is that Sparse G1 MSMs are not on the critical path in low-performance designs; therefore, even though Separate-G1 requires extra PADDs, they are small and slow and add minimal area overhead.

However, as we move to higher performance designs, Sparse G1 MSMs improve much slower than Dense MSMs and *start appearing on the critical path*. Thus, having separately optimized Sparse G1 cores enables efficient resource allocation toward Dense MSMs.

5.2.2 Optimizing Pipeline Depth. How much does pipelining actually benefit Sparse MSMs? Figure 10 compares the Pareto frontiers for 254b SZKP designs that have all Sparse MSMs fully pipelined versus II-optimized designs that save area. As we can see, there is virtually no performance penalty when we reduce pipelining. We highlight three designs, denoted as HP for high-performance, MP for medium-performance, and LP for low-performance. HP is chosen as the most performant and area intensive design, LP is chosen to be iso-area with our CPU (when scaled to 7nm), and MP

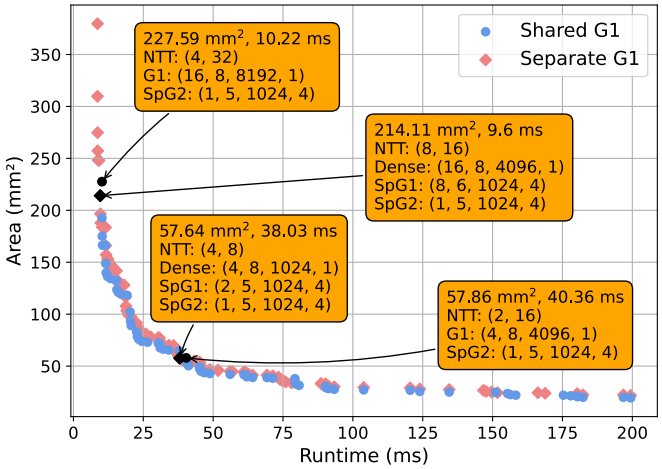

Figure 9: Comparison of Shared-G1 vs. Separate-G1 Pareto-optimal designs for Auction. The NTT designs are denoted as (K_N, U), and MSMs are denoted as (K_M, W, PPW, II). For shared designs, we denote G1 for both Dense and Sparse MSM

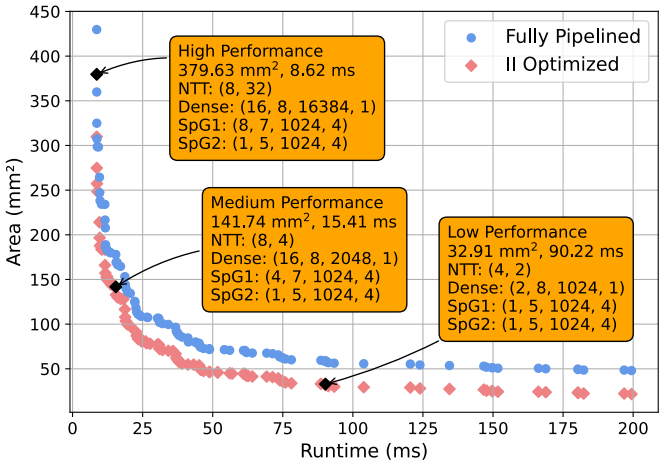

Figure 10: Effect of Pipelining on Area. HP, MP, and LP designs are highlighted.

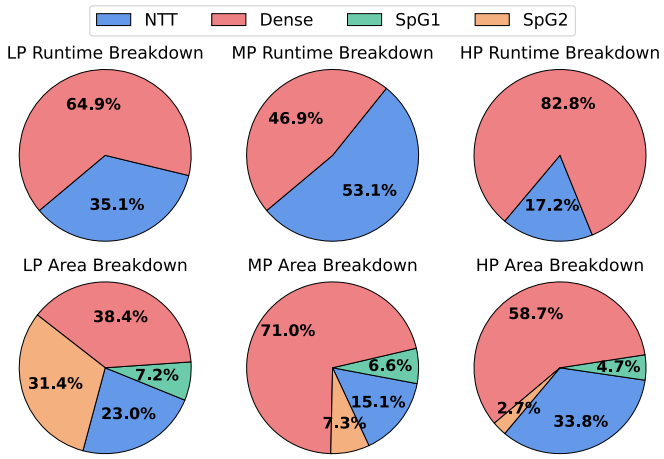

Figure 11: Runtime and Area breakdowns for chosen designs.

Table 7: Full Proof Runtime vs. CPU on BN128

Workload	Size	CPU Time (s)				SZKP-7nm Time (ms)					Speedup
		Poly	Dense	SpG1&G2	Total	Poly	Dense	SpG1	SpG2	Total	
AES	16383	0.103	0.345	0.127	0.576	0.30	0.90	1.07	0.43	1.20	**480×**
SHA2	32767	0.212	0.626	0.051	0.890	0.51	1.54	2.05	0.03	2.05	**434×**
RSA	131071	0.994	2.207	0.520	3.723	2.11	4.73	3.84	0.95	6.85	**544×**
RSASigVer	131071	0.996	2.213	0.423	3.634	2.11	4.73	5.15	0.90	6.85	**531×**
MerkleTree	131071	0.958	2.190	0.376	3.526	2.11	4.73	2.11	0.84	6.85	**515×**
Auction	1048575	8.912	14.834	0.919	24.666	18.63	34.44	36.87	1.02	53.07	**465×**

Table 8: Full Proof Runtime (ms) vs. PipeZK on MNT4753

Workload	Size	PipeZK				SZKP		
		Poly	G1 MSMs	Poly + G1	Total	Poly	Dense	Total[2]
AES	16383	2	21	23	97	1.03 (**1.94×**)	16.17	17.20 (**1.34× / 5.64×**)
SHA2	32767	3	27	30	102	2.00 (**1.50×**)	31.08	33.09 (**0.91× / 3.08×**)
RSA	98303/131071	14	80	94	1230	7.80 (**1.79×**)	90.28	98.09 (**0.96× / 12.54×**)
RSASigVer	131071	14	105	119	822	7.80 (**1.79×**)	120.50	128.30 (**0.93× / 6.41×**)
MerkleTree	294911/524287	63	226	289	2697	32.01 (**1.97×**)	268.87	300.88 (**0.96× / 8.96×**)
Auction	1048575	139	445	584	2053	65.39 (**2.13×**)	507.31	572.70 (**1.08× / 3.79×**)

Table 9: Full Proof Runtime (ms) vs. GZKP on MNT4753

Workload	Size	GZKP			SZKP-12nm		
		Poly	MSM	Total	Poly	Dense	Total
AES	16383	4	99	103	0.31 (**12.8×**)	0.88	1.19 (**86.5×**)
SHA2	32767	5	66	71	0.61 (**8.2×**)	1.54	2.15 (**33.0×**)
RSA	98303/131071	22	120	142	2.37 (**9.3×**)	4.17	6.54 (**21.7×**)
RSASigVer	131071	24	130	154	2.37 (**10.1×**)	5.56	7.93 (**19.4×**)
MerkleTree	294911/524287	60	220	280	9.71 (**6.2×**)	12.19	21.90 (**12.8×**)
Auction	1048575	150	370	520	19.82 (**7.6×**)	22.89	42.71 (**12.2×**)

is chosen as a rough midpoint along the Pareto curve. For each of these designs, we observe that *both* Sparse G1 and G2 cores use an II = 4, demonstrating that they are not on the critical path.

5.2.3 Area and Runtime Breakdowns. Figure 11 shows the runtime critical path and area breakdowns for each performance point. As expected, we see the Sparse G1/G2 PADD units progressively become smaller proportions of the area as Dense MSM and NTT cores become heftier. Interestingly, MP has a relatively balanced runtime. Designs similar to MP might be preferable for optimizing throughput at a *proof*-level of pipelining. For example, MP designs could benefit *proof-as-a-service* applications where a prover is computing multiple proofs for clients and is not significantly bottlenecked by the Dense MSM runtimes. This highlights yet another advantage of decoupling Sparse MSMs from the critical path, as it allows for designs that otherwise may not have appeared in the design space.

5.2.4 Full-proof Speedup. Similar to measuring the speedup of NTT/Poly and Dense MSMs, we measure the speedup of our architecture over CPU when scaled down to 7nm. We pick low-area designs near 33 mm^2 in 22nm and scale our runtimes to 7nm based on the aforementioned scale factors and report speedups in Table 7. SZKP achieves speedups of 434-544× over CPU benchmarks.

Tables 8 and 9 show full-proof speedups over PipeZK and GZKP on 753b data. We show speedups over only Poly and Total runtimes because neither GZKP nor PipeZK parallelize Sparse MSMs from Dense MSMs. Furthermore, SZKP's NTT architecture only supports power-of-2 workload lengths, while PipeZK and GZKP report results for workloads that are of "step-radix" length and require a different polynomial algorithm [2] than shown in Figure 1. For fair comparison, we report our power-of-2 length runtimes for Poly and "step-radix" length runtimes for the Dense MSM, since MSMs are not bound to power-of-2 lengths.

For comparison with PipeZK, we pick an MSM with 1 PE, 5-bit windows, 1024 points per window (71 mm^2) and an NTT with 1 PE, 4 butterflies/PE (20 mm^2). These designs are dominated in area by memory due to double buffering used in both MSM and NTT modules; additionally, we use pessimistic memory area estimates due to the memory compiler's inability to directly generate 753b memories. Our total runtime is roughly as fast as PipeZK's excluding G2 computation. As previously mentioned, our runtime estimates include bucket and window reductions while PipeZK performs them on CPU; PipeZK does not report reduction latencies. Left un-accelerated, these reduction steps could dominate the overall MSM latency. With offloaded-G2 latencies included, we achieve 3 – 12x speedup over PipeZK since G2 computations become their critical path. Additionally, PipeZK's 254b architecture is potentially less efficient than their 753b one. Given that their 753b single-PE

NTT/MSM design is faster than G2-on-CPU, their 254b 4-PE design is likely much faster than G2-on-CPU. This implies more idle accelerator time while waiting for G2-on-CPU. Even if PipeZK accelerated G2, for all bit-widths, they would likely incur a high area footprint (that SZKP would not) from complex FIFO control logic, since G2 uses pairs of points. This further limits their scalability.

For comparison with GZKP, we use the same design mentioned in section 5.1.3. Adjusted for length, SZKP achieves 12 – 86x full-proof speedup over GZKP, using *half* the area with *conservative* scaling factors at *slow* ASIC frequencies.

Table 10: Full-Chip Power and Power Density

Design	Area (mm^2)	TDP (W)	Power Density (W/mm^2)
AMD EPYC 7502	296	180 [4]	0.61
SZKP-7nm	78.7	34	0.43
NVIDIA V100	815	300 [1]	0.37
SZKP-12nm	832	184	0.22

5.2.5 Power. We estimate the thermal design power (TDP) for SZKP designs by constructing power traces of each module; we assume 100% utilization to estimate worst-case, peak, full-chip power. We also include a 1 TB/s, 32W HBM stack [19, 20]. We look at high-performance designs for both 254b and 753b and select those with highest peak power. We then scale down the 254b design to 7nm (area by 3.6×, power by 3.3×) and the 753b design to 12nm (area by 2×, power by 2.5×) [13, 28]. We compare these two designs' TDP with that of the AMD EPYC 7502 (for 254b) and the NVIDIA V100 (for 753b). Table 10 shows these comparisons. High-performance SZKP architectures fall well within the TDP and power density limits of high-performance CPUs and GPUs.

5.2.6 Bandwidth Analysis. Our initial Pareto analysis examines SZKP under *unconstrained* bandwidth to identify peak performance across the design space. However, recent ZKP and privacy-centric accelerators have demonstrated high bandwidth needs. We extend our analysis by modeling SZKP under 6 memory technologies including DDR4 with 4 and 8 channels and four generations of HBM.

Our results on 254b data are illustrated in Figure 13. We can see that across all workloads, low- and medium-performance design points can make do with a 4-channel DDR4 or HBM1. High-performance designs benefit from increased bandwidth, but here too HBM2E suffices to achieve close to ideal performance.

Extending our analysis to 753b, we choose 4 design points to reflect the wider range of areas in our design space. These are chosen to be iso-area (before scaling) with existing GPUs – LP at 475 mm^2 to match an NVIDIA GTX 1080i [7], MP at 815 mm^2 to match an NVIDIA Tesla V100 [1], HP at 1600 mm^2 to match 2 V100s, and UHP (Ultra High Performance) at the largest design point per workload, ranging from 2000 mm^2 to 3500 mm^2 to match multiple V100s. UHP represents an extreme for ASICs, but we include it to highlight the range of architectures available in our optimal design space. These GPU references are based on those used by GZKP's implementation [25]. As seen in Figure 14, we find that even the large LP and MP designs can reach close to optimal performance with an 8-channel DDR4, and HP and UHP in several cases can make do with HBM2 or

[2]speedup over Poly+G1 / speedup over total

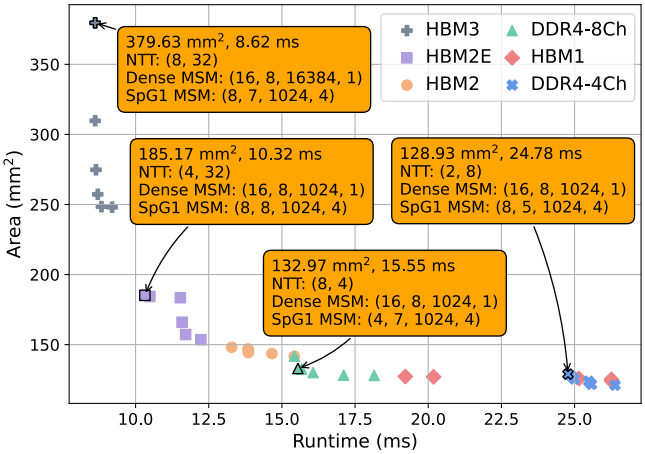

Figure 12: Bandwidth-Aware Pareto Frontiers on BN128. All SpG2 modules have their parameters as (1, 5, 1024, 4).

HBM2E, which are increasingly prevalent in recent fully homomorphic encryption accelerators [18–20, 33, 34]. These results show that SZKP, when scaled to GPU sizes, still can perform optimally with comparatively modest bandwidth requirements.

We next factor bandwidth constraints into the design space optimization. Figure 12 shows the fastest Pareto-optimal designs under each bandwidth constraint on the longest workload (Auction) for 254b datatypes. Because the NTT unit is the bandwidth-intensive feature of SZKP, we see that resource-intensive NTTs are preferable under less-stringent bandwidth constraints. Notably, each of the highlighted design points has 16 PEs in the Dense MSM, indicating that SZKP's Dense MSM is clearly bandwidth efficient and scalable. We observe these trends across all workloads on both bit-widths. These observations reinforce our intuitions that scalable ZKP design is achievable for ASICs when the correct dataflow is chosen.

6 Related Work

PipeZK [37] is currently the only other custom hardware accelerator for full-proof zkSNARKs. PipeZK assumes relatively small (50 mm^2) chips and has difficulty scaling to larger designs because of the need for large multi-ported memories and complex control logic. SZKP, in contrast, is designed to easily scale from small to large designs, enabling a large design space to cover a range of applications. Additionally, for the MNT4753 curve, the Sparse G2 module has a significant contribution to overall chip area even after II optimization. This is because field multiplier (and PADD) sizes grow quadratically with field size, suggesting that full-chip evaluations including all computational modules are necessary to fully understand the benefits of custom ZKP accelerators.

GZKP [25] is a recent GPU-based ZKP hardware acceleration framework that leverages a custom finite field library to accelerate ZKP primitives. Their NTT module uses the traditional Cooley-Tukey NTT dataflow; as such, their NTT heavily depends on caching mechanisms to address the irregular memory access patterns. GZKP's MSM implementation is similar to SZKP in the sense that they perform bucket reductions after all points have been processed by a window, to avoid intermittent bucket reductions. Additionally, they eliminate window reductions, but they require preprocessing

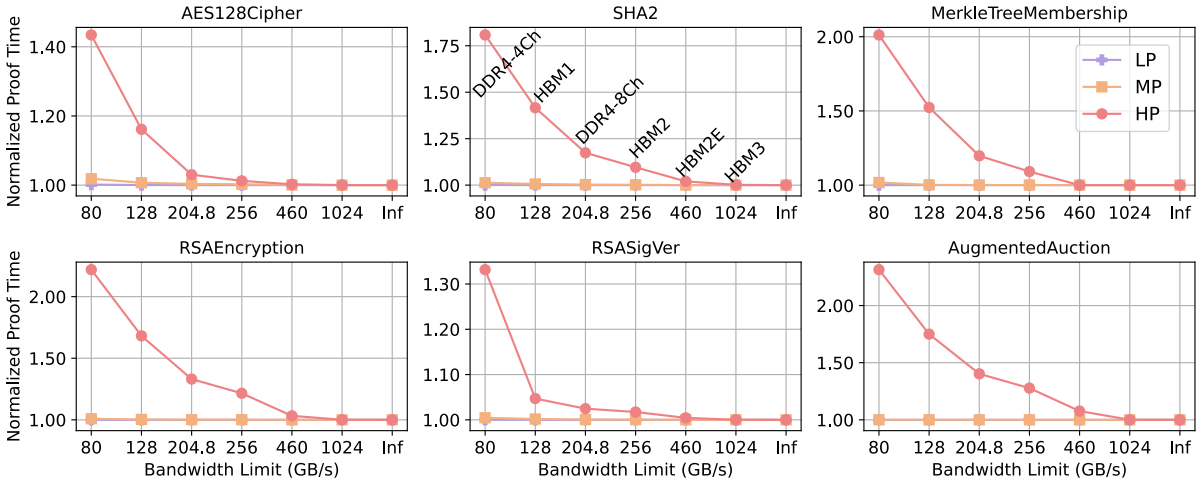

Figure 13: Normalized proof generation times on BN128 across memory technologies

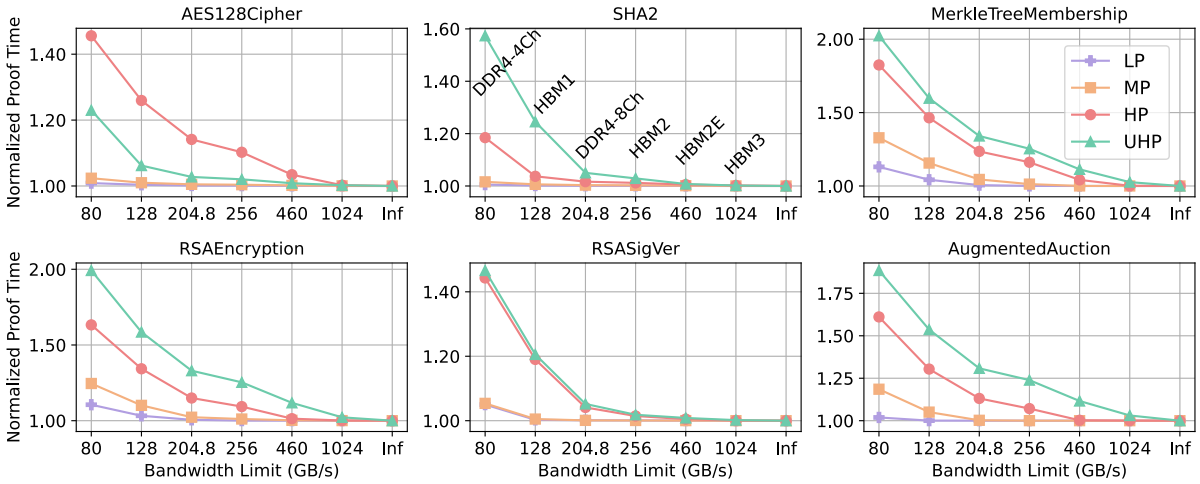

Figure 14: Normalized proof generation times on MNT4753 across memory technologies

overheads upwards of 5GB to store precomputed offsets for each window. They leverage a checkpointing scheme to achieve better balances between the space and time, but this solution, given the high memory overhead, is generally not scalable for ASIC-based designs. cuZK [24] is another recent GPU-based work that accelerates MSMs by converting all operations into sparse-matrix computations. They report linear speedup over Pippenger's algorithm, and additionally perform optimizations to minimize overheads for CPU-GPU data transfers. SZKP outperforms both of these prior works.

There is a large body of work on NTT acceleration, including hand-coded designs [18–20, 33, 34, 37] and designs generated using domain-specific languages like SPIRAL [14, 26]. SZKP uses HLS to synthesize high-performance NTT modules starting with a software implementation of constant-geometry NTTs. HLS is valuable because it enables both an easy design space exploration and a direct route from a software cryptography library to a hardware accelerator. As far as we know, prior work on HLS-generated NTTs only used Cooley-Tukey implementations which resulted in long synthesis runtimes and less performant designs [27].

7 Conclusion

In this paper, we present SZKP, the *first* ASIC framework that accelerates full zkSNARK proofs on-chip with a specific emphasis on scalability. Through a comprehensive design space exploration, we show how ZKPs can be constructed to yield orders of magnitudes of speedup over CPU, scale to GPU sizes while maintaining reasonable bandwidth needs, and outperform state-of-the-art GPUs and ASICs. SZKP makes a strong case for scalable ASIC solutions to the widespread adoption of ZKP hardware architectures.

Acknowledgments

We thank the reviewers for their valuable comments. Support for this work was provided in part by NSF CAREER award #2340137 and via a Google gift award, including GCP credits. This work was also supported by NSF RINGS #2148293 and ARL.

References

[1] 2017. NVIDIA TESLA V100 GPU ARCHITECTURE. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[2] 2018. libsnark: a C++ library for zkSNARK proofs. https://github.com/scipr-lab/libsnark

[3] 2019. AMD DOUBLES DOWN – AND UP – WITH ROME EPYC SERVER CHIPS. https://www.nextplatform.com/2019/08/07/amd-doubles-down-and-up-with-rome-epyc-server-chips/

[4] 2019. AMD EPYC 7502. https://www.amd.com/en/products/cpu/amd-epyc-7502

[5] 2024. AMD EPYC 7502. https://www.techpowerup.com/cpu-specs/epyc-7502.c2250

[6] 2024. Beyond ZK: Next Steps for Compliance and Constrained Encryption on Blockchains. https://a16zcrypto.com/posts/videos/beyond-zk-next-steps-for-compliance-and-constrained-encryption-on-blockchains/

[7] 2024. NVIDIA GTX 1080i. https://www.techpowerup.com/gpu-specs/geforce-gtx-1080-ti.c2877

[8] D. H. Bailey. 1989. FFTs in external or hierarchical memory. In *Supercomputing '89:Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*. 234–242. https://doi.org/10.1145/76263.76288

[9] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. 2013. Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture. Cryptology ePrint Archive, Paper 2013/879. https://eprint.iacr.org/2013/879 https://eprint.iacr.org/2013/879.

[10] Juan Benet and Nicola Greco. 2017. Filecoin: A decentralized storage network. In *Information and Communications Security*. Protocol Labs, Cham, 1–36.

[11] Daniel J. Bernstein, Jeroen Doumen, Tanja Lange, and Jan-Jaap Oosterwijk. 2012. Faster batch forgery identification. Cryptology ePrint Archive, Paper 2012/549. https://eprint.iacr.org/2012/549 https://eprint.iacr.org/2012/549.

[12] James W. Cooley, Peter A. W. Lewis, and Peter D. Welch. 1969. The Fast Fourier Transform and Its Applications. *IEEE Transactions on Education* 12, 1 (1969), 27–34. https://doi.org/10.1109/TE.1969.4320436

[13] Nanotechnology Products Database. 2023. TSMC 16FF+ (FinFET Plus). (2023). https://product.statnano.com/product/6775/16ff-(finfet-plus)

[14] Franz Franchetti, Tze Meng Low, Doru Thom Popovici, Richard M. Veras, Daniele G. Spampinato, Jeremy R. Johnson, Markus Püschel, James C. Hoe, and José M. F. Moura. 2018. SPIRAL: Extreme Performance Portability. *Proc. IEEE* 106, 11 (2018), 1935–1968. https://doi.org/10.1109/JPROC.2018.2873289

[15] Mario Garrido. 2022. A Survey on Pipelined FFT Hardware Architectures. *J. Signal Process. Syst.* 94, 11 (nov 2022), 1345–1364. https://doi.org/10.1007/s11265-021-01655-1

[16] Shafi Goldwasser, Silvio Micali, and Chales Rackoff. 2019. *The Knowledge Complexity of Interactive Proof-Systems*. Association for Computing Machinery, New York, NY, USA, 203–225. https://doi.org/10.1145/3335741.3335750

[17] Jens Groth. 2016. On the Size of Pairing-based Non-interactive Arguments. Cryptology ePrint Archive, Paper 2016/260. https://eprint.iacr.org/2016/260 https://eprint.iacr.org/2016/260.

[18] Jongmin Kim, Sangpyo Kim, Jaewan Choi, Jaiyoung Park, Donghwan Kim, and Jung Ho Ahn. 2023. SHARP: A Short-Word Hierarchical Accelerator for Robust and Practical Fully Homomorphic Encryption. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 18, 15 pages. https://doi.org/10.1145/3579371.3589053

[19] Jongmin Kim, Gwangho Lee, Sangpyo Kim, Gina Sohn, Minsoo Rhu, John Kim, and Jung Ho Ahn. 2022. ARK: Fully Homomorphic Encryption Accelerator with Runtime Data Generation and Inter-Operation Key Reuse. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1237–1254. https://doi.org/10.1109/MICRO56248.2022.00086

[20] Sangpyo Kim, Jongmin Kim, Michael Jaemin Kim, Wonkyung Jung, John Kim, Minsoo Rhu, and Jung Ho Ahn. 2022. BTS: An Accelerator for Bootstrappable Fully Homomorphic Encryption. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 711–725. https://doi.org/10.1145/3470496.3527415

[21] David G. Korn and Jules J. Lambiotte. 1979. Computing the Fast Fourier Transform on a Vector Computer. *Math. Comp.* 33, 147 (1979), 977–992. http://www.jstor.org/stable/2006072

[22] Ahmed Kosba. 2019. jsnark: a Java library for building circuits for preprocessing zk-SNARKs. https://github.com/akosba/jsnark

[23] Patrick Longa and Michael Naehrig. 2016. Speeding up the Number Theoretic Transform for Faster Ideal Lattice-Based Cryptography. Cryptology ePrint Archive, Paper 2016/504. https://eprint.iacr.org/2016/504 https://eprint.iacr.org/2016/504.

[24] Tao Lu, Chengkun Wei, Ruijing Yu, Chaochao Chen, Wenjing Fang, Lei Wang, Zeke Wang, and Wenzhi Chen. 2022. cuZK: Accelerating Zero-Knowledge Proof with A Faster Parallel Multi-Scalar Multiplication Algorithm on GPUs. Cryptology ePrint Archive, Paper 2022/1321. https://eprint.iacr.org/2022/1321 https://eprint.iacr.org/2022/1321.

[25] Weiliang Ma, Qian Xiong, Xuanhua Shi, Xiaosong Ma, Hai Jin, Haozhao Kuang, Mingyu Gao, Ye Zhang, Haichen Shen, and Weifang Hu. 2023. GZKP: A GPU Accelerated Zero-Knowledge Proof System. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 340–353. https://doi.org/10.1145/3575693.3575711

[26] Lingchuan Meng, Yevgen Voronenko, Jeremy R. Johnson, Marc Moreno Maza, Franz Franchetti, and Yuzhen Xie. 2010. Spiral-generated modular FFT algorithms. In *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation* (Grenoble, France) *(PASCO '10)*. Association for Computing Machinery, New York, NY, USA, 169–170. https://doi.org/10.1145/1837210.1837235

[27] Ahmet Can Mert, Emre Karabulut, Erdinç Öztürk, Erkay Savaş, and Aydin Aysu. 2022. An Extensive Study of Flexible Design Methods for the Number Theoretic Transform. *IEEE Trans. Comput.* 71, 11 (2022), 2829–2843. https://doi.org/10.1109/TC.2020.3017930

[28] Jianqiao Mo, Jayanth Gopinath, and Brandon Reagen. 2023. HAAC: A Hardware-Software Co-Design to Accelerate Garbled Circuits. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 10, 13 pages. https://doi.org/10.1145/3579371.3589045

[29] Peter L. Montgomery. 1985. Modular Multiplication Without Trial Division. *Math. Comp.* 44, 170, 519–521. http://www.jstor.org/stable/2007970

[30] Marshall C. Pease. 1968. An Adaptation of the Fast Fourier Transform for Parallel Processing. *J. ACM* 15, 2 (apr 1968), 252–264. https://doi.org/10.1145/321450.321457

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

[32] Sujoy Sinha Roy, Furkan Turan, Kimmo Jarvinen, Frederik Vercauteren, and Ingrid Verbauwhede. 2019. FPGA-based High-Performance Parallel Architecture for Homomorphic Computing on Encrypted Data. Cryptology ePrint Archive, Paper 2019/160. https://eprint.iacr.org/2019/160 https://eprint.iacr.org/2019/160.

[33] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald Dreslinski, Christopher Peikert, and Daniel Sanchez. 2021. F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 238–252. https://doi.org/10.1145/3466752.3480070

[34] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Nathan Manohar, Nicholas Genise, Srinivas Devadas, Karim Eldefrawy, Chris Peikert, and Daniel Sanchez. 2022. CraterLake: A Hardware Accelerator for Efficient Unbounded Computation on Encrypted Data. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 173–187. https://doi.org/10.1145/3470496.3527393

[35] Deepraj Soni, Negar Neda, Naifeng Zhang, Benedict Reynwar, Homer Gamil, Benjamin Heyman, Mohammed Nabeel, Ahmad Al Badawi, Yuriy Polyakov, Kellie Canida, Massoud Pedram, Michail Maniatakos, David Bruce Cousins, Franz Franchetti, Matthew French, Andrew Schmidt, and Brandon Reagen. 2023. RPU: The Ring Processing Unit. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 272–282. https://doi.org/10.1109/ISPASS57527.2023.00034

[36] Jiaheng Zhang, Zhiyong Fang, Yupeng Zhang, and Dawn Song. 2020. Zero Knowledge Proofs for Decision Tree Predictions and Accuracy. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20)*. Association for Computing Machinery, New York, NY, USA, 2039–2053. https://doi.org/10.1145/3372297.3417278

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

[38] Zhichao Zhao and T.-H. Hubert Chan. 2016. How to Vote Privately Using Bitcoin. In *Information and Communications Security*, Sihan Qing, Eiji Okamoto, Kwangjo Kim, and Dongmei Liu (Eds.). Springer International Publishing, Cham, 82–96.