Accelerating zkSNARKs on Modern Architectures

by

Maxim Vezenov

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 2022

This thesis is accepted and approved in partial fulfillment of the requirements for the Master of Science.

Maxim Vezenov

Accelerating zkSNARKs on Modern Architectures

Date

Thesis Advisor

Chairperson of Department

Acknowledgments

I would like to thank all the faculty in the Computer Science and Engineering department at Lehigh Universiy who have helped me over the past four years to become a better engineer and academic.

I would like to specifically thank Hank Korth, who acted as my mentor during completion of my undergraduate degree and again acted as my advisor during my graduate degree. Thank you as well to all members of the Scalable Systems and Software Research group, especially Roberto Palmieri and dePaul Miller, who helped me gain a better understanding of working with low-level systems.

Thank you to my family and friends who have supported me my entire life and throughout my entire Lehigh career. Without everyone I would not be able to achieve what I have so far, and I hope to continue to make you proud.

Finally, thank you to my parents, who early on instilled a sense of curiosity in me. Your support and advice have been irreplaceable in deciding my path.

Contents

Accelerating zkSNARKs on Modern Architectures

Maxim Vezenov

May 6, 2022

Abstract

Zero-knowledge proofs (ZKPs) enable a prover to convince a verifier that a given statement is true without revealing anymore information other than the fact the statement is true. Although this high-level description makes it sound simple, zero-knowledge proofs open up multiple use cases with regards to private transactions, verifiable outsourced computation, and much more. zkSNARKs are a particularly useful ZKP construction as the proof is very small and very fast to verify no matter the problem size. The cost of constructing the proof itself remains a massive bottleneck and current implementations of zkSNARKs are lacking in their ability to meet mainstream demand of this technology. However, the processes necessary to construct a zkSNARK proof, number theoretic transformations (NTT) and multi-scalar multiplication (MSM), are both highly parallel. CUDA programming on NVIDIA GPUs is inherently tuned to the type of programming zkSNARK proof require. In this thesis we explore techniques for how zkSNARKs are created and methods for accelerating zkSNARK creation on the GPU.

1 Introduction

Zero-knowledge proofs (ZKPs) are a cryptography primitive that allows for a party (the prover) to prove to another party (the verifier) that a given statement is true without revealing any more information. This concept can be more formally defined by stating that given a function F, an input x known to both parties, and a secret input w, the prover can show that $F(x, w) = y$ without revealing w to the verifier [1]. With the proliferation of blockchain technology in recent years, ZKPs have been gaining more attention as practical applications have started utilizing ZK protocols. Zcash [2] was one of the first examples of ZKPs being used for privacy preserving cryptocurrency transactions, where all transaction data is encrypted, but the validity of the results can still be confirmed. Aside from privacy-preserving cryptocurrencies, ZKPs can enable multiple use cases such as verifiable outsourced computation [3], anonymous credentials [4], efficient light client verification [5], and many more enterprise applications.

zkRollups such as Loopring [6] have been deployed on Ethereum as a method of scaling the Ethereum blockchain. A zkRollup allows batches of transactions to be executed off-chain and verified on-chain through state transitions of a Merkle tree and a ZK proof. The idea of a zkRollup can be generalized to verifiable outsourced computation, which can allow for a weak client to request heavy computational loads while still being able to verify the validity of the output in a lightweight manner. Largely due to zkRollups, the demand for ZK applications has been increasing significantly in recent years. However, our ability to verify a ZKP is still bound by the overhead of generating a proof.

In order to be able to prove anything about a given program, it has to first be transformed into a constraint system. This constraint system is a set of arithmetic gates that a prover must then operate over to construct a proof. These constraint systems can be quite large as they represent the most basic logical steps of a program. Current ZK protocols operating in production have constraint sizes in the millions. To provide a more specific example, Loopring requires about 64 million constraints to prove 1,024 trades, and that takes about 1 minute and 30 seconds to prove [7]. A Loopring trade is a relatively simple program and the disparity between the time for the underlying computation and the time to prove that computation is already drastic. The complexity and size of the applications to be proven is likely to only increase over time. More complex systems in the future are expected to have constraint sizes that reach the billions and even trillions [8]. Currently, the overhead of a prover makes a ZK system of such complexity infeasible and keeps ZKPs away from large-scale adoption. Luckily, the most computationally heavy processes required to generate a ZKP are highly parallel. These processes performed by the prover are number theoretic transforms (NTTs) and multi-scalar multiplication (MSM), which are elaborated on in Section 3.

1.1 Contribution

GPUs and FPGAs are the two top choices for executing the highly parallel computations required for ZK proof construction. FPGAs are the much more energy efficient option, but GPUs remain popular as they are much more programmable and a common consumer product. GPUs are becomingly an increasingly popular hardware choice for massively parallel clusters such as training machine learning models [9] and reading and writing to databases [10]. GPU programming allows developers to program in C++ with superior tooling to FPGA programming. This tooling on the GPU includes access to other innovative technologies like Remote Direct Memory Access (RDMA). RDMA allows a machine to access the main memory from a remote host without involving the remote machine's CPU. Further details can be found in Section 3.4. This technology is especially useful for parallel computations that require extremely high memory loads. ZKP construction requires a massive amount of memory.

Later, we will show how drastically memory requirements grow with increasingly complex programs and that a large part of the overhead for speeding up a prover on the GPU can be attributed to the size of the problem in memory. This paper will then walk through the changes implemented that enable a ZKP to interface with the GPU and RDMA.

2 Preliminaries

2.1 zkSNARKs Overview

A ZK protocol operates in three main steps, the setup, the prover, and the verifier.

- $Setup(R, secret\ \lambda) \rightarrow$ Common Reference String (pk, vk)

  R is a representation of the program for which proof statements are going to be generated. The details of this representation can be found in Section 2.3. The common reference string (CRS) contains a prover key pk and a verifier key vk. The λ represents secret random variables that are selected in order to construct the CRS. These variables are toxic waste and they must be destroyed to maintain the integrity of the system. If a malicious actor were to gain access to these secret variables, they could then produce false proofs. Techniques such as the powers of tau multi-party computation ceremony [11] help reduce the trust assumption that would otherwise have to be made.

- $Prove(pk, input, witness)$ - π

  The prover accepts the proving key, the input for the relation R, and the *witness* for R that remains secret to the verifier. The *witness* represents a solution that satisfies R when provided with the program *input*. The proof π that is outputted states that $R(input, witness) = true$ without revealing the *witness* to the verifier.

- $Verify(vk, input, \pi)$ - True or False

  The verifier takes in the verifier key, the *input* for the relation R, and the proof π. If the proof holds, the verifier returns true, otherwise it returns false.

zkSNARKs are one of the most popular ZKP constructions. zkSNARK stands for Zero-Knowledge Succinct Non-interactive ARgument of Knowledge. The name is representative of a zkSNARK's key properties.

- Completeness - Given a statement and a witness, the prover can convince the verifier that they know the witness.

- Knowledge Soundness - A malicious prover cannot convince the verifier of a false statement. This property combined with completeness property is the ARgument of Knowledge part of the zkSNARK acronym.

- Zero-Knowledge - The proof does not reveal anything to the verifier but the truth of the statement, in particular it does not reveal the prover's witness.

- Succinctness - The size of the proof is small and fast to verify.

2.2 Groups and Bilinear Maps

All the prover calculations take place within a finite field of prime order p that we will label $\mathbf{F_p}$. $\mathbf{F_p}$ is a field of integers modulus p, represented by the number $0, 1, ..., p-1$. The final major computation required by zkSNARK protocol is the multi-scalar multiplication operation over large bit elliptic curve groups whose elements are in $\mathbf{F_p}$. A group is a set of elements under a binary operation, where performing the operation on two elements in the group will produce another element in the group. We are only looking at cyclic groups of prime order p. Cyclic groups are groups where given a generator g, all elements in the group can be generated by taking a power of g like g^0, $g^1, g^2, ...,$ etc. More formally we can define a cyclic prime order group $G = \{g, g^2, ..., g^{p-1} = 1\}$ [12].

An elliptic curve is a set of points (x, y) which satisfy an equation of the form $y^2 = x^3 + ax^2 + bx + c$ where the roots are distinct [12]. The elliptic curve used in a zkSNARK will be defined over $\mathbf{F_p}$, and the elliptic curve itself will represent a cyclic group prime order group. This means it is possible to use elliptic curves for a bilinear map that acts as a trapdoor. We can define a bilinear map as the pairing e where given groups G_1, G_2, and G_T of prime order p, $G_1 \times G_2 \to G_T$. This means given g_1 and g_2 that generate G_1 and G_2 respectively, for every nonzero $p, q \in \mathbf{F_p}$ we will have $e(g_1^p, g_2^q) = e(g_1, g_2)^{pq}$.

This is how our system hides information. We are hiding p with g_1^p and q with g_2^q yet it can still be verified that $p * q$ using the pairing just mentioned in the previous paragraph [13][14]. This phenomenon where it is possible to operate over encrypted values is also known as homomorphic encryption. Granted we are using a large enough prime order, it is computationally hard to reverse this pairing.

In the setup phase of a zkSNARK random elements from $\mathbf{F_p}$ are selected to be the exponents of two generators for the groups G_1 and G_2 that will then form the prover key and verifier key. These are the secret variables mentioned in Section 2.1. Our ZK system will also have a pairing-friendly elliptic curve that lies in G_T on which pairings will be calculated. Knowing that bilinear maps enable homomorphic encryption, the prover will then use the encodings created during the setup to construct a final proof made up of elliptic curve points that is verified by performing the respective pairings. The final step of the prover is performing multi-scalar multiplications against points of an elliptic curve where the final elliptic curve points will be generated. This is necessary to create the final bilinear encoding that will be verified. This is expanded upon in Section 3.2.

2.3 Rank-1 Constraint Systems

The first step in constructing a ZKP is transforming our program into the correct form. Under the hood, a ZKP will prove the solution to a set of algebraic equations. This is possible by representing the initial program as an arithmetic circuit. An arithmetic circuit breaks down the logic of a program into discrete steps of basic arithmetic operations. The program whose solution a ZKP is proving will often be referred to as a circuit.

Each circuit is then represented as a Rank-1 Constraint System (R1CS). A R1CS is a sequence of groups of 3 vectors A, B, and C, where if given another vector s (the witness), the vector s must satisfy $(A \cdot s) * (B \cdot s) -$

$(C \cdot s) = 0$. The length of the sequence is the number of constraints n. An individual constraint is a single group of (A,B,C) vectors. In order to satisfy the equation, the size of each A, B, and C vector must match the size of s which is the number of m variables in the circuit. The number of variables m is made up of a dummy variable $s_0 = 1$ at the first index, the inputs to the circuit, and all the intermediate variables created from the R1CS representation. Due to these intermediate variables, m is often the number of constraints plus s_0 and a final output variable. The witness vector that satisfies the R1CS will contain all the inputs to the problem as well as all intermediate variables created by the constraint system.

The circuit represented by the R1CS is often very sparse. The n vectors must be satisfied by a witness of size m. Thus, most vector elements will be 0 as a constraint corresponds to a singular gate in the circuit and not every variable in the system. For a matrix of $n \times m$ we will see at most $n + m$ non-zero entities [14].

An R1CS can be represented as a relation. Given $s_0 = 1$ and $s_1 ... s_m \in \mathbf{F_p}$, where $\mathbf{F_p}$ is a finite field of prime order, $A_{i,n}$, $B_{i,n}$, and $C_{i,n}$ corresponds to the ith variable in the nth constraint [13].

$$\sum_{i=1}^{m} A_{i,n} s_i * \sum_{i=1}^{m} B_{i,n} s_i = \sum_{i=1}^{m} C_{i,n} s_i \tag{1}$$

2.4 Quadratic Arithmetic Program

Even with the Rank-1 Constraint System, there is still one more transformation to perform. We can check that every constraint is satisfied at the same time by transforming the R1CS into a Quadratic Arithmetic Program (QAP).

A QAP is the R1CS in polynomial form. Given a system with n constraints and m witness variables, our sequence of A, B, and C vectors will be reformulated into three $n-1$ degree polynomials $A(x)$, $B(x)$, and $C(x)$ [15]. For n constraints, the degree of these polynomials is $n-1$. These polynomials will then be combined into a single $H(x)$ polynomial with the same equality used to solve the R1CS, $A(x) * B(x) - C(x) = H(x)$. Using a QAP, the system can check every constraint at the same time by using the dot product of the polynomials. The $H(x)$ polynomial is then represented as a vector of n coefficients, H_n. H_n is the vector used in the final steps of proof construction that uses multi-scalar multiplication. Each of the polynomials can be deconstructed as below.

$$\begin{aligned}
A(x) &= A_1(x) + A_2(x)...A_m(x) \\
B(x) &= B_1(x) + B_2(x)...B_m(x) \\
C(x) &= C_1(x) + C_2(x)...C_m(x)
\end{aligned} \tag{2}$$

Evaluating each of the m sub-polynomials at a given x where $x \leq$ the number of constraints n, will yield the respective n-th constraint specified by x from the R1CS.

The resulting polynomial of $A(x)$ * $B(x)$ - $C(x) = H(x)$ may not necessarily evaluate to zero at every point, as there are points on the polynomial that do not correspond to a constraint. To work around this, we need to check whether $H(x)$ evaluates to zero at all the arithmetic gates set by the circuit. This can be done by generating another divisor polynomial to check $H(x)$ against. For m equations we pick distinct $r_1 ... r_m \in \mathbf{F_p}$ and define $t(x) = \prod_{i=1}^{m}(x - r_i)$. Put another way, $t(x)$ is the simplest polynomial that is equal to zero at all values that correspond to a constraint. Any polynomial that evaluates to zero at all values of x in $t(x)$ must be a multiple of $t(x)$. This allows us to verify the correctness of $H(x)$ by performing $H(x)/t(x)$ [15]. As we formally defined the R1CS before, we can also similarly define the QAP. However, the vectors previously used have now been replaced by polynomial evaluations and we must add in $t(x)$ for the correctness check of the relation.

$$\sum_{i=1}^{m} A_i(X)s_i * \sum_{i=1}^{m} B_i(X)s_i = \sum_{i=1}^{m} C_i(X)s_i + H(X)t(X) \tag{3}$$

In a practical implementation, the final $n - 1$ degree polynomials will be represented as a vector of coefficients in memory. Upon completion of the set-up phase the prover will receive three vectors A_n, B_n, C_n in order to compute H_n. The methods used to perform these operations are discussed further in Section 3.1.

2.5 Groth16

There are several new zkSNARK constructions that have been specified over the past few years. Over this same time, Groth16 has remained among the state-of-the-art for zkSNARKs. Of the newly proposed zkSNARK constructions, PLONK [16] provided the most practical benefits, such as its universal trusted setup, meaning that a setup is not necessary for every predicate to be proven. In PLONK, after one setup any program can use the system. Since being first announced, it has seen adoption by multiple teams in the ZK industry [17][18]. As PLONK is newer though, Groth16 has seen much more implementation work surrounding it. Groth16 implementations provided a better starting point that could potentially later allow for an RDMA integration. A Groth16 proof consists of 3 group elements, which are only about a few hundred bytes and take only 2 milliseconds to verify. It remains a fundamental ZK system and has seen a good amount of work done in an attempt to accelerate the prover. The Mina Protocol, previously the Coda Protocol, held a challenge to speed up libsnark [19], the industry standard C++ library for creating Groth16 proofs, using GPUs. This preliminary work on the prover provides a useful starting reference for any Groth16 GPU prover. For these reasons and for reasons highlighted in Section 3.4 and Section 3.5, libsnark was chosen as the focus for GPU acceleration.

3 Methods for Accelerating Proofs

Recall from Section 2 that a program must be converted to a R1CS instance, which will then be converted to a QAP. There are two key algorithmic operations used to perform these transformations, number theoretic transforms (NTTs) and multi-scalar multiplication (MSMs).

3.1 Number Theoretic Transforms

The R1CS is transformed into the polynomials $A(x)$, $B(x)$, and $C(x)$ by performing a sum of Lagrange interpolations. A Lagrange interpolation given a set of n points $(x_i, y_i)...(x_n, y_n)$ where x is unique, will produce a polynomial of degree $\leq n-1$ [20]. However, computing a Lagrange interpolation takes $\mathcal{O}(n^2)$ time. Due to this inefficiency novel techniques such as the Number Theoretic Transform (NTT) are employed instead. The Number Theoretic Transform (NTT) is also used for polynomial arithmetic to generate the H_n vector used in the MSM process. The NTT is very similar to a fast Fourier Transform, except due to ZKPs employment of elliptic curves, the algorithm must be executed over finite fields. NTTs replace the normal twiddle factors seen in an FFT with a new finite primitive root of unity w, where w is the nth root of unity if $w^n = 1$ modulo some large prime number [21]. Formally we can define an NTT as given two N-sized arrays a and q where $q[i] = \sum_{j=0}^{N-1} a[j]w_N^{ij}$. Then at every stage i, two elements with a 2^{n-i} stride perform a butterfly operation. The inverse can then be given as $a[j] = N^{-1} \sum_{i=0}^{N-1} q[i] w_N^{-ij}$ [22]. Both the forward NTT and the inverse are necessary for polynomial evaluation as we must use NTTs to convert polynomials to point-value form and perform the convolution, then the inverse NTT will be used to return back to polynomial form.

3.2 Multi-Scalar Multiplication

A multi-scalar multiplication (MSM) can be defined as given group elements $G_1, ..., G_n$ in a group G in a finite field $\mathbf{F}$ and scalars $a_1, ..., a_n$ in between 0 and $|G|$, find $a_1 * G_1 + ... + a_n * G_n$. The order $|G|$ has λ bit length. Each pair $a_1 * G_1$ is a point scalar multiplication and MSM adds up these products to get one final group element. The MSM in zkSNARKs is performed on the polynomial (H_n) and the witness (s_m) against previously chosen elliptic curve points. All the coordinate point values lie in $\mathbf{F_p}$ or an extension field of $\mathbf{F_p}$. This operation is essentially a map-reduce, but using group operations. The map-reduce operation heavily lends itself to parallelism as it can be easily split. Parallelism can be further exploited by splitting the scalars themselves into windows [22]. The most popular algorithm for fast MSMs are versions of the Pippenger algorithm [23], sometimes referred to as the bucket method, which is detailed in the steps below.

1. Divide the λ-bit scalar into λ/b chunks with b bits each, where b is the given window size.

2. The group elements to be multiplied against the scalar are mapped into

a bucket, B_i, based on the value of the chunk that is to be multiplied against. For example, for every chunk of bits translating to the value "5", the respective group element is placed into the "5" bucket. The sum of this bucket is then found. The number of buckets is bound by the maximum possible value of a chunk.

3. Take every bucket and find $G_i = \sum_{j=0}^{2^b-1} j * B_i$

4. Sum up $2^{ixb}G_i$ to arrive at the final point

This process allows substitution of a large number of the point multiplications needed for a MSM with point additions. In my contribution detailed in Section 3.5, I work with the map-reduce method as my focus was on memory requirements rather than algorithmic efficiency. However, the Pippenger method is still important to note as the prime method for accelerating MSM.

3.3 Hardware for Acceleration

The current state-of-the-art prover is an ASIC implementation [22], that for large applications achieves about 5x speedup over previous parallel implementations using the CPU and GPU. Although ASIC and FPGAs are very promising for accelerating ZK proof construction, ZK prover acceleration efforts on the GPU should not stall as a result. Another parallel Groth16 system, DIZK, uses Apache Spark to parallelize a ZK proof across many distributed nodes [14]. PipeZK mentions that their hardware acceleration is a complementary technology as it focuses on accelerating the individual node. Unlike an ASIC or FPGA, the GPU is an architecture organized for parallel computation while also still being a consumer product. A fast and well-distributed GPU prover would allow for many more individuals to participate in a ZK proving network without having to purchase specialized hardware.

GPU programming naturally lends itself to the structures constructed in a zkSNARK. NVIDIA's GPU programming library, CUDA, has a thread hierarchy where threads can be indexed inside thread blocks up to three dimensions [24]. This feature allows for programmers to write code that inherently spans the domains of the vectors and matrices it is operating over. If we are to expect future innovations in zkSNARK design, those zkSNARK protocols will be more easily programmed and update-able on GPUs than ASICs or FPGAs, especially if the prover is within a massively distributed system as mentioned above.

GPUs have continued to see improvements in the amount of parallelism possible as more cores have continued to be added with each next-generation chip. In 2022, the best NVIDIA GPUs are beginning to reach over 5 thousand CUDA cores. Just back in 2015, chips were being released with 512 CUDA cores [25]. GPUs are very capable at executing parallel processes, and seemingly we should be able to speed up as much as needed with more GPUs. Parallel execution on the GPU chip itself is not the true bottleneck when it comes to large programs such as Groth16, but rather the time it takes to synchronize the massive memory transfers required by a ZK prover to the GPU and back to the host.

Remember that in a zkSNARK, for a constraint size of n with m variables, to create a bilinear map, the MSM must be performed against a vector of elements from groups G_1 and G_2. In the preprocessing step, each constraint matrix is encoded into a vector of m elements in either G_1 or G_2. A Groth16 program with a λ-size of 768 bits and 2^{20} constraints will require over 24 GB for just its multi-scalar multiplication operations. For 2^{25} constraints, the G_1 elements are about 200 GB and the G_2 elements are about 400 GB. For some perspective on how these numbers translate to real programs, a 10MB SHA2 compression is represented by 2^{30} constraints [8] and as mentioned in Section 1, Loopring's largest trade circuit is 2^{25} constraints [7].

No matter the hardware architecture, a ZK prover must account for these memory requirements in its design. This means a careful memory pipeline that can bring values into main memory as computations on past values are running. Knowing that the top-of-the-line NVIDIA GPUs only support 80GB of on-chip memory [26], it becomes even more obvious that synchronizing memory transfers in a high-bandwidth pipeline is critical to speeding up the prover.

3.4 GPUDirect

Past ZK provers have taken steps to mitigate against the massive memory requirements of running a ZK prover. Loopring performed compression techniques on the prover data and precomputed multiple inputs to cut down on

MSMs [7], PipeZK focused on optimizing off-chip memory bandwidth with a highly optimized algorithm, while DIZK split computation over many distributed nodes. Multiple zkRollup industry leaders have plans for decentralizing the proving process in their systems. Creating an efficient GPU prover will require a mechanism that allows for fast memory bandwidth on a singular node and also over multiple nodes. GPUs have an advantage in solving this issue due to modern tooling that ASICs and FPGAs do not possess, or for which sufficient tooling is missing, such as Remote Direct Memory Access (RDMA). RDMA allows for accessing the main memory of a remote machine without involving the remote host's CPU. This grants access to much more storage without having to read from disk, thus resulting in greater speed when accessing memory that must be brought onto the GPU.

GPUDirect is a tool that uses RDMA to create a path for data exchange between a GPU and a remote host using PCI Express [27]. This is extremely powerful as a GPU program can access remote memory at close to the same speed as accessing its own host memory.

3.5 Stream Ordered Pipeline

The current best implementation of a libsnark GPU prover [19] uses CUDA Unified Memory [28] for loading memory onto the GPU. Unified Memory is a useful programming tool that provides a universal address space seen by both the CPU and GPU. This greatly simplifies GPU programming as explicit memory transfers between the host and device do not need to be specified. However, CUDA Unified Memory is not directly supported for integration with GPUDirect RDMA [27]. If we are to experience the potential benefits of a multi-GPU RDMA cluster moving away from Unified Memory is necessary based on the current state of the tooling.

This introduces its own challenges, because, as mentioned previously, zkSNARK provers have massive memory requirements. All tests have been run on a NVIDIA P100 GPU with 12GB of memory. Once the problem size starts to reach 2^{19} or 2^{20} constraints, the memory needed is greater than what is available on chip. To mitigate this restriction, we added a static partition. After loading the ZKP parameters and inputs from file the memory available on the GPU is fetched and the partition for chunking the data is determined by finding `(total size of data to be put on chip) / (free device memory) + 1`.

Splitting up our computation far from solves our problem. Now we must coordinate memory transfers and accesses for our newly split up problem. It is not a simple task to synchronize such large memory transfers and still maintain a high level of speedup. Even memory transfers between disk and the host's main memory must be considered, and not just transfers between the host's main memory and the GPU. Our prover uses CUDA streams to enable concurrency between our multi-scalar multiplication kernels. To make these streams concurrent with the host, we must "pin" the memory for the GPU to access it. This pinned memory is moved from disk to main memory. As the constraint sizes grow it becomes obvious that a ZK program must not only sync from the host memory to the GPU, but rather must sync from disk to host memory to GPU memory. In a system with 2^{20} constraint using 3 chunks, freeing the pinned G_1 multiples on the host takes about 30 seconds each time. This almost completely eliminates any speedup, resulting in about the same execution time as the parallel CPU version. Minimizing the amount of these allocations and frees is critical for achieving speedup, especially as more chunks are needed for larger constraint sizes. CUDA streams help us by providing a way to asynchronously perform tasks that would otherwise block execution.

A CUDA stream is simply a queue for work on the GPU, where operations within a stream cannot overlap but operations between separate streams can overlap. The outputs of the MSMs do not depend on one another until they need to be combined with the H coefficients. Our prover has a stream for each MSM operation being performed and then a default stream that runs on the CPU. Creating a stream-ordered system is necessary as it allows for our MSM calculations to run concurrently to each other on the GPU itself while also running concurrently to the CPU. The CPU can then run the NTTs at the same time that the GPU is working on the MSMs. Ideally, the NTT would also be brought onto the GPU; however, the MSM has not seen enough speed up so that the NTT is bottle-necking the process. Our prover for a constraint size of 2^{20} currently speeds up a parallel CPU implementation by about 1.5x from 180 seconds to 120 seconds. The MSM calculations remain as the majority of the computation.

The MSM operation for the libsnark prover is now in a state where it can be better split across multiple GPUs within a distributed system utilizing RDMA. The previous state of the prover utilizing Unified Memory had the ability to be distributed across multiple nodes, however, if we are to utilize RDMA, the prover needs to use the traditional CUDA allocation strategies.

4 Conclusion

Zero knowledge proofs are one of the biggest breakthroughs in cryptography, with a multitude of use cases. The potential for this technology to be adopted in the mainstream is currently impeded by the ability to construct ZK proofs in a fast and cost effective manner. However, the primary obstacle is not a lack of computational power. GPUs are an extremely parallel architecture that has been successfully applied to similar problems such as machine learning. In this paper we present a libsnark GPU prover that can split up its MSM computation in a statically partitioned pipeline. We moved away from using Unified Memory in order to enable distributing the computation across a RDMA cluster. Further optimizations that are mentioned in Section 4.1 can be applied to account for the memory requirements needed by the ZK proving process. Any prover running on a GPU must focus on memory bandwidth between the GPU, the CPU, and disk memory. We can always split up our GPU problem onto more machines, however, precisely synchronizing the GPU will remain necessary for any ZK construction looking to scale.

4.1 Future directions

The libsnark GPU prover has a lot of room to be heavily tuned with regards to memory transfers, numerical algorithms, and compression.

For exceptionally large problems, the data we are pinning for the stream ordered pipeline overtakes what is available in main memory. I created an implementation that allowed for problems of any memory size, however, the speedup is reduced by the costs associated with large allocations and frees. Synchronizing accesses between the disk, CPU, and GPU will be necessary in order to meaningfully speed up the largest ZK circuits. It is always possible to further parallelize a problem with more hardware, however, all the data must still be brought onto every GPU and returned to a singular source for the final result. RDMA would probably be most helpful here as any ZK proof that will exceed the host memory of a given machine could have other remote machines access that memory as if it was local memory. I foresee work here being the most promising as a distributed RDMA prover could split work in the fastest manner possible for a GPU cluster.

Aside from optimizing memory accesses, changes in the numerical algorithms used would be beneficial, such as Pippenger for MSM. The current prover uses a map-reduce operation with precomputed group elements. Precomputation essentially replaces computation with extra memory that must be brought onto the GPU. It is not entirely clear whether precomputation actually negates the cost of transferring data back and forth. Benchmarking work should be done to test whether precomputing these group elements is actually beneficial to speed up. Although the current prover's MSM operation remains the main system bottleneck, GPU algorithms that accelerate NTTs still would be beneficial once MSM calculations are improved.

As mentioned in Section 2.3, the constraint system and thus polynomial evaluation formed consists of sparse vectors. Further optimizations could be made to better align any parallelism with the sparsity of the constraint system. It is important to note that sparsity is different in every system, and for zkSNARKs while some rows in the evaluated QAP are sparse, some are dense [14]. This is because one variable could be used in multiple constraints. Any algorithm taking advantage of sparsity should also have a system for identifying sparse vectors. This identification can either be precomputed or on-the-fly and should be benchmarked just like the group elements precomputation. Compression techniques could also take advantage of this sparsity to help reduce the amount of memory that must be read and copied in the first place. Due to the memory required for compiling a zkSNARK circuit, compression would be useful even when not considering sparsity. It will be especially important to consider compression when the system becomes distributed. Although RDMA can bring fast remote memory access, reducing the amount of memory that needs to be sent across a network will always be beneficial, especially for any fallback network that does not use RDMA.

5 References

[1] M. Blum, P. Feldman, and S. Micali, “Non-interactive zero-knowledge and its applications,” in *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, ser. STOC ’88, Chicago, Illinois, USA: Association for Computing Machinery, 1988, pp. 103–112, ISBN: 0897912640. DOI: 10.1145/62212.62222. [Online]. Available: https://doi.org/10.1145/62212.62222.

[2] E. Ben Sasson, A. Chiesa, C. Garman, *et al.*, “Zerocash: Decentralized anonymous payments from bitcoin,” in *2014 IEEE Symposium on Security and Privacy*, 2014, pp. 459–474. DOI: 10.1109/SP.2014.36.

[3] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou, “Vsql: Verifying arbitrary sql queries over dynamic outsourced databases,” in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 863–880. DOI: 10.1109/SP.2017.43.

[4] F. Baldimtsi and A. Lysyanskaya, “Anonymous credentials light,” in *Proceedings of the 2013 ACM SIGSAC Conference on Computer amp; Communications Security*, ser. CCS ’13, Berlin, Germany: Association for Computing Machinery, 2013, pp. 1087–1098, ISBN: 9781450324779. DOI: 10.1145/2508859.2516687. [Online]. Available: https://doi.org/10.1145/2508859.2516687.

[5] https://minaprotocol.com/, *Mina: Economics and monetary policy*, Mina Protocol, 2020. [Online]. Available: `https://minaprotocol.com/wp-content/uploads/economicsWhitepaper.pdf`.

[6] loopring.org, *Loopring protocol 3.0 design doc*, Loopring, 2019. [Online]. Available: `https://blogs.loopring.org/loopring-protocol-3-0-design-doc/`.

[7] B. Devos, *Loopring's zksnark prover optimizations*, medium.com, Ed., [Online; posted 1-March-2020], Mar. 2020. [Online]. Available: `https://medium.loopring.io/zksnark-prover-optimizations-3e9a3e5578c0`.

[8] J. Drake, *Plonk without FFTs*, Youtube, 2020. [Online]. Available: `https://www.youtube.com/watch?v=ffXgxvlCBvo`.

[9] R. Raina, A. Madhavan, and A. Y. Ng, "Large-scale deep unsupervised learning using graphics processors," in *Proceedings of the 26th Annual International Conference on Machine Learning*, ser. ICML '09, Montreal, Quebec, Canada: Association for Computing Machinery, 2009, pp. 873–880, ISBN: 9781605585161. DOI: `10.1145/1553374.1553486`. [Online]. Available: `https://doi.org/10.1145/1553374.1553486`.

[10] d. Miller, J. Nelson, A. Hassan, and R. Palmieri, "Kvcg: A heterogeneous key-value store for skewed workloads," in *Proceedings of the 14th ACM International Conference on Systems and Storage*, ser. SYSTOR '21, Haifa, Israel: Association for Computing Machinery, 2021, ISBN:

9781450383981. DOI: 10.1145/3456727.3463779. [Online]. Available: https://doi.org/10.1145/3456727.3463779.

[11] S. Bowe, A. Gabizon, and I. Miers, "Scalable multi-party computation for zk-snark parameters in the random beacon model," *IACR Cryptol. ePrint Arch.*, vol. 2017, p. 1050, 2017.

[12] T. R. Shemanske, *Modern cryptography and elliptic curves: A beginner's guide*. American mathematical society, 2017.

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

[14] 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.

[15] R. Gennaro, C. Gentry, B. Parno, and M. Raykova, "Quadratic span programs and succinct nizks without pcps," in *IACR Cryptol. ePrint Arch.*, 2012.

[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.*, vol. 2019, p. 953, 2019.

[17] Zcash, *Plonkish arithmetization*, 2022. [Online]. Available: https://zcash.github.io/halo2/concepts/arithmetization.html.

[18] zkSync, *Cryptography used*, 2022. [Online]. Available: https://docs.zksync.io/userdocs/security.html#security-overview.

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

[20] E. Waring, "Vii. problems concerning interpolations," *Philosophical Transactions of the Royal Society of London*, pp. 59–67,

[21] S. Kim, W. Jung, J. Park, and J. H. Ahn, "Accelerating number theoretic transformations for bootstrappable homomorphic encryption on GPUs," in *2020 IEEE International Symposium on Workload Characterization (IISWC)*, IEEE, Oct. 2020. DOI: `10.1109/iiswc50251.2020.00033`. [Online]. Available: `https://doi.org/10.1109%2Fiiswc50251.2020.00033`.

[22] 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`.

[23] D. J. Bernstein, "Pippenger's exponentiation algorithm," 2002.

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

[25] NVIDIA, *Datasheet quadro k1200*, 2016. [Online]. Available: `https://www.nvidia.com/content/dam/en-zz/Solutions/design-`

visualization/documents/75509_DS_NV_Quadro_K1200_US_NV_HR.pdf.

[26] ——, *Nvidia a100 tensor core gpu*, 2021. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/a100-80gb-datasheet-update-nvidia-us-1521051-r2-web.pdf.

[27] ——, *Gpudirect rdma*, 2022. [Online]. Available: https://docs.nvidia.com/cuda/gpudirect-rdma/index.html.

[28] ——, *Unified memory programming*, 2022. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-unified-memory-programming-hd.

6 Vita

Maxim Vezenov grew up in Center Valley, Pennsylvania with his two parents Dmitri Vezenov and Marina Busuek. Maxim attended Lehigh University where after three years he graduated with a B.S. in Computer Science and Business in May 2021. During his time at Lehigh, Maxim was a grader and teaching assistant for courses such as Software Engineering, Computer Architecture, and Data Structures. He began his software engineering journey as an intern working for the decentralized cryptocurrency exchange Dolomite. Following this internship, the next summer Maxim was an intern at Klaviyo where he worked as a full-stack engineer. Upon completing his undergraduate degree Maxim interned at another blockchain firm, ChainSafe Systems, where he helped build out ChainBridge, a cross-blockchain communication protocol. After this internship Maxim pursued his M.S. in Computer Science in Fall 2021 which he graduated from a year later in Spring 2022. While completing his degree, Maxim was a part of the Scalable Systems and Software group studying under Professor Hank Korth. Following graduate school, Maxim plans to work as an engineer in the blockchain and distributed systems field.