Scaling Zero-Knowledge to Verifiable Databases

Tal Derei
tad222@lehigh.edu
Lehigh University
Bethlehem, PA, USA

Benjamin Aulenbach*
baulenbach@macguyvertech.com
Macguyver Tech
Glenolden, PA, USA

Victor Carolino
Caleb Geren
Michael Kaufman
Jon Klein
vjc225@lehigh.edu
cdg225@lehigh.edu
mpk225@lehigh.edu
jak325@lehigh.edu
Lehigh University
Bethlehem, PA, USA

Rishad Islam Shantho
mds222@lehigh.edu
Lehigh University
Bethlehem, PA, USA

Henry F. Korth
hfk2@lehigh.edu
Lehigh University
Bethlehem, PA, USA

ABSTRACT

Zero-Knowledge proofs are a cryptographic technique to reveal knowledge of information without revealing the information itself, thus enabling systems optimally to mix privacy and transparency, and, where needed, regulatability. Application domains include health and other enterprise data, financial systems such as central-bank digital currencies, and performance enhancement in blockchain systems. The challenge of zero-knowledge proofs is that, although they are computationally easy to verify, they are computationally hard to produce. This paper examines the scalability limits of leading zero-knowledge algorithms and addresses the use of parallel architectures to meet performance demands of applications.

CCS CONCEPTS

• **Security and privacy** → **Database activity monitoring**; **Information flow control**; **Cryptography**;

KEYWORDS

zero-knowledge, blockchain, performance, PlonK

ACM Reference Format:
Tal Derei, Benjamin Aulenbach, Victor Carolino, Caleb Geren, Michael Kaufman, Jon Klein, Rishad Islam Shantho, and Henry F. Korth. 2023. Scaling Zero-Knowledge to Verifiable Databases. In *Proceedings of Workshop on Verifiable Database Systems (VDBS).* ACM, New York, NY, USA, 9 pages. https://doi.org/XXXXXXX.XXXXXXX

*Work done while at Lehigh University.

VDBS, June 23, 2023, co-located with ACM SIGMOD Seattle, WA

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

The cryptograhic theory of zero-knowledge(ZK)[15] arose in the 1980s and led to a 2012 Turing Award for Goldwasser and Micali. ZK-proofs employ the mathematics of cryptography to enable one party (P, the prover) to prove to another party (V, the verifier) the P knows some particular secret (a fact, data item, etc.) without P revealing that secret to V. Practical application of this concept grew rapidly with the rise of blockchain technology, initially for privacy-focused blockchains such as Zcash[2], and several others that followed. A more recent application is blockchain speedup using ZK techniques to aggregate a large number of transactions into one, a process called a *ZK-rollup*. A third application domain is to apply ZK and blockchain technology to large reports and databases so that applications can commit cryptographically to data that are kept secret while proving that those data have certain properties, such as compliance with regulatory constraints or being internally consistent with some other set of data to which a cryptographic commitment exists.

That third application domain opens a powerful new concept of information management in which privacy, transparency, and compliance can be achieved with minimal compromise. We elaborate on this in Section 2. These applications generate a need to produce ZK-proofs pertaining to large-scale executions over large data volumes, which presents a challenge since although ZK-proofs are relatively easy to verify, they are computationally hard to generate. This demand for scalability has been addressed in part by new algorithms (e.g., [14]), use of specialized hardware (e.g., [28]), and use of commodity parallel architectures (e.g., [18, 19, 27]), but significant limitations remain.

After exploring the application domain in Section 2, and providing a fuller introduction to ZK in Section 3, we overview ZK proving systems in Section 4 followed by our benchmarking results and analysis in Section 5. Section 6 discusses the use of commodity parallel architectures to accelerate ZK-proofs so that they can scale up to the level of emerging applications.

2 APPLICATION DOMAIN

Traditional databases serve as an official record of enterprise information, but the degree to which they are truly "official" is based on trust of the owner/administrator of the database. Some commercial database vendors have incorporated hash-protected, immutable data in append-only tables. Here, immutability and privacy still rests in part on some degree of centralized trust. Such trust is often reasonable in enterprise settings (thus the frequent use of permissioned blockchains), but becomes problematic for public data stores (such as public blockchains), multi-enterprise data (where a trust/contract-based business relationship exists, but participants fear that future circumstances like a product recall might stress that trust to the breaking point), and global digital-currency applications where users may not trust the currency provider with details of their transaction data.

For these reasons, even blockchain-enhanced versions of traditional database platforms may require too much trust. Privacy is often of paramount concern. In business relationships, certain data may be considered proprietary (e.g., pricing in a supply chain) and must be disclosed selectively. Public-facing systems face stricter privacy challenges that go beyond user preferences to include mandates by governments. Privacy in a public setting is often viewed as a fundamental right with individuals reasonably valuing a right to privacy even if they "have nothing to hide."

Applications of ZK technology can be divided into three classes:

- The first, and oldest, is private transactions within a public blockchain. Whereas all transactions on a public blockchain like Ethereum or Bitcoin are public, chains such as Zcash enable the option of privacy. In public blockchains, if person A knows person B's wallet address, then A knows B's entire financial history on-chain. ZK-based blockchains allow for secret transactions, thus enhancing privacy. Such chains have raised concerns among government regulators, but it is possible to design such private chains with ZK-based compliance tools to alleviate those concerns.
- The second relates to blockchain speedup, where ZK techniques can be used to aggregate a large number of transactions into one, a process called a ZK-rollup. Aside from the single node performing aggregation, all other nodes then need only validate a single proof of that aggregation. This results in a significant performance enhancement since that one validation is less costly than re-executing full set of transactions. There are other ZK applications in blockchain infrastructure including state-proofs for light clients and cross-chain bridges[5, 16], and proof-of-storage in decentralized storage services[24].
- The third is the ability to provide proofs of general facts about secret information. Those proofs can then be verified at low computational cost by any node. Here, information can be kept secret, but desired metadata can be securely published. This is a powerful tool that can be applied in identity proofs, accounting systems, health records, regulatory compliance in financial applications, and global-scale central-bank digital currencies.

The range of real-world uses for this third class of application is virtually limitless. We give three examples here:

(1) A relatively simple application is proof-of-age for bars and other age-controlled venues without the need to reveal exact date-of-birth, drivers' license number, etc.
(2) A more complex application allows proof that corporate tax returns, government-mandated regulatory reports, etc., all come from the same set of accounting records. Thus, only the base ledger need be audited in the traditional manner, while all reports arising from that ledger can be audited via ZK and related technologies, including Merkle proofs[21]. A similar scenario exists around health records.
(3) Central-bank digital currencies (CBDCs) are gaining interest globally. China has begun deployment of its centrally-controlled digital e-yuan. Open societies such as the U.S. would most certainly seek a much less centralized framework, but full decentralization would raise risks regarding regulation and control of monetary policy. Nearly all of the G20 nations are taking some action in this regard. Prototypes are being developed in industry and government-led partnerships like Project Hamilton[11]. For digital currencies, ZK-proofs make it possible to have transactions with the same privacy as cash, while also proving compliance with regulations such as sanctions. They also serve as a means for a stablecoin issuer to provide proof of reserves and compliance with other regulatory requirements, while preserving privacy regarding the details of those reserves.

The second and third examples result in very large ZK-proofs and large volumes of medium-size ZK-proofs, respectively. The feasibility of these applications depends, respectively, upon cost-effective frameworks to generate large proofs and resource-efficient frameworks that allow high parallelism in proof generation.

With those needs in mind, we focus not on special-purpose hardware as in [28], but rather on high-performance commodity CPUs and GPUs. Our application-based motivation also guides our choice of algorithms, a matter we discuss in more detail in Section 4.

3 A BRIEF INTRODUCTION TO ZERO KNOWLEDGE

As we noted in the introduction, ZK-proofs employ the mathematics of cryptography to enable one party (P, prover) to prove to another party (V, verifier) that P knows some particular secret (a fact, data item, etc.) without P revealing that secret to V. At that point, we relied on the intuitive idea of proving "some particular secret," but indeed ZK can be used to show P knows the input (a *witness*) such that a given program generates a particular output. That given program must be compiled into a *circuit* representing its execution. A circuit is a computational structure of *gates* with inputs and outputs and *wires* connecting the output of a gate to the input of others. These circuits are then transformed into polynomials that form the basis for the proving system. For details see [4, 23, 25]. The most intuitive view of a ZK proving system is one in which the verifier interacts with the prover, presenting a series of challenges that, when met by the prover, provides the verifier confidence in the prover that increases with each iteration. Interactive proofs, however, are not only slow, but also fail to provide a means for public verification. Removing the interaction adds complexity both in terms of the proof-generation and verification time (which

depends on the proof size). To keep verification time constant, both the prover and verifier must maintain secrets that are not part of the published proof. Those secrets depend on an initial trusted setup of the proof system. The widely used, but older, Groth16[17] system requires a new setup for each circuit. The newer PlonK[14] system requires just one setup (*universal trusted setup*) that can then be reused across any set of circuits. Others in that category include Marlin[6] and Sonic[20].

The setup procedure is an elaborate process involving multiple independent parties selecting a secret random value that they contribute to the "ceremony" and then destroy[22]. The setup is designed such that as long as one trusts that at least one member in the setup has securely destroyed their contributed value, then the setup can be provably trusted.

4 ZK-PROVING SYSTEMS

There are a wide variety of ZK proving systems [25]. ZK-proof systems vary in:

- Proof size: constant for the Groth16 algorithm and the PlonK variants we are studying, but other algorithms have proof sizes that depend on the circuit (program execution code) over which the proof is being computed.
- Prover-secret size: Linear in circuit size for Groth16 algorithm and the PlonK variants, but constant for some others.
- Verification time: Constant for the Groth16 algorithm and the PlonK variants, but usually logarithmic in circuit size for others (though some are linear).
- Degree of setup required: While some algorithms require no setup[1], PlonK requires a single initial trusted setup, and Groth16 requires a separate setup for each circuit.
- Operation set: Groth16 operates over executions expressed as circuits with addition and multiplication operations, as does PlonK. TurboPLONK[13] reduces the number of gates in a circuit by allowing more operation types than just addition and multiplication. The added operations could be simple as exemplified by an exclusive OR operation(XOR), or highly complex as exemplified by operations that perform arithmetic over elliptic curves (which are used in many cryptographic hash functions). Reducing the circuit size promises faster proofs since the prover must read the entire circuit.
- Supplemental data structures: UltraPLONK, like TurboPLONK, allows operations beyond addition and multiplication. Beyond just computational operations, UltraPLONK includes a table-lookup operation that operates in the style of a key-value store. These tables are called *plookup* tables[12].

The tradeoffs among ZK proving systems are not as clear as the above list might suggest. Trading added memory requirements for improving performance may make sense given present-day memory capacity. But for sizable proofs, memory consumption is large enough not only to exceed the total available in the system, but also to inhibit effective parallelization of proof generation. An important first step in the creation of highly scalable proof-generation algorithms is to understand these tradeoffs.

In earlier work[26], our research group focused on the Groth16 algorithm and its execution on a modern GPU. While GPUs offer a high degree of parallelism, memory became a barrier at only 2^{20} constraints[1]. The word "only" here might seem to be a strange characterization of 2^{20} until one realizes that each gate represents a simple arithmetic operation and modern applications easily reach higher numbers of constraints that go well beyond 2^{30}. The applications we envisioned in Section 2 will result in further complexity growth.

Groth's constant verification time is attractive, but the per-circuit setup overhead makes it less desirable than the PlonK variants. For the scale of our applications, the one initial setup for PlonK is quite acceptable. The strongest factor in our focus on PlonK is the variants that use powerful techniques to reduce the circuit size and thus offer the promise of better overall performance. But since that promise comes at the price of additional memory consumption, the intuition that PlonK should perform best is not an obvious conclusion.

5 BENCHMARKING ZK ALGORITHMS

We have run an extensive study of Groth16, PlonK, TurboPLONK, and UltraPLONK on a CPU[8].[2] This study forms a basis for our evaluation of the PlonK family on a GPU, further plans for which are discussed in Section 6.

5.1 Benchmarking Groth16

Using a more powerful platform than that in [26], we evaluated Groth16 in terms of (1) parameter generation, (2) preprocessing generation, (3) CPU proof generation, and (4) GPU proof generation. Our platform is the following bare-metal machine instantiated on Oracle Cloud: 64-core Intel(R) Xeon(R) Platinum 8358 CPU @ 2.60GHz, NVIDIA A10 (Ampere) with 24 GB GDDR6, NVIDIA P100 (Pascal) with 16 GB GDDR6, 1024 GB DDR4 DRAM, 2 TB SSD.

Parameter generation, performed entirely on the CPU, describes the trusted-setup phase that produces the proving and verification keys. In addition to generating the parameters, execution on the GPU requires an additional preprocessing step that precomputes multiples of the base points (preprocessing generation). The precomputation can be reused over multiple multi-scalar multiplication (MSM) steps (described below) with different parameters, saving enough work to justify the precomputation. The parameter and preprocessing phase is done entirely on the CPU. Proof generation describes generating ZK-proofs on CPUs and GPUs, involving MSM operations over elliptic curves and Fast Fourier transforms (FFTs) over large fields. MSM requires multiplications over large vectors and dominates 75-80% of the proof-generation time. FFTs require complex polynomial calculations accounting for about 10-15% of the time. Finally, witness-generation consumes the remaining time.

We report the main results here, with full details in [8]. All benchmarks were run using the MNT4_753 pairing-friendly elliptic curve over a finite field. The MSMs were executed on a single GPU, while the non-blocking FFTs were performed on the CPU in parallel. The maximum constraint size for a program was 2^{25} constraints. A MNT4_753 curve field element is 753 bits in size, which is the main reason why computations are consumptive of processing power.

[1] Following custom, we reference the number of constraints in the rank-1 constraint system, which corresponds to the number of gates in the corresponding circuit.
[2] We show the most relevant data here and refer to https://github.com/TalDerei/Masters-Research/blob/main/Groth16-and-Plonk-Raw-Benchmarks.pdf for the full set of raw benchmark data.

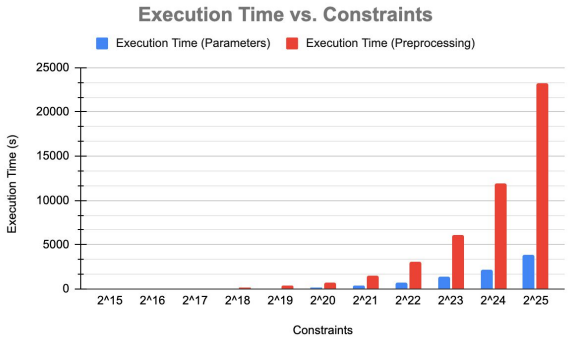

Figure 1: Groth16 CPU performance on parameter and preprocessing generation as a function of number of constraints.

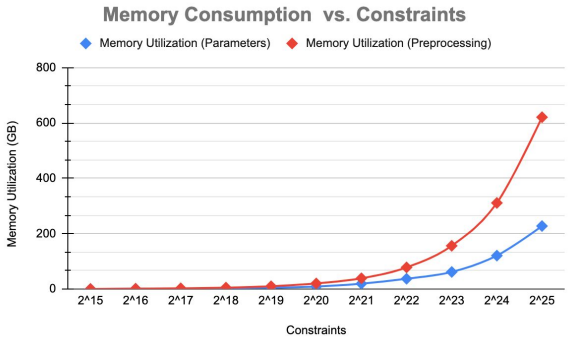

Figure 2: Groth16 memory consumption for parameter and preprocessing generation as a function of number of constraints.

Figures 1 and 2 show results for parameter and preprocessing generation as a function of the number of constraints. Execution time grows exponentially, but memory requirements grow super exponentially, driven by the size of the preprocessing file. CPU utilization was at 100% in all runs. The primary concern for practicality here is memory rather than runtime since prepreprocessing is done just once, but the files produced need to be loaded into memory every time we run the proving system. With further research, we seek to flatten this curve so that, despite the asymptotics, the range of applicability can extend several powers of 2 further.

We turn next to proof generation in Figures 3 through 5. Here, we compare CPU and GPU performance on our platform, both overall and broken down for MSM and FFT computations.

The results show that the GPU-based prover executes roughly two times faster than the CPU-based prover at the point where we hit the limits of our platform. Additionally, execution time and memory utilization grow log-linearly with respect to the number of constraints in the program. For programs larger than 2^{19} constraints, the P100's 16 GB of GPU VRAM is maxed out, which induces a spike in the system's main memory demands. Since the preprocessed parameter file is loaded into main memory, the parameters are then paged between host and device memory. Moving

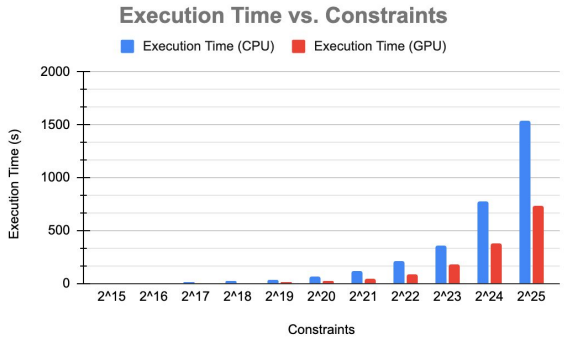

Figure 3: Groth16 CPU and GPU (A10) performance on proof generation as a function of number of constraints.

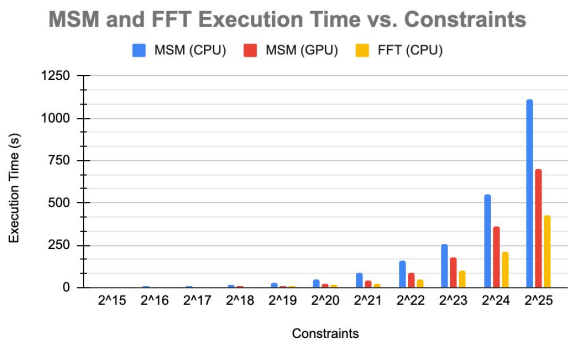

Figure 4: Groth16 CPU and GPU (A10) execution time for multi-scalar multiplication and fast Fourier transforms as a function of number of constraints.

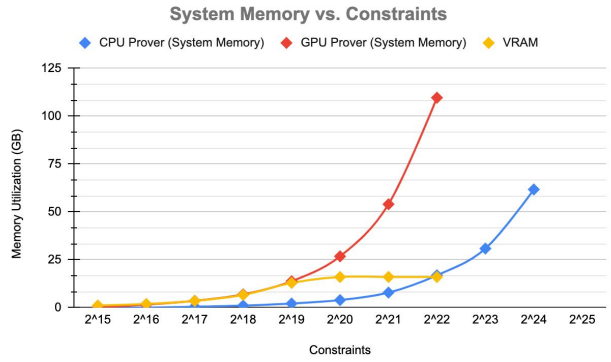

Figure 5: Groth16 system memory consumption for proof generation as a function of number of constraints, shown for CPU prover, GPU prover (P100), and GPU VRAM.

from the P100 to the more powerful A10 GPU allowed us to extend these memory figures to 2^{25} constraints, where the GPU prover maxes out the 24 GB of VRAM and consumes around 900 GB of

system memory. The MSMs and FFTs execute in parallel making it challenging to decouple their CPU and memory utilizations. On the CPU, the MSM dominates with 80% of the prover runtime, FFTs consume 10-15%, and witness generation consumes about 5%. On the GPU, the MSM and FFT runtimes are closer together since they execute in parallel. The results ultimately indicate that prover tradeoffs depend on the computing platform. CPU-based provers use a smaller parameter file and less system memory, but execute slower. GPU-based provers use a larger preprocessed parameter file requiring more system memory and VRAM, but execute faster.

While the exact results we show depend on the platform used, minor changes to the platform would not affect our results significantly. However, parallel GPUs remain an interesting domain for design studies. They offer both more performance and larger GPU memory capacities (40 GB on the NVIDIA A100). The size of the elliptic curve affects prover runtime and raises the issue of the optimal balance between security and performance. The elliptic curve cannot be small enough to allow successful brute force attacks, but cannot be too large and unnecessarily reduce performance.

5.2 Benchmarking PlonK

In this section, we consider the original PlonK algorithm, with variants deferred to the next subsection. The results we show here are for the following bare-metal machine instantiated on Oracle Cloud: 32-core Intel(R) Xeon(R) Platinum 8358 CPU @ 2.60GHz Base Frequency, 1024 GB DDR4 DRAM, 128 GB SSD. Since compiler optimization settings represent a tradeoff between memory and speed, we ran our tests in (1) normal (unoptimized) mode, (2) ASM (assembly instructions) mode, and (3) Full (-03) mode with both assembly instructions and clang compiler optimizations enabled.

We conducted benchmarks that fall into three distinct workloads: arithmetic, polynomial, and prover. The arithmetic benchmarks measure the performance of finite-field arithmetic and elliptic-curve operations over the bn254 elliptic curve. The polynomial benchmarks test the execution time and memory efficiency of the MSM and FFT. The prover benchmarks measured the performance of the entire proof generation process. Full details of our results are in [9]. Here, we show results only for the prover, since those data show where PlonK hits feasibility limits on our platform. Figures 6 and 7 show that the performance barrier for our hardware platform arises at 2^{26} constraints, at which point proof generation took roughly 100 seconds (using full optimization) with 255 GB of system memory used. Executing a single MSM for 2^{26} constraints, of which PlonK computes multiple in generating a proof, took 6.7 seconds and consumed 46 GB of memory. The execution-time curve shows that selecting the proper optimization tradeoffs is clearly valuable in pushing the (still log-linear) curve rightward.

We note that GPU data is still a work-in-progress, pending completion of our project to port PlonK to a GPU. Those results are thus the subject of future work. The matter of porting PlonK to a GPU is nontrivial as we explain in Section 6.

5.3 Benchmarking TurboPLONK and UtraPlonk

We discuss our results using the same CPU platform as in the previous section, but focus now on the tradeoffs among members of

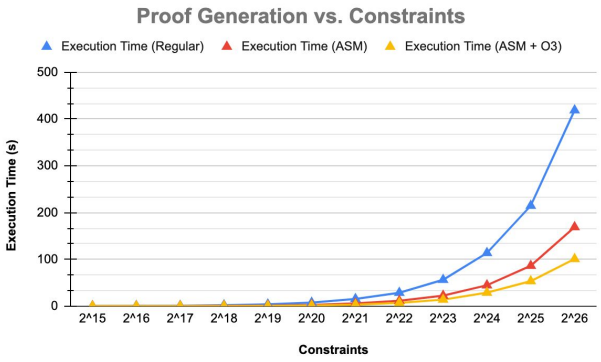

Figure 6: PlonK performance on proof generation as a function of number of constraints.

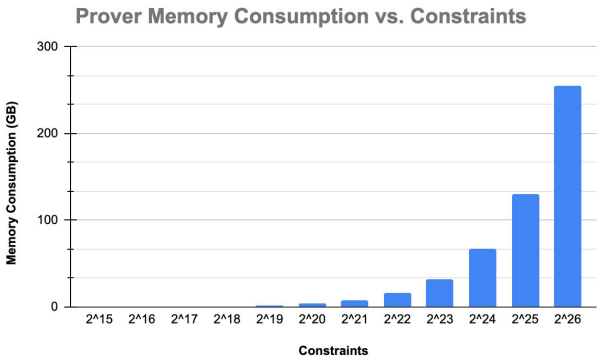

Figure 7: PlonK performance on memory consumption as a function of number of constraints.

the PlonK family. As we noted in Section 4, TurboPLONK[13] generalizes the constraint system by introducing custom gates that represent complicated statements, allowing the same computation to be expressed in a circuit with fewer gates. Custom gates represent more complex operations in a single gate, reducing the number of total gates in the circuit. For instance, cryptographic primitives like a fixed-base elliptic-curve scalar-multiplication, elliptic-curve point arithmetic, and bitwise XOR and AND can be expressed and evaluated with a single custom gate[3]. Consequently, it uses an additional wire, requiring an additional commitment to be computed. Incorporation of more operations comes at a price, however, since the polynomials produced from a circuit are more complex as they must be able to encode selection of the (now several) operations.

UltraPLONK[12] extends this construction with precomputed lookup tables, which represent efficient key-value mappings. This enables a prover to prove that a witness is in a table instead of proving the computation itself. Said differently, by storing precomputed values, the circuit must encode only a table lookup rather than the

[3]This is a more substantial reduction in the number of gates than it might initially appear. These are not standard **int** computations, but rather are performed over large finite fields whose elements do not fit in an **int** nor a **long**.

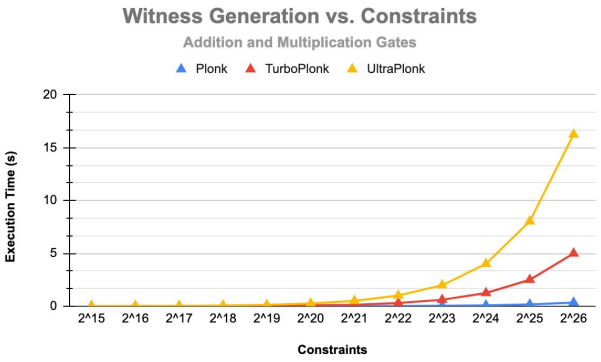

Figure 8: Comparing the PlonK family in performance on the witness-generation component of ZK-proof generation.

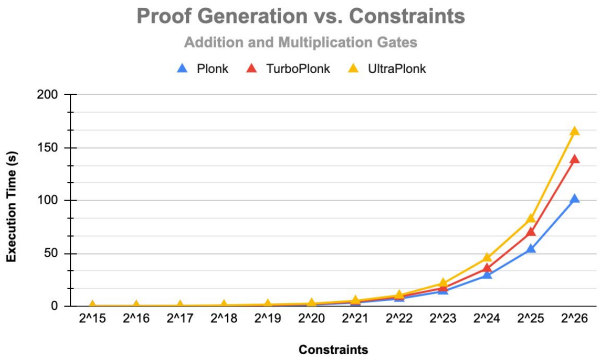

Figure 9: Comparing the PlonK family in performance on the final step in the generation of a ZK proof.

collection of gates and wires that would represent a re-execution of the computation. This results in a reduction in the circuit size. The protocol preprocesses a table T and a lookup operation that checks if a value x is in T. Lookup tables in this context are commonly used to avoid expensive bit-decompositions for bitwise operations. For example, rather than computing an XOR operation bit by bit, you can instead encode the 8-bit result in the table and perform a lookup operation. Bit decompositions are expensive because each bit is represented as a finite field element.

5.3.1 Performance Using Only Addition and Multiplication Gates. We first ran experiments on all three versions using only the same operations as in PlonK, addition and multiplication. This obviously wastes the entire set of advantages of TurboPLONK and UltraPLONK but serves the purpose of allowing us to gain insight into the overhead imposed by their additional infrastructure. We then ran experiments to test the practical benefits of the enhancements in TurboPLONK and UltraPLONK.

The benchmark consists of (1) constructing the arithmetic circuit, (2) calculating witness polynomials, (3) computing the proving key, (4) computing the verifier key, (5) generating the proof, and (6) verifying the proof. In [10], we break down costs at a detailed level, but for space consideration here, we show only results for witness generation (Figure 8) and proof generation (Figure 9) as they represent extremes in differentiating performance.[4]

TurboPLONK and UltraPLONK exhibit worse performance because they are structured to optimize performance in the presence of custom gates, which were not used here. In general, performance and memory are proportional to the number of gates in the circuit.

For 2^{26} constraints, generating a TurboPLONK proof (running at roughly 138 seconds) is 37% slower and UltraPLONK (running at roughly 165 seconds) is 63% slower compared to generating a PlonK proof (running at roughly 101 seconds). UltraPLONK is 19% slower than TurboPLONK in the same setting. TurboPLONK and UltraPLONK further display larger memory consumption footprints than PlonK. For 2^{26} constraints, generating a TurboPLONK proof consumed 29.5% more memory (330 GB) and UltraPLONK consumed nearly twice the memory (506 GB) with respect to generating a PlonK proof (255 GB). In this setting with only addition and multiplication gates, PlonK is more efficient because it requires fewer group exponentiations. Consequently, less prover work translates to reduced proof generation and memory consumption.

Let N represent the number of gates in the circuit, and let P represent the polynomial. Then PlonK requires $9N$ scalar multiplications and a proof size of approximately 9 curve elements, TurboPLONK requires $11N$ scalar multiplications and a proof size of approximately 11 curve elements, and UltraPLONK requires $13N$ scalar multiplications and a proof size of 13 curve elements. Group exponentiation dominates approximately 70-80% of the prover runtime. All three proving systems compute a collection of polynomials and stores each in 3 forms:

(1) Coefficient form: $N * P$
(2) Lagrange form: $N * P$
(3) Coset-FFT form: $4N * P$

One could run the PlonK prover storing just the coefficient form, which would reduce the memory consumption by a factor of 6. This comes at an additional runtime computational cost resulting from not pre-computing and storing the polynomials in all three forms.

5.3.2 Performance Using the Full Power of TurboPLONK and UltraPLONK. In using the full power of each member of the PlonK family, we expect performance to improve for TurboPLONK and UltraPLONK, compared to the results shown above since custom gates replace gates used in the circuit. We expect further improvement in UltraPLONK since repeated computations are replaced with a table lookup operation. If we have a custom gate to calculate $y = f(x)$ for some complex function, or a lookup table to encode $y = g(x)$ mapping, these improvements should improve performance. But if these operations are sparsely used in large circuits, then TurboPLONK and UltraPLONK will actually be slower compared to PlonK.

With our goal here of showing the power of additional operations, we focus on the computation of Pedersen hashes, which are used to map a bit string to a compressed point on an elliptic

[4]We note that our charts display only the x-axis in logarithmic scale to highlight the execution time rather than time complexity. Presenting these charts in log-scale accentuates a O(n log n) growth rate, but excludes some of our data on the y-axis.

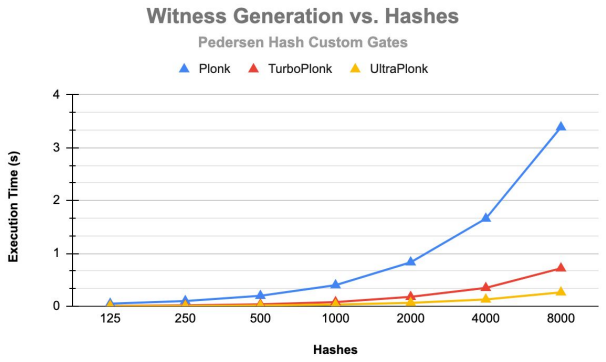

Figure 10: Comparing the PlonK family in performance on witness generation.

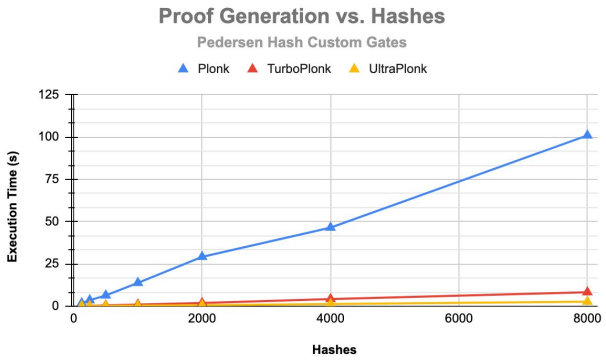

Figure 11: Comparing the PlonK family in performance on the final step in the generation of a ZK proof.

curve. This is a highly complex cryptographic computation consuming a large number of addition and multiplication gates. It is important to note here that although we continue our pattern of showing graphs with the x-axis as "number of constraints," each algorithm now requires a different number of constraints for the same computation, with PlonK requiring more than TurboPLONK, and TurboPLONK requiring more than UltraPLONK. Our discussion below will enable a conversion between number-of-constraints and number-of-hashes.

We show in Figures 10 and 11, results for witness and proof generation that can be compared to the similar curves for the baseline case of only addition and multiplication gates that we showed earlier (Figures 8 and 9). Figure 12 shows a comparison of memory consumption. A full set of performance data appears in [10]. For 8K hashes, generating a TurboPLONK proof (at roughly 8 seconds) is about 12 times faster and UltraPLONK (at roughly 3 seconds) is about 38 times faster compared to generating a standard PlonK proof (at roughly 100 seconds). UltraPLONK is 3 times faster than TurboPLONK in the same setting. PlonK further displays a larger memory consumption. Generating a TurboPLONK proof consumed about 16 times less memory (16 GB) and UltraPLONK consumed about 32 times less memory (8 GB) compared to generating a standard PlonK proof (254 GB). The memory of PlonK is greater than TurboPLONK, as expected. But the memory of TurboPLONK is greater compared to UltraPLONK, which is contrary to what one might expect given the need for UltraPLONK to store lookup tables. This is due to the fact that, in our discussion here, we are evaluating the proof systems based on the same task size, i.e., 8K hashes, rather than the same problem size, e.g., 2^{25} constraints.

The task sizes used above generated a number of constraints that stressed PlonK to its limits on our platform. We therefore abandoned PlonK in pursuing larger task sizes in a further exploration of the relative performance of TurboPLONK and UtraPlonk over a range of 4k to 128k Pedersen hashes.

We show results for proof generation (Figure 13), and memory consumption (Figure 14). See [10] for our full set of results. At the highest number of hashes shown, rover times for UltraPLONK (roughly 42 seconds) were about 3 times faster than TurboPLONK (roughly 125 seconds), and the memory consumption

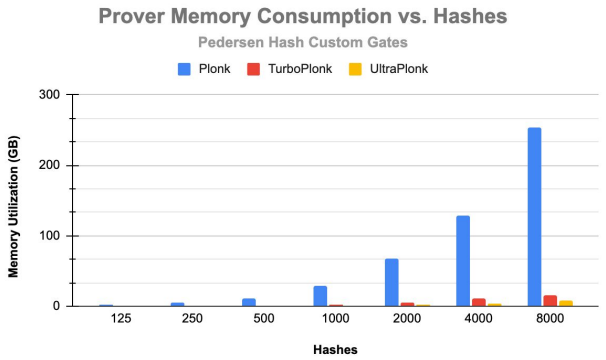

Figure 12: Comparing the PlonK family in memory consumption.

for UltraPLONK (about 130 GB) was about 2.5 times lower than TurboPLONK (about 330 GB). The verification time (not shown) for UltraPLONK is slightly slower than TurboPLONK since there is a tradeoff between proof-generation time and verification time. There is an inverse relationship where faster proof generation yields slower proof verification, and vice versa. In general, fast provers have large proofs with slow verifiers, while slow provers have small proofs with fast verifiers.

Generating a proof for 128K hashes requires radically different circuit sizes between TurboPLONK and UltraPLONK:

- UltraPLONK: 13,191,211 constraints / 103 gates per hash = 128K hashes
- TurboPLONK: 44,184,152 constraints / 345 gates per hash = 128K hashes

These figures are comparable because they compute the same number of hash operations. TurboPLONK ultimately has roughly 2.5 times increased memory consumption than UltraPLONK when evaluating 128K hashes because it exhibits about a 3.3 times increase in circuit size. TurboPLONK has more gates per hash than UltraPLONK, since UltraPLONK employs lookup tables that reduce the circuit's constraint size. Therefore, 128K hashes in TurboPLONK is

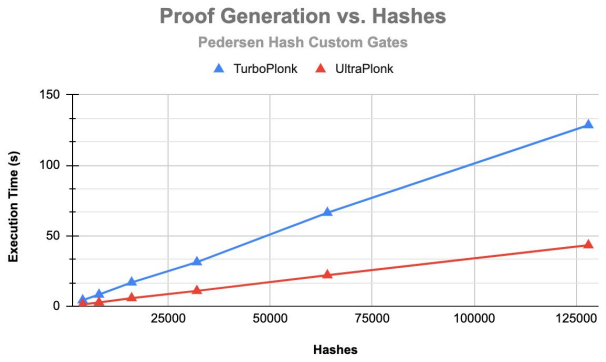

Figure 13: Comparing final proof generation time in TurboPLONK and UtraPlonk over an extended range of task sizes.

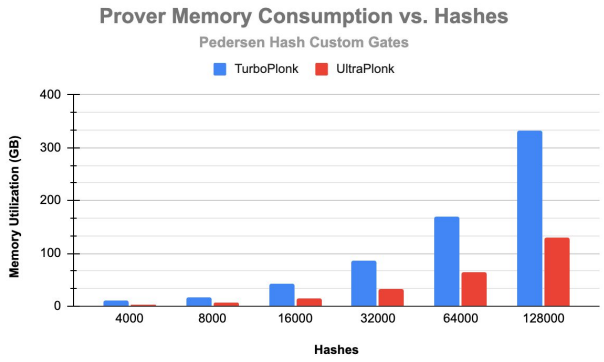

Figure 14: Comparing memory consumption in TurboPLONK and UtraPlonk over an extended range of task sizes.

more expensive than 128K hashes in UltraPLONK in terms of the number of gates, execution time, and memory.

In general, custom gates increase the degree of the identity to be proven, which increases the computational work for the prover. This is in tandem with the reduction in circuit size. For example, the degree of the identity may double, leading to increased computational work for the prover, while the circuit size may be cut by half. This simultaneously increases prover work for the custom gate operation, and reduces the circuit size. If the latter has a greater effect on reducing the prover work than the increase in prover time incurred by the former, this is a net positive effect. Although we have not yet benchmarked this relationship directly, the tradeoffs need to be considered for different applications and use cases.

If we instead compare these proving systems based on circuit size rather than task size (number of hashes), we see that for the same circuit sizes, UltraPLONK exhibits 1.5x larger memory footprint than TurboPLONK for the same number of constraints. When examining the throughput based on the circuit size as a function of the number of hashes it can process, PlonK requires 5113 gates per hash, TurboPLONK requires 345 gates per hash, and UltraPLONK requires 103 gates per hash. For circuits with 2^{25} constraints, PlonK processed 6,562 hashes, TurboPLONK processed 97,259 hashes, and UltraPLONK processed 325,771 hashes. TurboPlonk and UltraPLONK were able to prove 14.8x and 49.6x more hashes than PlonK respectively for the same circuit size. UltraPLONK was able to prove 3.4x more hashes than TurboPLONK. In summary, if the circuit sizes of TurboPLONK and UltraPLONK are the same, they must compute a different number of hashes.

6 PARALLELIZATION AND FUTURE WORK

At present, we are converting Aztec's Barretenberg cryptographic library and backend to be compatible with GPUs[7]. Porting CPU-based code to a GPU framework is not straightforward since computations are not over normal **int** and **float** data types but rather over large finite fields whose range exceeds the capacity of standard numeric datatypes. Since these computations are key to the proof process, they must be implemented with care in regards to performance. As a result, validating the correctness of computations is nontrivial. We have implemented a majority of the underlying mathematical structures over which our proving system operates: primarily finite-field arithmetic and elliptic-curve operations. We use Cuda-Fixnum, a fixed-precision SIMD library that targets CUDA. We are porting Supranational's MSM kernel (Pippenger's Bucket Method[3]) over the bn254 elliptic curve in order to be compatible with our GPU proving system. Once the GPU port of PlonK is complete, we can benchmark it against both the CPU version and Groth16. With a working PlonK GPU implementation in hand, we can extend it to an implementation of TurboPLONK and UltraPLONK.

With a deeper understanding of how the PlonK family behaves in a GPU setting, future work entails seeking higher parallelism with multiple GPUs and devising algorithms for effectively transferring intermediate results among the memories of the GPUs. These memory management issues may, in turn, lead to consideration of alternatives to current algorithms that are more amenable to extreme parallelization. Further improvements appear feasilbe by designing numerical algorithms specifically for GPU computation. This, too, is a component of our future work.

While much future work remains, these approaches offer the prospect of significant extension of practical problem sizes for ZK systems, enabling the deployment of ZK in large scale information management applications that require privacy, transparency, and regulatability.

ACKNOWLEDGMENTS

This work is supported by Oracle Cloud and related resources provided by the Oracle for Research program, and by gifts from Steel Perlot and Google. A Lehigh CORE grant also provided support. We thank Suyash Bagad from Aztec for his insights into the TurboPLONK and UltraPLONK proving systems. We thank Maxim Vezenov from Aztec for his review and comments. We thank dePaul Miller, our colleague in the SSS Lab at Lehigh, for providing his GPU expertise.

Finally, we acknowledge our colleagues, the Scalable Systems and Software Research group (sss.cse.lehigh.edu) and Lehigh Blockchain (blockchain.cse.lehigh.edu).

REFERENCES

[1] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. 2018. Scalable, transparent, and post-quantum secure computational integrity. *IACR Cryptol. ePrint Arch.* (2018), 46. http://eprint.iacr.org/2018/046

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

[3] Daniel J. Bernstein. 2002. Pippenger's Exponentiation Algorithm. Manuscript, University of Illinois at Chicago. https://cr.yp.to/papers/pippenger-20020118-retypeset20220327.pdf.

[4] Dan Boneh and Victor Shoup. 2023. *A Graduate Course in Applied Cryptography, version 0.6.* cryptobook.us.

[5] Joseph Bonneau, Izaak Meckler2 andVanishree Rao2, and Evan Shapiro2. 2020. Mina: Decentralized Cryptocurrency at Scale. Mina White Paper. https://minaprotocol.com/wp-content/uploads/technicalWhitepaper.pdf

[6] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Psi Vesely, and Nicholas Ward. 2019. Marlin: Preprocessing zkSNARKs with Universal and Updatale SRS. Cryptology ePrint Archive, Report 2019/1047. https://eprint.iacr.org/2019/1047.pdf.

[7] Tal Derei. 2022. Cuda-Barretenberg. https://https://github.com/TalDerei/cuda-barretenberg/tree/ff-and-ecc-operations.

[8] Tal Derei and Benjamin Aulenbach. 2022. *Benchmarking Groth16 Proving System.* Technical Report. Lehigh University. https://github.com/TalDerei/Masters-Research/blob/main/Benchmarking%20Groth16%20Proving%20System.pdf.

[9] Tal Derei and Benjamin Aulenbach. 2022. *Benchmarking PlonK Proving System.* Technical Report. Lehigh University. https://github.com/TalDerei/Masters-Research/blob/main/Benchmarking%20PlonK%20Proving%20System.pdf.

[10] Tal Derei, Caleb Geren, Michael Kaufman, Jon Klein, and Rishad Islam Shantho. 2023. *Benchmarking PlonK Proving System.* Technical Report. Lehigh University. https://github.com/TalDerei/Masters-Research/blob/main/Benchmarking%20Plonk%2C%20TurboPlonk%2C%20and%20UltraPlonk%20Proving%20Systems.pdf.

[11] U. S. Federal Reserve and MIT. 2022. Project Hamilton Phase 1: A High Performance Payment Processing System Designed for Central Bank Digital Currencies. Federal Reserve Bank of Boston and Massachusetts Institute of Technology Digital Currency Initiative. https://www.bostonfed.org/-/media/Documents/Project-Hamilton/Project-Hamilton-Phase-1-Whitepaper.pdf.

[12] Ariel Gabizon and Zachary J. Williamson. 2020. plookup: A Simplified Polynomial Protocol for Lookup Tables. Cryptology ePrint Archive, Report 2020/315. https://eprint.iacr.org/2020/315.pdf.

[13] Ariel Gabizon and Zachary J. Williamson. 2020. The Turbo-PLONK Program Syntax for Specifying SNARK Programs. https://docs.zkproof.org/pages/standards/accepted-workshop3/proposal-turbo_plonk.pdf.

[14] Ariel Gabizon, Zachary J. Williamson, and Oana-Madalina Ciobotaru. 2019. PlonK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge. *IACR Cryptol. ePrint Arch.* (2019), 953.

[15] S. Goldwasser, S. Micali, and C. Rackoff. 1985. The Knowledge Complexity of Interactive Proof-Systems. In *Proc. 17th ACM Symp. on the Theory of Computing.*

[16] Noah Grossman. 2022. Algorand State Proofs. Medium. https://medium.com/algorand/algorand-state-proofs-707d64038e35

[17] J. Groth. 2016. On the Size of Pairing-based Non-interactive Arguments. https://doi.org/10.1007/978-3-662-49896-5_11.

[18] Tao Lu, Chengkun Wei, Ruijing Yu, Yi Chen, Li Wang, Chaochao Chen, 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.

[19] 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

[20] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. 2019. Sonic: Zero-Knowledge SNARKs from Linear-Size Universal and Updateable Structured Reference Strings. Cryptology ePrint Archive, Paper 2019/099. https://eprint.iacr.org/2019/099 https://eprint.iacr.org/2019/099.

[21] Ralph C. Merkle. 1988. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology — CRYPTO '87*. Springer Berlin Heidelberg, 369–378.

[22] Valeria Nikolaenko, Sam Ragsdale, Joseph Bonneau, and Dan Boneh. 2022. Powers-of-Tau to the People: Decentralizing Setup Ceremonies. Cryptology ePrint Archive, Paper 2022/1592. https://eprint.iacr.org/2022/1592 https://eprint.iacr.org/2022/1592.

[23] Maksym Petkus. 2019. Why and How zk-SNARK Works: Definitive Explanation. arXiv 1906.07221v1.

[24] Protocol Labs. 2017. FileCoin: A Decentralized Storage Network. Web document. https://filecoin.io/filecoin.pdf.

[25] Justin Thaler. 2023. *Proofs, Arguments, and Zero-Knowledge.* Now Foundation and Trends. https://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.html.

[26] Maxim A. Vezenov. 2022. *Accelerating zkSNARKs on Modern Architectures.* Technical Report. Master's Thesis, Lehigh University.

[27] Charles. F. Xavier. 2022. PipeMSM: Hardware Acceleration for Multi-Scalar Multiplication. Cryptology ePrint Archive, Paper 2022/999. https://eprint.iacr.org/2022/999 https://eprint.iacr.org/2022/999.

[28] 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 *48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2021, Valencia, Spain, June 14-18, 2021*. IEEE, 416–428. https://doi.org/10.1109/ISCA52012.2021.00040.