Accelerating the Plonk zkSNARK Proving System using GPU Architectures

by

Tal Derei

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Computer Science

Lehigh University

May 2023

© Copyright by Tal Derei

All Rights Reserved

DocuSign Envelope ID: 9CD24B7C-19F8-45F4-B130-6E67AAE0FB14

Thesis Signature Sheet

Thesis is accepted and approved in partial fulfillment of the requirements for the Master of Science in Computer Science.

Thesis Title:
Accelerating the PlonK zkSNARK Proving System using GPU Architectures

Name: Tal Derei

LIN: 812376338

5/4/2023
Date Approved

DocuSigned by:
Henry F Korth
Director/Advisor Henry F Korth

Co-Advisor

DocuSigned by:
Brian D. Davison
Department Chair/Second Reader Brian D. Davison

Committee Member

Acknowledgements

I would like to express my sincere appreciation and gratitude to the Computer Science and Engineering department at Lehigh University for all their support and encouragement during my time at Lehigh. I want to acknowledge all the members in my research group, the Scalable Systems and Software (SSS) group, and Lehigh Blockchain who have extended their expertise and enriched my research experience.

I would like to specifically thank Hank Korth who acted as my mentor during my undergraduate degree and advisor in the completion of my master's degree. I first became interested in blockchains from one of the blockchain courses he taught, and I am grateful for his advice and inspiration throughout my academic journey.

I would also like to express my gratitude to Roberto Palmieri and dePaul Miller, our colleagues in the SSS Lab, for sharing their GPU expertise. Thank you to Maxim Vezenov and Benjamin Aulenbach from SSS for your comments and review. Thank you to Suyash Bagad from the Aztec network, Dr. Karthik Inbasekar and Hadar Sackstein from Ingonyama for your ideas and suggestions in this work. Thank you to Tobin Herringshaw for pushing me in my academic studies and encouraging me to pursue my interests early on.

Finally, I want to thank my invaluable family and friends without whom my accomplishments mean nothing. Without your unwavering support and

guidance, I would not have been able to achieve this milestone in my life. To my parents, I will strive to continue to make you proud.

Contents

1 Introduction 2

1.1 Contribution 5

2 Preliminaries 6

2.1 Overview of Universal zkSNARKs 6

2.2 Methods for Proof Generation 8

2.2.1 Plonkish Arithmetization 8

2.2.2 Lagrange Interpolation 12

2.2.3 Quotient Polynomial 13

2.2.4 KZG Polynomial Commitments 16

2.3 Plonk 20

3 Techniques for Proof Acceleration 21

3.1 Number-Theoretic Transform 21

3.2 Multi-Scalar Multiplication 23

3.2.1 Pippenger's Bucket Method 24

3.2.2 Design 27

3.3 Hardware Architectures 31

3.3.1 GPUs, FPGAs, and ASICs 31

3.3.2 GPU Acceleration of Groth16 and Plonk 35

3.4 High-Bandwidth Pipeline 39

4 Architecture Directions and Impact on Future Work **44**

4.1 Unified Memory Architectures 44

5 Concluding Remarks **46**

5.1 Summary 46

5.2 Future Directions 47

6 References **49**

7 Appendix **58**

8 Vita **63**

List of Tables

1 Matrix of Constraints 10

2 Matrix of Polynomials 11

3 Hardware Configuration of Testbeds 36

4 Baseline Implementations 36

List of Figures

1 Groth16 CPU and GPU (A10) performance on proof generation as a function of number of constraints. 58

2 Groth16 system memory consumption for proof generation as a function of number of constraints. Shown for CPU prover, GPU prover (P100), and GPU VRAM. 58

3 Groth16 performance on parameter and preprocessing generation as a function of number of constraints. 59

4 Groth16 memory consumption for parameter and preprocessing generation as a function of number of constraints. 59

5 MSM execution time on GPU (A10 and A40) for the ZPrize Baseline Benchmark. 60

6 MSM memory consumption on GPU (A10 and A40) for the ZPrize Baseline Benchmark. 60

7 Plonk performance of MSM on CPU as a function of number of constraints. 61

8 Plonk memory consumption of MSM on CPU as a function of number of constraints. 61

9 Plonk performance of proof generation on CPU as a function of number of constraints. 62

10 Plonk performance of memory consumption on CPU as a function of number of constraints. 62

Accelerating the Plonk zkSNARK Proving System using GPU Architectures

Tal Derei

May 16, 2023

Abstract

Zero-knowledge proofs (ZKPs) are cryptographic primitives that enable a prover to convince a verifier that a statement about some secret is true without leaking information about the secret. The prover ultimately does not reveal anything beyond the validity of the statement itself to the verifier. In recent years, blockchains have been leveraging the properties of zero-knowledge proofs for private transactions and verifiable outsourced computation, leading to improved privacy and scalability for various systems and applications. The specific problem is that the cost of generating proofs is a computationally intensive task which remains a bottleneck for proving systems, while verifying proofs is computationally inexpensive and fast. However, the proof generation process is primarily dominated by operations such as multi-scalar multiplication (MSM) and number theoretic transforms (NTT), which happen to be embarrassingly parallel. This paper describes the hardware acceleration techniques and challenges for constructing zero-knowledge proofs using NVIDIA GPUs in the CUDA programming language.

1 Introduction

Zero-knowledge proofs enable one party (the prover) to convince another party (the verifier) they possess confidential information without revealing said information. For instance, suppose the prover claims to know the pre-image w of a public hash function H that evaluates to y such that $H(w, x) = y$, where witness w and x are private and public inputs respectively. The prover can convince the verifier of this claim without leaking information about the private input w. Without ZKPs, the prover would need to disclose sensitive information like w with the verifier, who can check the prover's result by recomputing the hash on w and comparing the outputs. This naive approach is neither succinct nor work-saving since the verifier needs to perform work proportional to the size of the original problem in order to validate the result. The inputs to the program may be intractably large, and recomputing the hash can be a compute-intensive task.

With ZKPs, the verifier can validate the correctness of the result without explicitly recomputing the hash. Instead, the prover generates a succinct zero-knowledge proof π that shows y was computed correctly, where the size of π is small on the order of a few hundred bytes or kilobytes. The size of π is logarithmic or constant with respect to the problem size depending on the proof system. The verification time for verifying π is in the range of a few milliseconds [1].

The notion of fast verification, independent of the complexity of the orig-

inal computation, has enabled many practical use cases such as outsourced verifiable computation, private transactions, decentralized storage, trustless cross-chain bridges, and more. Notably, scaling solutions like zkRollups [2] are designed to increase the transaction throughput of Ethereum by outsourcing transaction processing and computation off-chain. Every proof generated off-chain proves the correct execution of a transaction in a smart contract, and many proofs can be batched together and composed into a single proof by recursively aggregating them in a Merkle-tree [3]. The final proof represents the correct execution of a batch of transactions, usually in the thousands, and is verified on-chain. The verifier is encoded as a smart contract that performs the lightweight proof verification and state updates.

Mina [4] applies these compression techniques to maintain a fixed-size, 22 kB, blockchain that compresses blocks of transactions into a single proof that validates the entire history of the chain. This allows light clients to easily synchronize to the latest state of the chain and verify state transitions. Cryptocurrency exchanges like Binance have also been experimenting with zero-knowledge for their proof-of-reserve system [5], enabling them to prove their financial solvency and safe custody of user assets.

In addition to applications related to scalability and compliance, a third use case has been gaining popularity; private payments on public blockchains. Zcash [6] emerged as one of the first private cryptocurrencies based on shielded addresses where the sender and receiver addresses and transaction amounts are not publicly visible. Similarly, Aztec [6] is a private zkRollup that sup-

ports encrypted payments by logically separating client-side and server-side computation. A weak client proves the correctness of a single private transaction by generating a proof locally on its machine, and a powerful server batches these proofs into blocks and validates them for correctness. More recently, privacy-preserving payment systems that emerged from Zcash have been expanding to support general private computation. Aleo [7] for example is an encrypted blockchain that supports an environment for private smart contract execution.

These use cases are a few examples of how zero-knowledge proofs can be used, with numerous other possibilities being developed. The domain of zero-knowledge applications has been steadily growing in complexity, and consequently, in size. To create a ZKP of a given program, the program is codified as an arithmetic circuit with gates and wires. The size of the circuit is quantified by the number of constraints, which define the number of logical operations in the program. At the current rate of exponential growth, the highest number of constraints in an application circuit is estimated to reach 2^{40} by 2032 [8]. In 2016, Zcash's private payments circuit was 1 million, approximately 2^{20}, constraints and required 30 seconds to generate a proof [9]. In 2023, Polygon's zkRollup circuit [10] requires 127 million, approximately 2^{27}, constraints for generating a STARK proof for a batch size of nearly 500 transactions in 2 minutes using a 192-core CPU and 512 GB of memory. The steady increase in the number of constraints demands more computational work to construct a proof, resulting in longer proof generation times. The

use of specialized parallel architectures like GPUs, FPGAs, and ASICs can dramatically accelerate this process to support larger circuits, lower proof latency, and cheaper proofs. Hardware acceleration ultimately reduces the prover cost, defined by the computational cost per transaction, by generating more proofs in parallel in a shorter period of time.

1.1 Contribution

This work describes a minimal implementation, Cuda-Barretenberg [11], of an existing general-purpose zero-knowledge proof construction, Plonk, that takes advantage of GPU hardware acceleration. To accomplish this, Aztec's Barretenberg cryptographic library and backend [12] is modified to be compatible with GPUs. Barretenberg includes a C++ implementation of Plonk, while most other implementations are written in Rust. For this reason, Barretenberg was chosen as the focus for GPU acceleration. The majority of the underlying mathematical structures the proving system operates over, primarily finite field arithmetic and elliptic curve operations, are implemented on the GPU. The implementation references Cuda-Fixnum [13], a Bignum fixed-precision SIMD library that targets CUDA, for implementing the multi-threaded field and curve arithmetic used in computing the MSM and NTT operations. The multi-scalar multiplication kernel for Pippenger's Bucket Method is based on the algorithms used in PipeMSM [14], and was ported over the BN-254 elliptic curve in order to be compatible with the Barretenberg's proving system. Moreover, a wrapper was created to override the

existing Barretenberg queue with a carefully designed CUDA pipeline to perform the proof generation. We further explore unified-memory architectures in order to address memory concerns for programs with exceptionally large constraint sizes. As of the publication of this thesis, efforts to port the complete prover are still ongoing.

2 Preliminaries

2.1 Overview of Universal zkSNARKs

Zero-knowledge systems require a preprocessing step referred to as a *Trusted Setup Ceremony*. The procedure coordinates a distributed multi-party computation among a group of N independent participants. This involves a round-robin computation where each participant P_i receives a message M_i from the previous participant P_{i-1}, contributes their randomness to generate a new message M_i', and passes M_i' to the next participant P_{i+1}. The final output is a set of public parameters known as a Structured Reference String (SRS). The SRS consists of the prover and verifier keys, S_p and S_v, for generating and verifying proofs. The setup ceremony has a 1-of-N trust assumption, and is considered secure as long as at least one participant in the setup has securely destroyed their contributed randomness.

The widely used, but older, Groth16 [15] system requires a new setup, $S(C, r) \rightarrow (S_p, S_v)$, for each circuit C. Per-circuit trusted setups are unfavorable because even minor changes to an existing circuit require coordinating

a new setup. The more recent Plonk proving system [16] requires just one setup, a universal trusted setup denoted by $S = (S_{init}, S_{index})$, that can then be reused across any set of circuits. This means it can be used to create proofs for any computation, rather than being specific to a particular circuit. The universal trusted setup is comprised of a two-step procedure:

- $S_{init}(\lambda,\ r) \rightarrow PP$.

  S_{init} represents the randomized algorithm for generating the generic public parameters that can be used for all circuits, where the randomness r must be kept secret from the prover. In 2017, the Power's of Tau setup ceremony [17] generated these public parameters PP commonly used in industry.

- $S_{index}(PP,\ C) \rightarrow (S_p, S_v)$.

  S_{index} is a circuit specific step for generating S_p and S_v, where r is independent of the circuit C. Since the verifier can only operate in constant or logarithmic time with respect to C, preprocessing the circuit in this way creates a short summary of C, S_v, that's easily readable by the verifier.

Zero-knowledge proofs posses several properties:

- ***Completeness*** – A honest prover must be able to prove true statements and convince a verifier.

- ***Soundness*** – A prover cannot convince a verifier of false statements and produce invalid proofs.

- ***Zero-Knowledge*** – The circuit, prover and verifier keys, and proof reveal nothing about the private witness.

Succinct Non-Interactive Arguments of Knowledge (SNARKs), such as Groth16 and Plonk, are popular zero-knowledge proof systems that exhibit two additional succinctness properties.

- ***Small Proof*** – $P(S_p, x, w) \rightarrow$ small proof π.
- ***Fast Verification*** – $V(S_v, x, \pi) \rightarrow$ accept or reject π.

2.2 Methods for Proof Generation

We discuss the arithmetization scheme for Plonk, which converts a circuit into a collection of polynomials. This is loosely termed *Plonkish Arithmetization*. Compared to Groth16's constraint format known as *R1CS*, Plonk is more flexible by supporting constraints of degree larger than two.

2.2.1 Plonkish Arithmetization

A program P must first be converted into an arithmetic circuit C composed of gates and wires. The circuit is internally structured as a Directed Acyclic Graph (DAG). The gates in the circuit represent either addition or multiplication operations and each gate supports two input wires and a single output wire. Once the circuit is constructed, the *arithmetization* process transforms it into a structured system of polynomial equations. In this process, each gate in the circuit is associated with a separate equation represented by a

low-degree polynomial. The set of polynomials that comprises the computation is known as the *constraint system.* Each equation in the constraint system is modeled by a general form:

$$(Q_{L_i})a_i + (Q_{R_i})b_i + (Q_{O_i})c_i + (Q_{M_i})a_ib_i + (Q_{C_i}) = 0 \quad (1)$$

where a_i, b_i, $c_i \in \mathbb{F}_{\mathbb{P}}$ are the left input, right input, and output *wires* of the i_{th} gate in circuit C, respectively, and the finite field $\mathbb{F}_{\mathbb{P}}$ is of prime order $\mathbb{P}$. The wires are assignments to the circuit and represent the prover's private witness values. The boolean *selectors* Q_{L_i}, Q_{R_i}, Q_{O_i}, Q_{M_i}, $Q_{C_i} \in \{0, 1\}$ represent the i_{th} gate and can be toggled on or off to activate a specific gate. The selectors encode the structure of the circuit. For instance, an addition gate is represented as $Q_{L_i} = 1$, $Q_{R_i} = 1$, $Q_{M_i} = 0$, $Q_{0_i} = -1$, and $Q_{C_i} = 0$, which yields the constraint $a_i + b_i = c_i$. This can be rewritten as $a_i + b_i$ - $c_i = 0$, where the sum of the input wires has to equal the output wire. This constraint needs to be satisfied for that specific gate.

For a circuit with n gates, n constraints must satisfied. Arithmetic circuits must adhere to two types of constraints: gate constraints and copy constraints. *Gate constraints* enforce the relationship between wires attached to the same gate, for example $a_i * b_i = c_i$ for a single multiplication gate. *Copy constraints* make consistency claims about wires attached to different gates in the circuit, for example $a_1 = c_3$ where the left wire of the first gate is the same as the output wire of the third gate. Said differently, copy constraints

connect wires between disconnected gates and force them to be equal. The constraints are in place to ensure the gates and wires are evaluated correctly.

We can more easily represent these constraints in matrix form. The constraint system of a circuit is modeled as a 2D matrix of rows and columns, which contains the result of executing the computation and placing intermediary wire and gates values into the correct cells.

Table 1: Matrix of Constraints

a_n	b_n	c_n	Q_L	Q_R	Q_M	Q_0	Q_C
$a_{(1)}$	$b_{(1)}$	$c_{(1)}$	1	1	0	1	0
$a_{(2)}$	$b_{(2)}$	$c_{(2)}$	0	0	1	1	0
$a_{(3)}$	$b_{(3)}$	$c_{(3)}$	1	1	0	1	0
...	...	...	...	...	...	...	...
$a_{(n)}$	$b_{(n)}$	$c_{(n)}$	0	0	1	1	0

The matrix in Table 1 consists of finite field elements. Each column can be considered a length-n vector of field elements, and each row encodes a distinct constraint in Table 1. Equation (1) can then be rewritten in polynomial form:

$$Q_L(x)A(x) + Q_R(x)B(x) + Q_O(x)C(x) + Q_M(x)A(x)B(x) + Q_C(x) = 0 \quad (2)$$

Here, the vectors a_n, b_n, c_n, Q_L, Q_R, Q_M, Q_O, Q_C are now represented in a single polynomial expression, where $A(x)$, $B(x)$, $C(x)$ are *wire polynomials*, and $Q_L(x)$, $Q_R(x)$, $Q_M(x)$, $Q_O(x)$, $Q_C(x)$ are *gate polynomials* [18]. Wire polynomials encode the user's input, gate polynomials encode the gate con-

straints. Permutation polynomials σ, emitted from the table below, encode the copy constraints. Each column vector has been reformulated as a large-degree polynomial of degree $n-1$.

Table 2: Matrix of Polynomials

A(x)	B(x)	C(x)	Q$_L(x)$	Q$_R(x)$	Q$_M(x)$	Q$_0(x)$	Q$_C(x)$
$A(x^1)$	$B(x^1)$	$C(x^1)$	1	1	0	1	0
$A(x^2)$	$B(x^2)$	$C(x^2)$	0	0	1	1	0
$A(x^3)$	$B(x^3)$	$C(x^3)$	1	1	0	1	0
...	...	...	...	...	...	...	...
$A(x^n)$	$B(x^n)$	$C(x^n)$	0	0	1	1	0

In order to represent the column vector a_n as a polynomial $A(x)$ that passes through all of the points in a_n, *Lagrange Interpolation* is used. Consider for instance the column A in the table where the ith element of A corresponds to the evaluation of $A(x^i) = a_i$, where $x^i \in \mathbb{F}_{\mathbb{P}}$. This means that evaluating the polynomial A at a specific x^i will yield the vector value a_i for each cell in column A. The set of x^i's is known as the evaluation domain H, which is the set of points that the polynomial $A(x)$ is evaluated over. More precisely, $H = \{\omega_0, \omega_1, ..., \omega_i\}$ is the set of nth *roots of unity*, where $\omega^n = 1$. We associate each row i with a unique root of unity ω_i, and each point in polynomial $A(x)$ is represented in evaluation form by evaluating it on ω^i [19].

2.2.2 Lagrange Interpolation

Lagrange Interpolation is a method where given a set of n arbitrary points, it constructs a polynomial $F(x)$ that passes through all points. Said another way, for an arbitrary n-dimensional vector of field elements $v \in \mathbb{F}_{\mathbb{P}}$, there exists a unique univariate polynomial $F(x)$ with degree $n-1$ that passes through the points (i, v_i) for $i = 1, 2, ..., n$. Lagrange interpolation is a two-step procedure [20]:

1. Construct a *Lagrange Basis*, which is a set of n polynomials of degree $n-1$. Each polynomial $L_i(x)$ evaluates to 0 at all points except for one, where it evaluates to 1.

$$L_i(x) = \begin{cases} 0, if\ x = x_j,\ j \in [n],\ j \neq i \\ 1,\ if\ x = x_i \end{cases} \tag{3}$$

The individual polynomials $L_i(x)$ can be constructed as follows.

$$L_i(x) = \prod_{0 \leq j < n, j \neq i} \frac{x - x_j}{x_i - x_j} \tag{4}$$

At this point, we have n polynomials $L_0(x)$, $L_1(x)$, ..., $L_n(x)$. Each of these polynomials is evaluated over the roots of unity $\mathrm{H} = \{\omega^0, \omega^1, ..., \omega^n\}$. Consider for example the unique polynomial $L_0(x)$ which can be decomposed into its component parts $L_0(\omega^0) = 1$, $L_0(\omega^1) = 0$, ..., $L_0(\omega^n)$ $= 0$ respectively. Here the 0th lagrange polynomial evaluates to 1 on

the 0th root of unity, and 0 at every other point.

2. Scale each lagrange polynomial $L_i(x)$ to a target value and sum them together to express the final polynomial $F(x)$.

$$F(x) = \sum_{i=0}^{n-1} Y_i * L_i(x) \tag{5}$$

For each column in the matrix table, the vector is converted into a polynomial. However, there is a more efficient method called Number Theoretic Transforms, which we discuss in section 4.2.

2.2.3 Quotient Polynomial

The set of polynomials discussed capture the execution trace of the original computation. In order to prove the execution trace is valid, all the constraints arising from the circuit must be satisfied. Thus far, we have converted the n individual constraint equations,

$$Q_L(\omega_i)a_i + Q_R(\omega_i)b_i + Q_O(\omega_i)c_i + Q_M(\omega_i)a_ib_i + Q_{C_i} = 0$$

into a single general polynomial equation that can be formally expressed as,

$$Q_L(x)A(x) + Q_R(x)B(x) + Q_O(x) + Q_M(x)A(x)B(x) + Q_c(x) = \psi_i(x) \tag{6}$$

The set of constraints is denoted by $\psi_i(x)$, and $\psi_i(x) = 0$ for all $X \in$ the domain $H = \{\omega_0, \omega_1, ..., \omega_n\}$ in order for the constraints to hold.

We have n constraint polynomials $\psi_0(x)$, $\psi_1(x)$, ..., $\psi_{n-1}(x)$ that need to evaluate to 0. As the execution trace of the computation becomes exceedingly large, checking that each individual constraint holds becomes computationally challenging. Instead of checking each constraint individually, the constraint checks can be compressed into a single check. We batch them together into a single constraint polynomial, denoted by $\psi(x)$, by sampling a random field element $\delta \in F_p$ and then taking a random linear combination of the individual constraints [21]:

$$\psi(x) = \delta^0 * \psi_0 x + \delta^1 * \psi_1(x) ... \delta^{n-1} * \psi_{n-1}(x) \tag{7}$$

The constraint $\psi(x)$ evaluates to 0 over the domain if all the individual constraints do. Therefore, the constraints satisfied for each row means that $\psi(\omega_i) = 0$ for all inputs.

So far we have compressed all the constraints into a single polynomial $\psi(x)$. To prove $\psi(x)$ holds for each row in the matrix table with a single check, we can derive an equivalent statement which is computationally easy to check:

$$\psi(x) = Z(x) * Q(x) \tag{8}$$

where $Q(x)$ is known as the *quotient polynomial* and $Z(x)$ is known as the *vanishing polynomial*. The vanishing polynomial is shorthand for $x^n - 1$, which can be expanded and evaluates equivalently to $(x - \omega^1)(x - \omega^2)...(x -$

ω^n). This represents the minimal polynomial that evaluates to 0 for each ω^n in the evaluation domain since it includes elements of H as roots.

$$Z(x) = \prod_{i \in [n]} (x - \omega_i) \tag{9}$$

We can verify the correctness by performing the division $\frac{\psi(x)}{Z(x)}$. The constraint system is satisfied if $\psi(X)$ is divisible by $Z(x)$, and this allows us to perform n constraint checks using a single polynomial division check [17]. The quotient polynomial can be rewritten and computed as follow.

$$Q(x) = \frac{\psi(x)}{x^n - 1} = \frac{\delta^0 * \psi_0(x) + \delta^1 * \psi_1(x) ... \delta^{n-1} * \psi_{n-1}(x)}{x^n - 1} \tag{10}$$

In summary, zkSNARKs like Groth16 and Plonk are protocols to prove a polynomial $\psi(x) = 0 \ \forall \ x \in H$, where the domain $H = \{\omega_0, \omega_1, ..., \omega_n\}$. The prover is ultimately trying to prove to the verifier that the polynomial is vanishing in the domain of H. Proving that statement is equivalent to proving that polynomial is divisible by the vanishing polynomial $Z(x)$ which includes elements of H as roots. This equivalently reduces down to instead proving that $\psi(x) = Q(x) \cdot Z(x) \ \forall \ x \in H$. The prover then commits to polynomials $\psi(x)$ and $Q(x)$ and sends them to the verifier. Instead of the verifier naively checking the polynomial over the entire domain H, a more succinct verification method exists. The verifier samples a random field element r and the prover evaluates two polynomials $\psi(r)$ and $Z(r)$ at that random point.

Prover sends back f(z) and t(z) evaluations and proof that these evalaution is correct. The verifier checks whether $\psi(r) = Z(r) * Q(r)$ [22].

The polynomials described here, however, are too large to send to the verifier. The Plonk protocol uses a *commitment scheme*, discussed in the next section, for compressing the polynomials into succinct representations called commitments that can be efficiently evaluated by the verifier.

2.2.4 KZG Polynomial Commitments

In zkSNARKs, a polynomial commitment scheme is a cryptographic protocol that enables a small, $O(1)$ sized-object to represent an arbitrarily large $O(N)$-sized object like a large polynomial of degree d. The degree d can range in the hundreds of millions for large circuits. The Kate Polynomial Commitment scheme **(KZG10)** was introduced by Kate, Zaverucha and Goldberg [23] in 2010. KZG10 has strong security properties, enabling proving systems to have a constant proof size and verification time, regardless of the proven statement's size.

- ***Hiding*** - A prover commits to a large secret polynomial P of degree d, where the commitment C represents P as a single elliptic curve point. The commitment hides P such that the verifier does not learn anything about the original polynomial P. The prover can later *open* the committed polynomial at a random evaluation point γ chosen by the verifier, for example showing $P(\gamma) = y$ without leaking information about P.

- ***Binding*** - The prover cannot cheat by changing the original polynomial P after sending the commitment C to the verifier.

The KGZ protocol provides a way for a prover to succinctly commit to a set of large-degree polynomials and produce a small proof that verifies the correct execution of these polynomials. The prover starts by sending the verifier a commitment C associated with polynomial P. The verifier in turn can only ask the prover to check the commitment at a single random challenge point γ. The prover opens the commitment C at γ, outputting the evaluation $P(\gamma) = y$. The prover then generates a proof of knowledge π that shows the commitment to the polynomial is valid and the original polynomial P evaluated at γ, $P(\gamma)$, yields the correct output. These steps are described in greater detail as follows.

1. ***Setup*** – A Trusted Setup ceremony is performed to generate the public parameters used in the commitment scheme. During the setup phase, a random element α is sampled from the prime finite field $\mathbb{F}_{\mathbb{P}}$. The public parameters that comprise the Structured Reference String (SRS) are represented by $\{H_0 = G, H_1 = G^{\alpha}, H_2 = G^{\alpha^2}, ..., H_d = G^{\alpha^d}\}$. G is a generator, and there are $d + 1$ elements comprising the public parameters [1]. The parameter d represents the upper bound for the problem size supported by the trusted setup. For Aztec's ignition ceremony, d is around 100 million constraints.

2. ***Generate Commitment*** – The objective is for the prover to compute

the commitment C to a polynomial P evaluated at the secret point α, where $C = G^{P(\alpha)} \in$ G. The resulting commitment C is a single curve element. However, since α was deleted, the prover cannot directly compute $G^{P(\alpha)}$ and uses the public parameters PP to commit to the polynomial instead. Let $P(x) = P_0 + P_1X + ... + P_dX^d$, then the commitment C to polynomial P is computed by $C = P_0H_0 + ... + P_dH_d = P_0G + P_1G^{\alpha} + P_2G^{\alpha^2} + ... + P_dG^{\alpha^d} = G^{P(\alpha)}$ Generating a commitment to an arbitrary polynomial can be reformulated as:

$$P(x) = \sum_{i=0}^{n-1} a_i X^i \tag{11}$$

where a_i are the coefficients of the polynomial P and X^i are values from the SRS public parameters. To perform this operation, the multi-scalar multiplication algorithm detailed in Section 3.2 is required.

3. ***Generate Proof*** – The prover needs to demonstrate to the verifier the polynomial P evaluated at a random evaluation point γ chosen by the verifier yields the correct output, $P(\gamma) = y$. Equivalently, we can define another polynomial $G(x) = P(x) - y$ with root at $x = \gamma$ which evaluates to 0. Since $G(x)$ is divisible by the linear polynomial $(X - \gamma)$, there must exist a quotient polynomial $Q(x)$ such that $Q(x) = \frac{P(x)-y}{X-\gamma}$. Therefore, the existence of $(X - \gamma)$ as a factor of $P(x) - y$ confirms $P(\gamma) = y$. $Q(x)$ exists if and only if $P(\gamma) = y$, and its existence serves as the proof.

After computing the quotient polynomial $Q(x)$, the prover generates a commitment C' to the quotient polynomial and sends it to the verifier. The commitment C' represents the evaluation proof π [24]. Importantly, the size of the proof is a single curve element and is independent of the degree d of the polynomial.

$$\pi := C' \in G \tag{12}$$

4. ***Verify Proof*** – The verifier uses elliptic curve pairings to check (1) the prover has evaluated the polynomial P correctly, and (2) the commitment C is associated with the Polynomial P. The verifier either accepts or rejects the proof π.

In summary, the prover will commit to wire polynomials $A(x)$, $B(x)$, and $C(x)$ that encode the user's private inputs and gate polynomials $Q_L(X)$, $Q_R(X)$, $Q_M(X)$, $Q_O(X)$, $Q_C(X)$ that encode the structure of the circuit. The prover will also commit to permutation polynomial $\sigma_1(X)$, $\sigma_2(X)$, $\sigma_3(X)$ that encodes all copy constraints to ensure consistency between wires of different gates. The quotient polynomial $Q(x)$ summarizes the entire circuit, and computing the commitment to $Q(x)$ along with an evaluation proof serves as the proof π.

2.3 Plonk

Plonk (Permutations over Lagrange-bases for Oecumenical Non-Interactive Arguments of Knowledge) is a newer zkSNARK construction developed at Aztec in 2019 by Ariel Gabizon, Zachary J. Williamson, Oana Ciobotaru [16]. Plonk is a multi-round zero-knowledge protocol between a prover and verifier with a universal and updatable trusted setup. The proof generation process progress in five rounds, producing a message at every round. The messages collectively make up the proof. Plonk requires calculating 9 scalar multiplications which domainate the prover's work. Additionally, the proof size is 9 G1 group elements for a total of approximately 400 bytes and 6 milliseconds to verify.

In recent years, more performant proving systems like TurboPlonk and UltraPlonk have been developed. TurboPlonk generalizes the constraint system by introducing custom gates that represent complicated statements with fewer gates, for expressing the same computation, in a circuit [7]. Custom gates represent more complex operations in a single gate, reducing the number of total gates in the circuit. UltraPlonk extends this TurboPlonk construction with precomputed lookup tables, enabling a prover to prove a witness exists in a table instead of proving the computation itself, which ultimately reduces the circuit size.

Our experiments pertain specifically to standard Plonk and the techniques for accelerating the prover will be discussed in the next sections.

3 Techniques for Proof Acceleration

Most SNARKs rely on similar primitives for generating proofs: multi-scalar multiplication (MSM) over elliptic curves and number theoretic transformations (NTTs) over large finite fields. MSM involves multiplications between large vectors and dominates approximately 85% of the proof-generation time in Plonk, and NTTs involve complex polynomial calculations and dominate approximately 10% of the run-time [14]. MSMs are highly parallelizable and have predictable memory access patterns, but require a high amount of raw computation and memory. NTTs frequently shuffle data during the runtime, have non-uniform memory access, and require high memory bandwidth [25]. Implementing conventional algorithms like Pippenger's Bucket Method [26] for multi-exponentiation or Cooley–Tukey algorithm for NTTs [27] on parallel architectures like GPUs can dramatically accelerate these operations by running them over thousands of CUDA cores. These techniques will be discussed in more detail in Sections 3.1 and 3.2. Section 3.3 will discuss the specific parallel hardware architectures, including GPUs, FPGAs, and ASICs on which these operations are performed, and a discussion of prior works. Section 3.4 will discuss our GPU-based Plonk implementation.

3.1 Number-Theoretic Transform

Recall the wire polynomials $A(x)$, $B(x)$, $C(x)$ are constructed by performing a series of *Lagrange Interpolations*. Recall given a set of n points, La-

grange Interpolation constructs a polynomial $F(x)$ of degree $N-1$ that passes through all of them. This algorithm incurs a deficient $O(n^2)$ time complexity, prompting Number Theoretic Transforms (NTTs) with $O(nlogn)$ complexity to be employed instead.

The Number-Theoretic Transfer (NTT) algorithm is a variation of the Fast-Fourier Transform (FFT) algorithm that enables efficient conversions between different polynomial representations like coefficient and evaluation forms. These conversions are necessary because arithmetic like multiplications among large polynomials are more amenable in evaluation form since they can be performed in log-linear time. While traditional FFTs are defined over complex numbers, NTTs are defined over finite fields F with a multiplicative subgroup H of order $n = 2^k$. The subgroup H corresponds to the domain consisting of the nth roots of unity $H = \{\omega^0, \omega^1, ..., \omega^n\}$ and are known as the twiddle factors. The most common algorithm for performing these conversions is Cooley-Tukey which uses a divide-and-conquer approach to recursively decompose the NTT of size N into smaller size $\frac{N}{2}$ by performing a butterfly operation at every stage. Formally we can define NTT as two N-sized vectors X and Y where $X_i = \sum_{j=0}^{N-1} Y_j w_N^{ij}\ (mod\ P)$, and x is the size of the input vector, and w are the twiddle factors. The inverse NTT is defined as $Y_i = N^{-1} \sum_{j=0}^{N-1} X_j w_N^{-ij}\ (mod\ P)$. NTTs are necessary for converting a polynomial from coefficient form to evaluation form in order to perform the convolution, and then inverse NTT are used to convert back to polynomial coefficient form.

NTTs suffer from bad memory access patterns due to the butterfly computation. Memory-optimized FFT algorithms like gNTT [28] have been developed that significantly reduce the memory accesses required to compute an NTT by a logarithmic factor. The twiddle factors are re-ordered to avoid random access to memory. Other techniques [29] use a smaller field size like Goldilocks [30], which uses a finite field with prime $P = 2^{64} - 2^{32} - 1$, to minimize the amount of storage required to store points and corresponding twiddles. PipeZK, an ASIC ZK implementation uses techniques to decompose large NTTs into smaller ones. This can reduce 2^{28} NTTs into 2^{14} NTTs, dramatically reducing bandwidth requirements [31].

3.2 Multi-Scalar Multiplication

Multi-scalar multiplication (MSM) is defined as follows. Given an elliptic curve group G of prime order P, let $G = [G_0, G_1, ..., G_{N-1} \in G^N]$ and $x = [x_0, x_1, ..., x_{N-1} \in \mathbb{F_P}]$ be N-element vectors of elliptic curve points and scalars. Then the MSM can be formally defined as:

$$MSM(G, x) = \sum_{i=0}^{N-1} x_i \cdot G_i \tag{13}$$

which expands to $x_0 \cdot G_0 + x_1 \cdot G_1 + ... + x_{N-1} \cdot G_{N-1}$. The problem with this scheme is that computing a single scalar multiplication, $x_i \cdot G_i$, involves performing an elliptic curve addition of G_i to itself x_i times. These arithmetic operations are expensive since elliptic curve points are composed

of large finite field elements on the order of usually 254-bit or 381-bits. Curve additions also decompose into many expensive modular field multiplication operations.

The naive method for computing the MSM is using the *Double-and-Add* algorithm, where the number of group operations scales like $384 \cdot N$, where N is the number of points. For large circuits with many constraints, N can be 2^{26} or more. The objective for computing the MSM is minimizing the number of group operations as a function of the problem size N. Pippenger's Bucket Method described in the next section reduces this scalar factor down to $16 \cdot N$, resulting in a $24x$ improvement.

3.2.1 Pippenger's Bucket Method

We implement the current state-of-the-art algorithm for computing an MSM, Pippenger's Bucket Method [26]. Pippenger introduces a windowing technique that breaks up each $x_i \cdot G_i$ problem into K windows of size c such that $x_i \cdot G_i = x_{i_0}G_i + x_{i_1}2^cG_i + x_{i_2}2^{2c}G_i + ... + x_{i_{k-1}}2^{c(k-1)}G_i$ [32]. This can be formally written as:

$$G = \sum_{i=0}^{N-1} \sum_{k=0}^{K-1} 2^{kc} x_i^{[k]} \cdot G_i \tag{14}$$

The equation describes a *scalar decomposition* where each scalar x_i of size b-bits (254-bits for BN-254) is partitioned into K smaller sub-scalars of size c-bits, where $K = \lceil \frac{b}{c} \rceil$ windows. The sub-scalars are denoted by $x_i^{[k]}$, which

represents the kth partition of scalar x_i. The choice of c is variable to optimize performance for the specific MSM implementation used and input size. To provide a concrete example, consider $b = 254$-bits, $c = 16$-bits, and $K = \lceil \frac{b}{c} \rceil$ = 16 windows. Each window will have 2^c - 1 = 65535 buckets, for a total of $(2^c - 1) \cdot K$ or approximately 1 million buckets across all windows. We can change the order of the summations and rewrite the equation above as

$$G = \sum_{k=0}^{K-1} 2^{kc} B_k \tag{15}$$

where $G^{[k]} = \sum_{i=0}^{N-1} x_i^{[k]} \cdot G_i$. After decomposing the scalars into sub-scalars, we can focus on computing each B_k efficiently. Each B_k is called a *Partial Sum*, and represents a multi-scalar multiplication with b-bit scalars. There are K partial sums in total, one for each window. At this stage, the MSM is logically broken up into 3 phases:

1. ***Phase 1: Bucket Accumulation*** – In order to compute K independent partial sums B_k, start by iterating over all the elliptic curve points G_i. Associate each c-bit scalar with a curve point, and place each point into the appropriate bucket. After distributing all the points into buckets, accumulate the points inside each respective bucket. The result of each bucket will be a single elliptic curve point after performing the addition. Recall there are K windows, and $2^c - 1$ buckets in each window. This step is inherently parallelizable as for each bucket, we can assign a different thread to perform the bucket accumulation step.

2. ***Phase 2: Bucket Aggregation*** – After accumulating all the points in each bucket, we need to sum up all the individual buckets together for each window. In order to add up all the buckets, each bucket needs to be multiplied by its bucket index. This step is performed separately for each window.

$$G^{[k]} = \sum_{i=0}^{N-1} x_i^{[k]} \cdot G_i = \sum_{l=0}^{2^c-1} W_l^{[k]} \cdot l \tag{16}$$

Let $W_l^{[k]}$ represent the bucket sum for a single bucket l in window k. This expands to the series $1 \cdot W_{l_1} + 2 \cdot W_{l_2} + ... + 2^c - 1 \cdot W_l$. This seemingly easy looking problem looks exactly like our initial MSM problem, but only smaller. To calculate this series, we can perform the *Running Sum* method, which is equivalent to the equation above. Continuing with our earlier example of c = 16,

$$\begin{aligned} T_{j_1} &= W_{j_{65535}} \\ T_{j_2} &= T_{j_1} + W_{j_{65534}} \\ &... \\ T_{j_{65535}} &= T_{j_{65534}} + W_{j_1} \end{aligned}$$

This can formally be written as,

$$G^{[k]} = \sum_{i=0}^{2^c-1} T_{j_i} \tag{17}$$

This is a highly serial operation. A more efficient segmented version is described in 3.2.2.

3. ***Phase 3: Result Aggregation*** – Now we need to perform the final accumulation of all the partial sums across the windows. Each window will now have a single partial sum represented as a single elliptic curve point. Since there are K windows, with a single partial sum each, K point additions are performed, resulting in a single elliptic curve point. This step is usually performed on the CPU.

3.2.2 Design

Our MSM design implements the Pippenger algorithms described in PipeMSM [14] [33] to enable a higher degree of parallelism. Although some of algorithms' design choices were developed specifically for the FPGA, we can leverage their parallel construction over CUDA cores. The general MSM is specifically bottlenecked by the bucket aggregation step described in phase 2, which is an inherently sequential workload. It requires K sequential computations, allowing for only $2K$ parallel additions to be performed a time. Our design enables computing the K partial sums in parallel. The implementation uses a *segmented* version of the buckets by breaking up the buckets in each window into M segments. For example, we can break 2^c buckets in the K_{th} window into M segments of size U, where $U = \frac{2^c}{M}$. We show this step for a single window below.

$$
\begin{aligned}
G^{[k]} = & \left(1 \cdot W_1^{[k]} + 2 \cdot W_2^{[k]} + ... + U \cdot W_U^{[k]}\right) \\
& + \left((U+1) \cdot W_{U+1}^{[k]} + ... + 2U \cdot W_{2U}^{[k]}\right) \\
& + ... \\
& + \left((U(M-1)) \cdot W_{U(M-1)+1}^{[k]} + ... + 2^c - 1 \cdot W_{2^c-1}^{[k]}\right)
\end{aligned} \tag{18}
$$

Although this leads to more computation upon recombining the results, the calculations are performed in parallel. The implementation leverages the BN-254 curve, which is supported by Ethereum, along with projective coordinates for cheaper curve additions since they avoid expensive inversions. The MSM also uses a modified version of a highly optimized Bignum finite field library, Cuda-Fixnum [13], to accelerate basic operations like modular multiplication and large integer additions. I modified the bignum library to support the BN-254 and Grumpkin curves, projective coordinates, and variable limb sizes. The library also has existing support for multi-threaded field operations for performing curve operations in parallel. For example, 256-bit curve elements are decomposed into an array of 64-bit limbs, and each thread can operate on a separate limb. Multi-thread variants of field operation can have a significant data exchange overhead for performing parallel arithmetic since the threads need to communicate. Operating over a smaller field like BN-254 over BLS-381 can reduce this overhead and consumes less memory, but a smaller field

leads to lower parallelization potential compared to larger fields. Balancing these tradeoffs is critical to achieving high performance when generating a proof that involves billions of these basic operations. In our implementation, the parallelization potential is higher than the data exchange overhead. This means the performance hits incurred from the data exchange overhead are smaller than the speedup we experience from parallelizing the arithmetic using multi-threaded field operations. Therefore performing these arithmetic operations in parallel using multiple threads is worth-while.

Multi-threaded field operations are possible with *cooperative groups*, which is a flexible model for synchronization and communication within groups of threads in CUDA. This allows for finer granularity for synchronization between threads and allows threads in a block to communicate. We also take advantage of cooperative groups for global synchronization within a kernel. This reduces the number of kernel calls since we're synchronizing the threads directly within the kernel itself, rather than using a separate kernel launch as a synchronization directive.

In practice, although our MSM implementation does *not* achieve any meaningful speedup over the Barrenteberg's MSM CPU results in its current development stage, there is significant room for improvement at the expense of memory. The current implementation does not use any precomputation in calculating the MSMs, and there's an inherent tradeoff between execution time and memory consumption.

The input points $G_1, ..., G_n$ are known in advance in the proving key.

Therefore the prover can precompute and store many multiples of these base points. With precomputation, the preprocessing phase will generate multiples of the base points, which can be reused over multiple MSMs. More precomputation requires additional space, but accelerates each subsequent MSM.

To handle the increased memory demands, a larger GPU with access to more memory will be necessary. For example, gnark [34], a Golang-based zkSNARK library, normally requires 60 GB to store enough BLS-377 curve points for $n = 10^8$ or 100 million points [35]. This memory consumption is without any precomputation. Moreover, once the problem size becomes too large, relying on disk storage is not feasible since disk reads are significantly slower than memory operations.

Specifically, Yrrid Software [36] implemented a GPU version of Pippenger that precomputes 6 points for every input point. The precomputation requires 38 GB for storing points, and 9.5 GB is used for sorting tables, for a total of about 48 GB of memory for 2^{26} constraints. Another GPU solution from Matter Labs [37] uses fewer points for every input point because it needs more space for the sorting steps, but similarly uses close to 48 GB for 2^{26} constraints. These implementations were executed on Nvidia A40s, which has a maximum VRAM of 48 GB. An A10 GPU with only 24 GB would certainly not work for either implementation. The precomputation needs to be loaded into the GPUs VRAM in order to be used at runtime, leading to huge storage overheads at the cost of improved performance. The

performance of these state-of-the-art implementations is discussed in Section 3.3.2.

3.3 Hardware Architectures

3.3.1 GPUs, FPGAs, and ASICs

Selecting the proper hardware architecture to accelerate zkSNARKs requires carefully examining the trade-offs between different hardware architectures and the prior works that use them. Our analysis suggests that GPUs currently remain the most suitable architecture for accelerating zkSNARKs.

The rapidly evolving zero-knowledge space has left the blockchain community fragmented on the ideal hardware architecture for accelerating zkSNARKs. Modern CPU processors remain sub-optimal for generating proofs because integer computation is performed in the arithmetic logic unit (ALU), which only accounts for 5% of the core's area used in the computation [38]. ALUs are not optimized for finite-field operations where the bit-width is significantly larger than typical 32-bits or 64-bits. Modular arithmetic performed over large prime fields is approximately 10x slower on existing processors [39] as a result. CPUs are further limited by the maximum number of computing cores available on the chip. These factors collectively result in large computation overheads. Still, prior works like DIZK [40] leverage Apache Spark to distribute the proof generation of zkSNARKs across expensive CPU clusters.

Using parallel hardware architectures instead, zkSNARKs can be accelerated by running them over thousands of simultaneous cores using GPUs, crafting low-latency pipelines using FPGAs, or building customized ASICs that drastically enlarge the size of the ALU in the silicon die. GPUs are a natural choice for programming SNARKs using NVIDIA's programming library, CUDA [41]. CUDA virtualizes the hardware with a three-dimensional hierarchy of thread blocks that can be indexed to span the vectors and matrices encoding the computation. GPUs are built using streaming multiprocessors (SMs) that perform the actual computation, and each SM is logically composed of many physical CUDA cores running the same instruction set. In 2023, the newest generation NVIDIA GPUs have reached over 16 thousand CUDA cores in a single chip, a nearly $30x$ increase from 2015 [42].

The current state-of-the-art GPU-accelerted Groth16 zkSNARK systems include Bellperson [43], adopted by cryptocurrency application Filecoin [44], and Mina [45]. cuZK [46], a more recent GPU-based implementation in industry, is an end-to-end Groth16 design [47] that performs all zkSNARK operations entirely on the GPU. cuZK achieves about $19x$ and $2x$ speedups over Mina and Bellperson respectively and supports up to 8 GPUs in parallel. Additionally, open-source GPU libraries, like Icicle [33] developed by Ingonyama and Sppark by Supranational [48], have been developed for accelerating common zkSNARK operations. Recently, works by Yrrid [36] and Matter Labs [37] have been soley focusing on acceleration of MSM on GPUs.

The advantages of GPUs over FPGAs are clear-cut, with GPUs sup-

porting high-performance parallel computing with better performance than FPGAs in terms of transistor density. GPUs support 8nm nodes compared to FPGAs' 16nm nodes respectively, enabling GPUs to outperform FPGAs by about $2x$ for MSM workloads [49]. GPUs offer high bandwidth and larger memory capacities, with on-chip HBM2 memory of up to 80 GB, whereas FPGAs mostly support DDR4 and PCIe 3.0 standards. Moreover, GPUs are more cost-effective than FPGAs, offering higher raw compute per unit cost and are more readily available to consumers on the open-market than FPGAs. The price point for top-of-the-line Xilinx FPGAs can range upwards of 10 thousand dollars, making them $10x$ more expensive than GPU cards with comparable performance like NVIDIA 3090s. Programming GPUs using CUDA is also more user-friendly than individually wiring circuits with a Hardware Description Language (HDL) on FPGAs.

FPGAs benefit from lower power consumption, by a factor of $10x$ on average compared to GPUs, and lower latencies. There have been several implementations focused on improving MSM on FPGAs, such as Hardcaml by Jane Street [50], CycloneMSM by Jump Crypto [51], and PipeMSM by Ingonyama [14]. Since FPGAs focus on latency rather than raw throughput, these implementations include pipelined elliptic curve adder designs that can start a new operation every clock cycle. For example, FPGAs with clock frequency of 250 MHz can start a new point addition every 4 nanoseconds, and receive the result after 100 cycles, or 400 nanoseconds [52]. This design hides the latency of the pipeline as a result.

ASICs alternatively, although more powerful than GPUs and FPGAs, incur large development cycles and the risk of newer rapidly developing proof constructions. It will take time for the community to settle on the most performant proving systems and algorithms, and some ASICs may be outdated at that point. Still, there are teams building ASIC cards with custom power deliveries, higher clock frequencies, and customized interconnect systems. PipeZK [31] is an ASIC implementation with a pipelined architecture focusing on optimizing off-chip memory bandwidth, achieving a 5x speedup over previous CPU and GPU implementations. Ingonyama is a hardware accelerator building ASICs for production-grade systems like ZKSync, Plonky2, Halo2 [53]. Cysic [54] is another FPGA and ASIC hybrid implementation that achieves about $2x - 5x$ speedup over the other FPGA implementations like PipeMSM and CycloneMSM.

The aforementioned trade-offs suggest GPUs remain a more suitable choice for accelerating zkSNARKs than FPGAs and ASICs due to their higher level of programmability and updatabaility. GPUs are highly parallel commodity architectures that can also work with provers in a distributed setting, efficiently splitting the workload across multiple simultaneous GPUs as discussed in prior works [55]. We describe our GPU-based implementation of Plonk [11] in Section 3.4.

3.3.2 GPU Acceleration of Groth16 and Plonk

To understand how the performance and bottlenecks for proving systems have changed over the years, it's necessary to examine prior works focusing on hardware acceleration. We measured the sustained CPU, GPU, system memory and VRAM utilizations for prior works on several bare-metal architectures presented in Tables 3 & 4. The benchmark figures are referenced in the Appendix. Our full benchmarking results can be found at [56] [57] [58], and were submitted formally to the Verifiable Database Systems (VDBS) workshop [59].

The Mina Protocol, a blockchain network using zkSNARKs for verifiable data compression, previously held a competition in 2019 to speed up Libsnark, the industry standard C++ library for Groth16 [60]. Libsnark focuses on performing parallel proof generation on multiple CPU cores. The benchmark GPU prover implementation by Mina [45] executes roughly $2x$ faster than the standard CPU prover. In recent years, significant strides have been made in accelerating zero-knowledge provers. Aleo, a privacy-preserving blockchain environment for executing private applications and smart contracts, held the ZPrize competition [49] in 2022 with over $2M in prizes for speeding up zero-knowledge computations on GPUs and FPGAs.

In our experiments for Mina's Groth16 provers, the limiting factor was available main memory for the GPU and CPU. In the CPU implementation, generating a Groth16 proof over the MNT4-753 curve with 768-bit elements and 2^{25} variables took approximately 1550s and consumed about 200 GB on

Table 3: Hardware Configuration of Testbeds

Testbeds	P100	A10	A40
Device	Tesla P100	Ampere A10	Ampere A40
Core Count	3584	9216	10752
VRAM	16 GB HMB2	24 GB GDDR6	48 GB GDDR6
Host (CPU)	Xeon(R) Gold, 5120 Intel(R) 28-cores 192 GB DDR4 128 GB SSD	Xeon(R) Platinum, 8358 Intel(R) 32-cores 1024 GB DDR4 1 TB SSD	Xeon(R) Platinumm 8358 Intel(R) 32-cores 1024 GB DDR4 1 TB SSD

Table 4: Baseline Implementations

Implementations	Platform	Multi-GPU	Operations	Elliptic Curves
Barretenberg	CPU	X	Plonk	BN-254
Cuda-Barretenberg	GPU	X	Plonk	BN-254
Mina	GPU	X	Groth16	MNT4753
Yrrid	GPU	X	MSM	BLS377
Matterlabs	GPU	X	MSM	BLS377

a 32-core Intel(R) Xeon(R) Platinum 8358 CPU *(Figure 1 in Appendix)* [57]. In the GPU implementation, the execution time is roughly $2x$ faster, but the memory results were more drastic.

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 *(Figure 2 in Appendix)*. Moving from the P100 to the more powerful A10 GPU allowed us to extend these memory figures to larger constraints.

Mina's Groth-16 GPU prover requires a substantial amount of main memory in its preprocessing phase and limited runs to systems of 2^{25} constraints

on the A10 GPU. Generating a Groth16 proof using a GPU took approximately 750s and consumed nearly 900 GB for 2^{25} variables on a NVIDIA A10 (Ampere Architecture) with 24 GB GDDR6 VRAM *(Figure 1 in Appendix)* [57]. The GPU implementation performs a map-reduce to implement the G1 and G2 multi-exponentiation with a batched double-and-add algorithm. The execution time and memory utilization grew log-linearly with respect to the number of constraints in the program.

The GPU memory is much higher since the prover performs a precomputation step that computes multiples of the base points, which accounts for the significantly high memory consumption. The precomputation can be reused over multiple MSMs, reducing the overall computational work. The more precomputation requires additional storage, but accelerates each subsequent MSM.

This results in public parameters $3x$ larger than the program's input *(Figures 3 & 4 in Appendix)*. Preprocessing the parameters produces a file that grows super log-linearly with respect to the size of both the parameters and inputs. These parameters are ultimately injected into the GPUs VRAM and system memory for the prover to access. Since the preprocessed parameter file is loaded into main memory, the parameters are then paged between host and device memory. The results ultimately indicate that prover tradeoffs depend on the computing platform. CPU-based provers use a small 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 proportionally faster. Although the transfers of precomputed multiples are specific to Mina's GPU implementation, newer implementations that outperform these results don't require any precomputation.

Aleo's ZPrize competition for accelerating MSM operations on GPUs and FPGAs is already significantly advancing the hardware acceleration field. The multi-exponentiation algorithm developed by Supranational in their Sppark library [48] represents a complex variation of Pippenger's algorithm. This serves as the baseline benchmark for the competition. The implementation does not require a precomputation step, which massively decreases the parameter file sizes loaded into VRAM and main memory. The benchmarking harness can be configured to run multiple MSM batches of maximum size 2^{26}, all using the same points, but with different scalars. The results indicate MSM execution time is about $250x$ faster than Mina's Groth16 prover for 2^{25} constraints on an A10 GPU. Aleo's codebase also uses about $2.5x$ less memory for 2^{26} constraints on an A10 with 24 GB GDDR6 VRAM. The baseline benchmarks for running a single 2^{26} MSM takes approximately 2.8 seconds and 13 GB VRAM on a NVIDIA A10 *(Figure 5 & 6 in Appendix)*. The more performant A40 GPU computes a proof about about $1.5x$ faster than the A10, with similar memory profiles. The winners of the competition were able to achieve a $2x$ speedup over the baseline.

The current CPU-based Plonk library on which we're working, Barretenberg [12], can benefit from these prior algorithmic optimizations. CPU benchmarks performed on the same testbeds show that for 2^{26} constraints, comput-

ing a single multi-scalar multiplication takes about 6.7 seconds and consumes 46 GB of memory *(Figures 7 & 8 in Appendix)*. The figures exemplify that execution time and memory utilization grow log-linearly with respect to the number of constraints in the program. For 2^{26} constraints, proof generation took about 100 seconds with 255 GB of system memory *(Figures 9 & 10 in Appendix)* [56]. The prover memory consumed for generating the proof is about $6x$ higher than the multi-scalar multiplication here because Plonk precomputes and stores each of the polynomials used in the computation in three forms. The prover memory can be reduced by $6x$ by only storing the polynomials in coefficient form, at the cost of increased computation at runtime.

Consequently, the results from prior works show that the prover bottleneck has slowly been shifting from memory to computation over the years. Still, limited on-chip GPU memory remains a bottleneck for larger circuits. These results provide sufficient motivation for applying these algorithms in Barretenberg for accelerating zkSNARKs and consuming less memory on specialized hardware like GPUs. The details will be discussed in the following section.

3.4 High-Bandwidth Pipeline

The CPU-based Plonk prover in Barretenberg uses a queue to perform the zkSNARK operations in FIFO order. We implement a wrapper class that overrides the queue with a carefully designed, high-bandwidth execution pipeline.

The pipeline replaces the MSM used in the KZG polynomial commitment scheme with our GPU kernel for Pippenger's Bucket Method discussed earlier. The pipeline further swaps out the memory model with traditional CUDA memory allocation techniques to be compatible with the GPU.

The original GPU work used unified memory, an abstraction layer that allocates a single memory space accessible by both the CPU host and GPU device. This simplifies CUDA programming by allowing data transfers to be automatically handled by the driver instead of manually performing memory copies. Unified memory unfortunately performed worse in multi-threaded workloads since it implements a critical section to access memory where threads are serialized, resulting in substantial performance degradation.

Pinned memory was used to load memory onto the GPU instead to achieve higher performance and memory bandwidth over unified memory. While unified memory is pageable by default and can be swept to disk, pinned memory "pins" allocations in the kernel space. This allows for asynchronous, non-blocking data transfers that avoid page faults. To achieve a greater degree of concurrency, pinned memory is necessary for creating multiple streams. A stream is a sequence of CUDA operations that execute in order on the GPU. The operations in a single stream cannot overlap, but operations between multiple separate streams can run concurrently with each other and the CPU. CUDA streams can be strategically used for hiding communication latency by overlapping data transfers with device computation.

Our prover implementation uses multiple CUDA streams to concurrently

execute the multi-scalar multiplication GPU kernels. Recall the KZG polynomial commitment scheme computes a degree d-sized multi-exponentiation (MSM) of G1 elliptic-curve points. In the first round of Plonk, the prover computes the MSM of the wire polynomials $A(x)$, $B(x)$, and $C(x)$ concurrently in separate streams, with a default stream running on the CPU. The prover later computes the quotient polynomial $Q(x)$, which encodes the majority of the information contained in our circuit and assignments all at once, and generates the commitment to $Q(x)$. The prover cannot directly commit to $Q(x)$ since the polynomial's degree is $3n + c$, where c is a constant, which is too large and exceeds the upper-bound of Aztec's trusted setup of $n =$ 100 million constraints. Instead $Q(x)$ is split into three smaller polynomials t_{low}, t_{mid}, and t_{high} each of degree $n + 1$. The prover commits to each of these smaller polynomials in parallel in different CUDA streams. The current implementation performs the MSM calculations in dedicated GPU kernels concurrently with the NTTs running on the host CPU. Since the NTTs are performed on the CPU while the GPU is busy working, it may not be blocking anything.

The prover needs to commit to 9 polynomials during the 5-round protocol. Unfortunately, it's not possible to parallelize the MSM computation across rounds, which would break soundness. Commitments from different rounds cannot be overlapped since the computation of round i depends on the challenges computed in round i, which are dependent on the commitments added to the Transcript until round $i - 1$. After computing each round,

the CUDA streams need to be synchronized and memory transfers need to be coordinated before continuing execution on the host CPU. Synchronizing massive memory transfers in a high-bandwidth pipeline while speeding up the SNARK prover is a challenging task.

SNARK provers' have massive memory requirements for large constraint sizes. Recall the C++ CPU-based implementation of Barretenberg requires 46 GB of system memory for computing 2^{26}-sized MSMs, and the entire prover consumes 255 GB of system memory. The prover stores multiple copies of these large polynomials in different forms which significantly increases the memory consumption during the proof-generation phase. This explains the 6x memory increase for generating a proof, and can be reduced by 6x by storing the polynomials in a single form at the expensive of longer computation time. As the constraint sizes grow past 2^{23} constraints, the prover memory we are pinning in our GPU implementation overtakes the GPUs on-chip VRAM. All tests are performed on NVIDIA $A10$ with 24 GB of VRAM. To mitigate these on-chip memory limitations, the problem needs to be statically partitioned into chunks and distributed among multiple GPU devices. The results from each chunk can then be transferred back onto a single GPU where the final result is aggregated.

Additionally, there are other limitations imposed by the underlying hardware. The PCIE interconnect connecting the CPU and GPU presents another bottleneck for proving systems. As the constraint sizes for a program, grow significantly larger past the upper-bound of the PCIE bandwidth, the PCIE

bus is saturated and bottleneck shifts from memory bandwidth to IO bandwidth. The PCIE 4.0 standard has a limited transfer throughput rate of 64 GB/s, while PCIE 5.0 doubles the memory bandwidth to 128 GB/s. This newer standard is more expensive and not universally adopted by hardware vendors. To mitigate the IO bandwidth limitation, a multi-GPU cluster solution would help here as well. Once the problem is broken up and distributed among multiple GPU devices, NVIDIAs NVLink interconnect system can be used for direct GPU-GPU data. NVLink takes advantage of the raw memory bandwidth of GPUs to facilitate much faster data transfers than PCIE. However, NVLink is about $20x$ more expensive than standard PCIE [61]. As the industry starts to progressively decentralize their prover networks, using custom interconnects would make the cost of generating proofs more expensive.

Our current GPU implementation is at a stage where multiple MSM kernels are running in concurrent CUDA streams, and operations between the CPU and GPU are efficiently accessed in a high-bandwidth pipeline. Although the memory curve have shifted rightwards to support large constraint sizes with the adoption of newer MSM algorithms that require less memory, limited GPU memory is still a bottleneck as the constraint sizes grow exponentially larger.

4 Architecture Directions and Impact on Future Work

4.1 Unified Memory Architectures

Executing zero knowledge proofs on powerful servers with extensive hardware resources merits evaluating the trade-offs between computational efficiency and memory consumption. Acquiring memory is less likely to be a resource limitation compared to processing power. With the decreasing costs of consumer memory chips, it is reasonable for a server to have access to multiple terabytes of DDR4 dynamic random-access memory (DRAM). A larger memory profile can often lead to better performance and computational efficiency through reduced disk IO, faster memory access times, reduced memory swapping, and better caching.

Simultaneously, it is anticipated that cloud data centers and powerful workstation machines will support physical unified memory architectures in the future. Unified memory, similar to Apple's ARM-based M1 silicon chips, enables both the CPU and GPU to have access to the same physical memory address space. CUDA currently supports a unified memory model, but that is a virtual abstraction with a limited memory space. The shift from Non-Uniform Memory Access (NUMA) to unified memory will drastically open the performance space for zkSNARKs, especially for computing zero-knowledge proofs on mobile devices and web browsers.

Consider for instance a recent proof construction, Hyperplonk [62]. Hyperplonk is a sum-check based protocol with multi-linear polynomial commitment scheme. The prover runs in linear-time $O(n)$ instead of $O(n \; log \; n)$, compared to previous SNARKs, by eliminating the super-linear NTTs. Hyperplonk instead implements Multi-Linear Extension (MLE), and the MSM and MLE kernels account for a majority of the proof generation time. The trade-offs compared with Groth16 and Plonk are a larger proof size and verification time, going from $O(1)$ to $O(log \; n)$. The performance bottlenecks for NTT are the random memory access patterns and memory bandwidth. MLE shifts the performance bottleneck from memory access to IO bandwidth which is the PCIE connectivity between GPU device and CPU. If the community widely embraces Hyperplonk, we can expect that GPUs, FPGAs, and ASICs will perform at the same level, as they are limited by the speed of the PCIE standard [61].

Unified memory architectures makes sense for applications bottlenecked by limited GPU VRAM and slower PCIE bus, since the integrated GPU can access the entire memory of the host machine while bypassing explicit memory transfers of memory over PCIE. The memory accessible by the GPU surpasses even the most powerful NVIDIA graphics cards, which have a maximum VRAM of 80 GB [42]. The customized memory bandwidth interconnects are also significantly faster with higher memory bandwidth limits than PCIE. For example, Apple's M1 Ultra chip can be configured with 128GB of unified memory and the infinite fabric interconnect supports memory band-

width up to 800GB/s compared to 64 GB/s for PCIE. The transition of Apple's Mac Pro (the cheese grater) to M1 will include multiple terabytes of unified memory and a more performant interconnect for transferring data.

However, achieving the highest level of performance and scalability for a prover is currently achieved with NVIDIA GPUs using CUDA on powerful servers. The advantage of Metal, Apple's hardware acceleration framework similar to CUDA, are more geared towards generating proofs on mobile devices and web browsers with access to limited computational and memory resources. Since unified memory architectures enable a GPU to directly access the entire system memory, zero-knowledge computations that trade memory for speed can benefit. For example, a larger system memory for the GPU to access would enable MSMs to use more precomputation, resulting in improved performance. As unified memory architectures become more accessible and integrated GPUs become more powerful, these ideas become more feasible in practice.

5 Concluding Remarks

5.1 Summary

Zero-knowledge proofs are powerful tools that provide verification of a certain event or fact, without revealing any information to the verifier. While they are useful in keeping sensitive information private and scaling blockchains, they are computationally expensive to generate. These proofs can be accel-

erated using modern parallel architectures using GPUs by distributing the workload over thousands of cores. As the constraint sizes of these programs become larger, provers' running on GPUs must focus on optimizing the limited on-chip memory and the memory bandwidth between the GPU and host CPU. By distributing the problem among multiple GPUs and overlapping data transfers with device computation, synchronizing the GPU memory becomes vital to achieve a meaningful speedup.

5.2 Future Directions

The CUDA-Barretenberg implementation uses a single GPU to execute the MSM kernels using multiple streams, with FFTs concurrently executing on the CPU. Our GPU prover does not achieve meaningful speedup over Barretenberg's existing prover in its current development stage. Although techniques like precomputation that trade memory for speed would be beneficial here to improve performance. For larger problems that exceed the VRAM of a single GPU, the workload can be further split up and distributed across a multi-GPU cluster to take advantage of massive parallelism and more memory. One complementary technology here is RDMA [63], which can be useful in addressing the significant memory demands of proof generation [55]. RDMA would enable a prover to access the remote host memory of another machine in a cluster without involving its CPU at all, and access the remote memory at close to the same speed of accessing its own host memory. GPU acceleration can also be extended from standard Plonk to

more performant proving systems like TurboPlonk, UltraPlonk, and Hyperplonk. Moreover, unified memory architectures for larger constraint sizes will become increasingly more important for running zero-knowledge workloads with higher memory demands.

6 References

[1] D. Boneh. "Zk whiteboard sessions - module one: What is a snark?" (2022), [Online]. Available: https://www.youtube.com/watch?v=h-94UhJLeck.

[2] Chainlink, "What are zk-rollups (zero-knowledge rollups)?," 2023. [Online]. Available: https://blog.chain.link/zero-knowledge-rollup/.

[3] E. F. B. Vitalik Buterin, "Merkling in ethereum," 2015. [Online]. Available: https://blog.ethereum.org/2015/11/15/merkling-in-ethereum.

[4] J. Bonneau, I. M. andVanishree Rao2, and E. Shapiro2, *Mina: Decentralized cryptocurrency at scale*, Mina White Paper, 2020. [Online]. Available: https://minaprotocol.com/wp-content/uploads/technicalWhitepaper.pdf.

[5] Binance, "Binance proof of reserves," 2022. [Online]. Available: https://www.binance.com/en/proof-of-reserves.

[6] "Aztec's zk-zk rollup: Looking behind the crypto curtain." (2021), [Online]. Available: https://medium.com/aztec-protocol/aztecs-zk-zk-rollup-looking-behind-the-cryptocurtain-2b8af1fca619.

[7] "Demystifying zk programming: Decentralized private computations on aleo." (2023), [Online]. Available: https://www.youtube.com/watch?v=uMmAUssK-PA.

[8] "Zksummit5: Plonk without ffts - justin drake (ef)." (2020), [Online]. Available: https://www.youtube.com/watch?v=ffXgxvlCBvo.

[9] D. Hopwood, S. Bowe, T. Hornby, and N. Wilcox, "Zcash protocol specification," Tech. Rep., 2022. [Online]. Available: https://zips.z.cash/protocol/protocol.pdf.

[10] Polygon, "Polygon zkevm specifications," 2023. [Online]. Available: https://docs.hermez.io/zkEVM/Overview/Overview/.

[11] "Cuda-barretenberg." (2023), [Online]. Available: https://github.com/TalDerei/cuda-barretenberg.

[12] "Barretenberg." (2019), [Online]. Available: https://github.com/AztecProtocol/barretenberg.

[13] "Cuda-fixnum." (2019), [Online]. Available: https://github.com/data61/cuda-fixnum.

[14] C. F. Xavier, *Pipemsm: Hardware acceleration for multi-scalar multiplication*, Cryptology ePrint Archive, Paper 2022/999, 2022. [Online]. Available: https://eprint.iacr.org/2022/999.

[15] J. Groth, *On the size of pairing-based non-interactive arguments*, https://doi.org/10.1007/978-3-662-49896-5_11, 2016.

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

[17] V. Nikolaenko, S. Ragsdale, J. Bonneau, and D. Boneh, *Powers-of-tau to the people: Decentralizing setup ceremonies*, Cryptology ePrint Archive, Paper 2022/1592, https://eprint.iacr.org/2022/1592, 2022. [Online]. Available: https://eprint.iacr.org/2022/1592.

[18] V. Buterin, *Understanding plonk*, VitalkButerin Blog, https://vitalik.ca/general/2019/09/22/plonk.html, 2019.

[19] Jake, *Plonk arithmetization*, HackMD, https://hackmd.io/@jake/plonk-arithmetization, 2022.

[20] "Arkplonk book." (2022), [Online]. Available: https://github.com/ZK-Garage/plonk.

[21] A. Arditi and Y. Zhang, "The anatomy of proof generation," 2023. [Online]. Available: https://scroll.io/blog/proofGeneration#phase-1-filling-in-the-trace-table.

[22] D. Wong, *Understanding plonk*, Cryptologie Blog, https://www.cryptologie.net/article/527/understanding-plonk/, 2021.

[23] A. Kate, G. M. Zaverucha, and I. Goldberg, "Constant-size commitments to polynomials and their applications," *ASIACRYPT 2010 - 16th International Conference on the Theory and Application of Cryptology and Information Security*, p. 177, 2010.

[24] A. Arditi, “Kzg in practice: Polynomial commitment schemes and their usage in scaling ethereum,” 2023. [Online]. Available: https://scroll.io/blog/kzg.

[25] G. Konstantopoulos, “Hardware acceleration for zero knowledge proofs,” 2022. [Online]. Available: https://www.paradigm.xyz/2022/04/zk-hardware.

[26] D. J. Bernstein, *Pippenger's exponentiation algorithm*, Manuscript, University of Illinois at Chicago, https://cr.yp.to/papers/pippenger-20020118-retypeset20220327.pdf, 2002.

[27] A. Bekele, *Cooley-tukey fft algorithms*, Comp 5703 at Carleton University, http://people.scs.carleton.ca/~maheshwa/courses/5703COMP/16Fall/FFT_Report.pdf, 2016.

[28] K. J. Bowers, R. A. Lippert, R. O. Dror, and D. E. Shaw, “Improved twiddle access for fast fourier transforms,” *IEEE Transactions on Signal Processing*, vol. 58, no. 3, pp. 1122–1130, 2010. DOI: 10.1109/TSP.2009.2035984.

[29] K. Aasaraai, K. J. Bowers, E. Cesena, R. Maganti, N. Stalder, and J. A. V. and, “Improving twiddle access for number theoretic transforms: The beginnings,” 2022. [Online]. Available: https://jumpcrypto.com/improving-twiddle-access/.

[30] D. Lubarov, “Plonky2: A deep dive,” 2022. [Online]. Available: https://polygon.technology/blog/plonky2-a-deep-dive.

[31] Y. Zhang, S. Wang, X. Zhang, *et al.*, "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, 2021, pp. 416–428. DOI: 10.1109/ISCA52012.2021.00040.. [Online]. Available: https://doi.org/10.1109/ISCA52012.2021.00040.

[32] LambdaClass, "Multiscalar multiplication: Strategies and challenges," 2023. [Online]. Available: https://www.notamonadtutorial.com/multiscalar-multiplication-strategies-and-challenges/.

[33] "Icicle." (2022), [Online]. Available: https://github.com/ingonyama-zk/icicle.

[34] "Gnark." (2021), [Online]. Available: https://github.com/ConsenSys/gnark.

[35] "Zkstudyclub: Multi-scalar multiplication: State of the art new ideas with gus gutoski (consensys)." (2020), [Online]. Available: https://www.youtube.com/watch?v=Bl5mQA7UL2I.

[36] "Z-prize msm on the gpu submission." (2023), [Online]. Available: https://github.com/yrrid/submission-msm-gpu.

[37] "Z-prize msm on the gpu submission." (2023), [Online]. Available: https://github.com/matter-labs/z-prize-msm-gpu.

[38] "Zksummit6: Accelerating zkp's through specialized gardware - guy granot - chain reaction." (2020), [Online]. Available: https://www.youtube.com/watch?v=nPJTi11vAsg.

[39] J. Thaler, "Measuring snark performance: Frontends, backends, and the future," 2022. [Online]. Available: https://a16zcrypto.com/content/article/measuring-snark-performance-frontends-backends-and-the-future/.

[40] H. Wu, W. Zheng, A. Chiesa, R. A. Popa, and I. Stoica, "Dizk: A distributed zero knowledge proof system," in *IACR Cryptol. ePrint Arch.*, 2018.

[41] NVIDIA, P. Vingelmann, and F. H. Fitzek, *Cuda, release: 10.2.89*, 2020. [Online]. Available: https://developer.nvidia.com/cuda-toolkit.

[42] M. Andersch, G. Palmer, R. Krashinsky, *et al.*, "Nvidia hopper architecture in-depth," 2022. [Online]. Available: https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/.

[43] "Bellperson." (2019), [Online]. Available: https://github.com/filecoin-project/bellperson.

[44] "Filecoin: A decentralized storage network." (2017), [Online]. Available: https://filecoin.io/filecoin.pdf.

[45] MinaProtocol, *gpu-groth16-prover-3x*, version 1, 2019. [Online]. Available: https://github.com/MinaProtocol/gpu-groth16-prover-3x.

[46] T. Lu, C. Wei, R. Yu, *et al.*, *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, 2022. [Online]. Available: https://eprint.iacr.org/2022/1321.

[47] "Cuzk." (2022), [Online]. Available: https://github.com/speakspeak/cuZK.

[48] "Supranational's sppark: Zero-knowledge template library." (2022), [Online]. Available: https://github.com/supranational/sppark.

[49] "Announcing the inaugural zprize competition results." (2022), [Online]. Available: https://www.zprize.io/blog/announcing-zprize-results.

[50] A. Ray, B. Devlin, F. Y. Quah, and R. Yesantharao, "Accelerating zk-snarks - msm and ntt algorithms on fpgas with hardcaml," 2022. [Online]. Available: https://blog.janestreet.com/zero-knowledge-fpgas-hardcaml/.

[51] K. Aasaraai, D. Beaver, E. Cesena, R. Maganti, N. Stalder, and J. Varela, *Fpga acceleration of multi-scalar multiplication: Cyclonemsm*, Cryptology ePrint Archive, Paper 2022/1396, 2022. [Online]. Available: https://eprint.iacr.org/2022/1396.pdf.

[52] K. Aasaraai, D. Beaver, E. Cesena, R. Maganti, N. Stalder, and J. A. Varela, "Accelerating multi-scalar multiplication in hardware and soft-

ware," 2022. [Online]. Available: https://jumpcrypto.com/accelerating-msm/.

[53] "Hardware accelerators for zero knowledge cryptography." (), [Online]. Available: https://www.ingonyama.com/.

[54] "Hardware accelerating zero-knowledge proof." (), [Online]. Available: https://cysic.xyz/.

[55] M. A. Vezenov, "Accelerating zkSNARKs on modern architectures," Master's Thesis, Lehigh University, Tech. Rep., 2022.

[56] T. Derei and B. Aulenbach, "Benchmarking PlonK proving system," Lehigh University, Tech. Rep., 2022, https://github.com/TalDerei/Masters-Research/blob/main/Benchmarking%20PlonK%20Proving%20System.pdf.

[57] T. Derei and B. Aulenbach, "Benchmarking Groth16 proving system," Lehigh University, Tech. Rep., 2022, https://github.com/TalDerei/Masters-Research/blob/main/Benchmarking%20Groth16%20Proving%20System.pdf.

[58] T. Derei and B. Aulenbach, "Benchmarking benchmarking plonk, turboplonk, and ultraplonk proving systems," Lehigh University, Tech. Rep., 2022, https://github.com/TalDerei/Masters-Research/blob/main/Benchmarking%20Plonk%2C%20TurboPlonk%2C%20and%20UltraPlonk%20Proving%20Systems.pdf.

[59] T. Derei, B. Aulenbach, R. I. Shantho, *et al.*, *Scaling zero-knowledge to verifiable databases*, VDBS, https://github.com/TalDerei/Masters-Research/blob/main/VDBS.pdf, 2023. [Online]. Available: https://github.com/TalDerei/Masters-Research/blob/main/VDBS.pdf.

[60] "Libsnark: C++ library for zksnarks." (), [Online]. Available: https://github.com/scipr-lab/libsnark.

[61] "Openzl open discussion, hosted by shumo chu, cofounder at manta network." (), [Online]. Available: https://www.youtube.com/watch?v=qA2wjqv3yUw.

[62] B. Chen, B. Bünz, D. Boneh, and Z. Zhang, *Hyperplonk: Plonk with linear-time prover and high-degree custom gates*, Cryptology ePrint Archive, Paper 2022/1355, https://eprint.iacr.org/2022/1355, 2022. [Online]. Available: https://eprint.iacr.org/2022/1355.

[63] Nvidia, "Nvidia gpudirect rdma," [Online]. Available: https://developer.nvidia.com/gpudirect.

7 Appendix

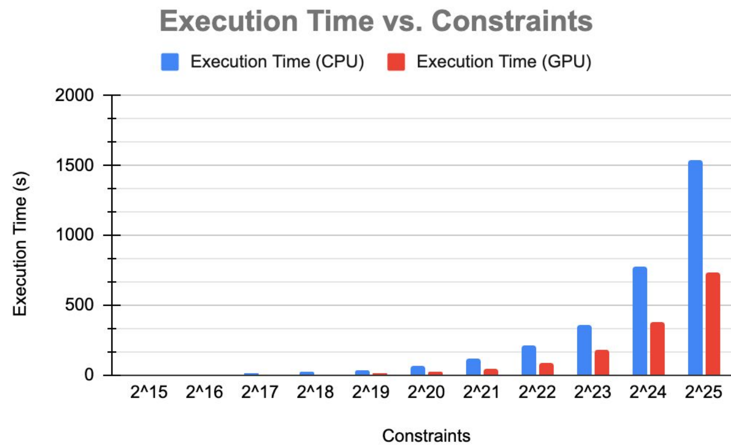

Figure 1: Groth16 CPU and GPU (A10) performance on proof generation as a function of number of constraints.

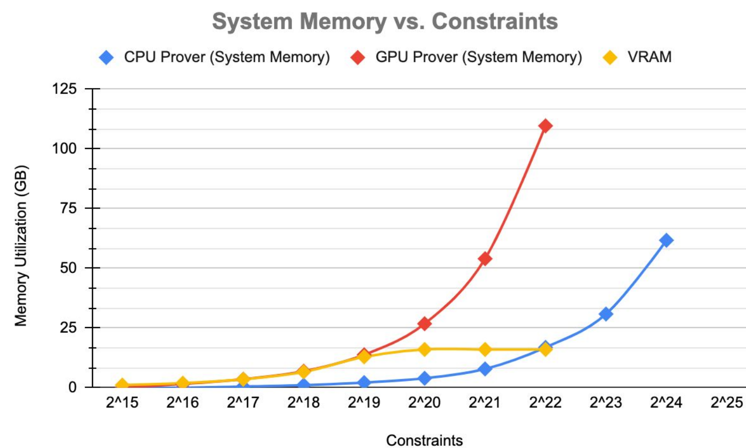

Figure 2: Groth16 system memory consumption for proof generation as a function of number of constraints. Shown for CPU prover, GPU prover (P100), and GPU VRAM.

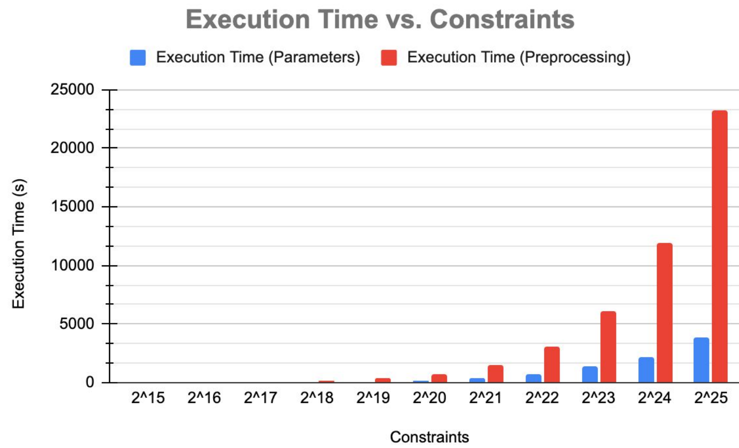

Figure 3: Groth16 performance on parameter and preprocessing generation as a function of number of constraints.

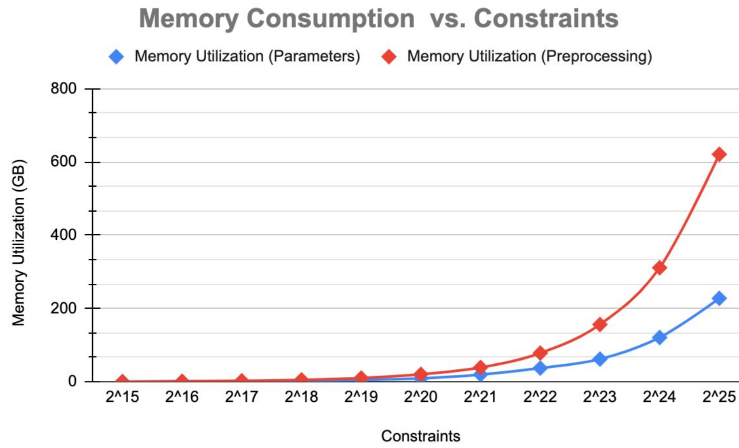

Figure 4: Groth16 memory consumption for parameter and preprocessing generation as a function of number of constraints.

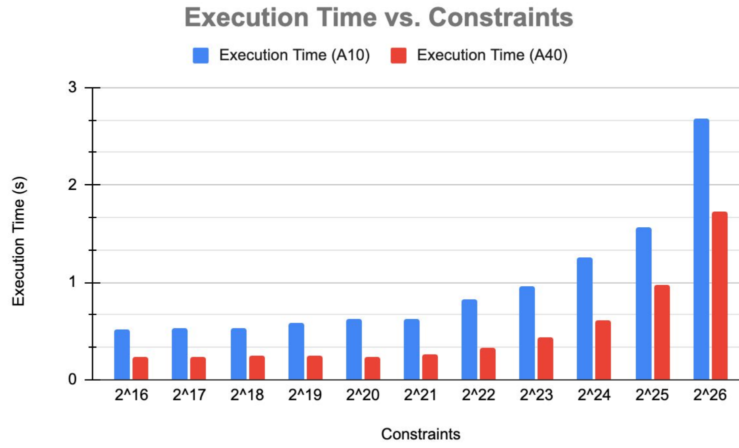

Figure 5: MSM execution time on GPU (A10 and A40) for the ZPrize Baseline Benchmark.

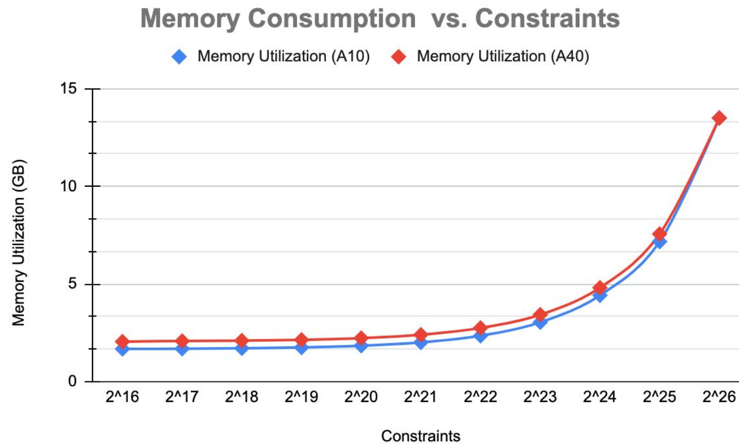

Figure 6: MSM memory consumption on GPU (A10 and A40) for the ZPrize Baseline Benchmark.

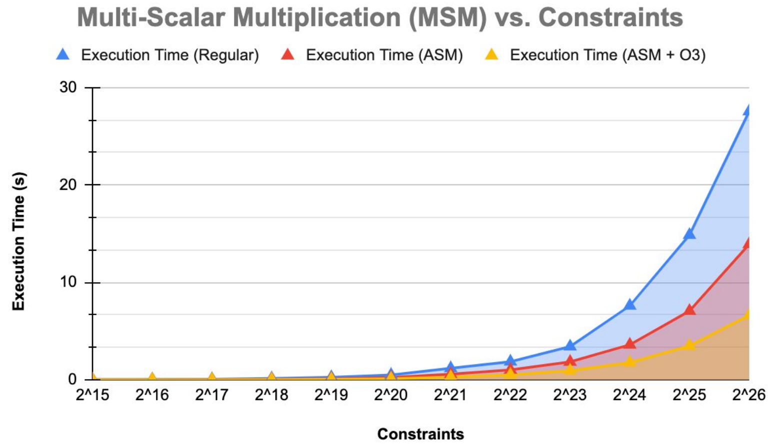

Figure 7: Plonk performance of MSM on CPU as a function of number of constraints.

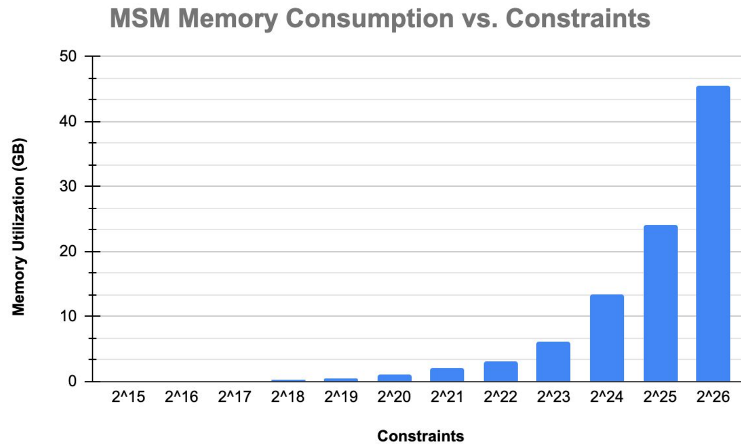

Figure 8: Plonk memory consumption of MSM on CPU as a function of number of constraints.

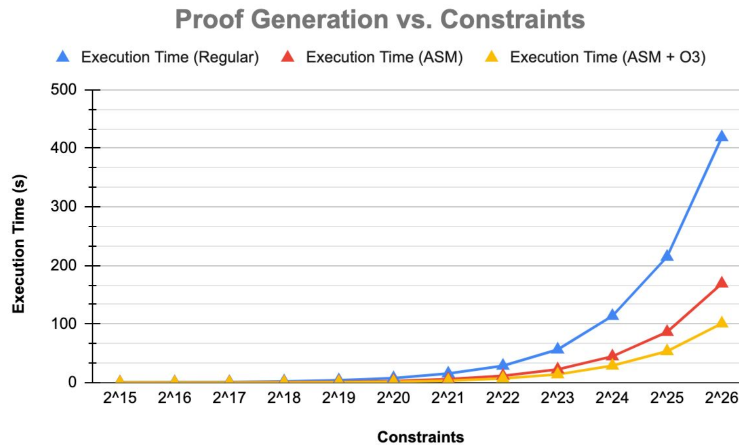

Figure 9: Plonk performance of proof generation on CPU as a function of number of constraints.

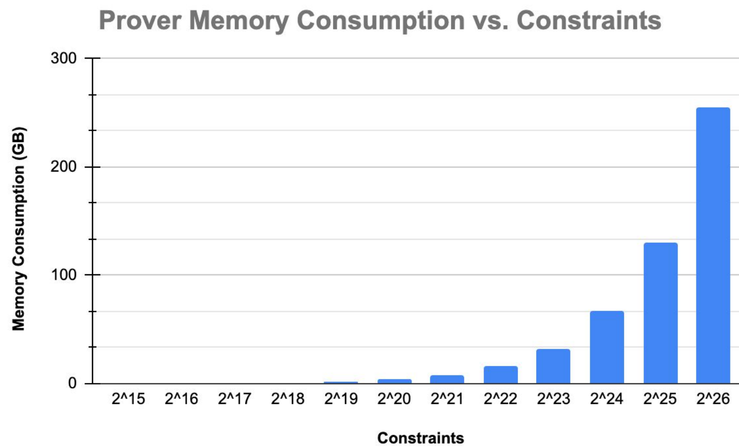

Figure 10: Plonk performance of memory consumption on CPU as a function of number of constraints.

8 Vita

Tal Derei grew up in Las Vegas, Nevada, and graduated from Lehigh University with B.S. in Computer Science and Engineering in May 2022. During his undergraduate years at Lehigh, Tal was a grader and teaching assistant for the Web Systems and Blockchain Algorithms & Systems courses. Tal completed his capstone with Lutron Elecontronics in 2021 where he implemented large-scale distributed systems and Raft consensus protocol for industrial & smart-home lighting systems. In Fall 2022, Tal pursued his M.S. in Computer Science and graduated in Spring 2023. While completing his degree, he was a part of the Scalable Systems and Software group and Lehigh Blockchain studying under Professor Hank Korth. Following graduate school, Tal plans on contributing meaningfully to distributed computing and blockchain technology at a large scale as a software engineer and blockchain developer.