ETH ZÜRICH

MASTER THESIS

GPU Accelerated zk-SNARKs

Author:
Uroš Tešić

Supervisor:
Prof. Dr. Srđan Čapkun

Advisors:
Karl Wüst
Moritz Schnedier

System Security Group
D-INFK

June 13, 2019

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

GPU Accelerated zk-SNARKs

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Tešić	Uroš

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zürich, 28. 5. 2019.	Uroš Tešić

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.

ETH ZÜRICH

Abstract

D-INFK

MSc in Computer Science

GPU Accelerated zk-SNARKs

by Uroš Tešić

Cryptocurrencies promise a new age of money and payment systems. By distributing trust they prevent any party from controlling the flow of resources. However, this comes at a price – most cryptocurrencies rely on a public ledger. This puts user's privacy at risk because anyone can monitor and track transactions.

The cryptocurrency Zcash provides completely private transactions by using zero-knowledge proofs (zk-SNARKs) to validate them. Unfortunately, zk-SNARKs are expensive to compute limiting their widespread adoption on computationally limited devices such as mobile phones. They are also complex to implement, preventing the development of hardware wallets for Zcash.

In this thesis, we take a look at porting computationally expensive part of zk-SNARKs to a GPU to take full advantage of the available processing power. We also explain the difficulties involved in developing OpenCL code meant to be executed on GPUs from multiple vendors. Finally, we identify the hardware constrains that need to be satisfied to achieve a significant speedup.

Keywords: zk-SNARKs, Zcash, OpenCL, GPU, multiexponentiation, elliptic curves, scalar multiplication

Acknowledgements

I'd like to thank my advisors Karl Wüst and Moritz Schneider for meeting with me every week, discussing the results and helping me shape this thesis. Their patience has been invaluable. I'd also like to thank my supervisor Professor Srđan Čapkun for teaching me computer security, even though I'll be wearing a tin-foil hat for the rest of my life.

Finally, I am grateful to my family and friends for supporting me during the studies, and tolerating me while I was studying for my exams.

This thesis wouldn't be possible without MSOP scholarship from ETHZ, which enabled me to live and study in Zürich.

Contents

List of Figures

Chapter 1

Introduction

One of the biggest events in computer science in the last decade was the invention of Bitcoin [Nak+08]. On the surface, Bitcoin offers perfect anonymity. Users can generate an arbitrary number of new addresses. Tumblers offer increased privacy by transfering Bitcoin through thousands of different accounts and send laundered funds to the user (for a small fee). However, data in the blockchain is public. Transaction history can be combined with out-of-blockchain data to de-anonymize users of Bitcoin [BKP14]. Further graph analysis can be used to defeat tumblers as well [BHC17].

Zcash [Hop+19] is a fork of Bitcoin that tries to address this issue. It contains two types of addresses: transparent (**t-addr**) and shielded (**z-addr**). Transparent addresses behave like Bitcoin addresses – all transaction history (identities and amounts) are public. Shielded addresses encrypt this data to prevent leaks – the transactions reveal nothing about its users, or the amounts transferred. For transparent addresses, miners can easily check if the transaction is valid (eg. the account has enough money) by iterating through the previous transactions in the blockchain. For shielded addresses this isn't possible, so the party creating the transaction needs to provide one more piece of information – a zero-knowledge proof that the transaction is valid.

It isn't enough for a proving system to be zero-knowledge to be used in practice. Its proofs need to be small because it will be stored in the blockchain. Furthermore, miners need to verify every transaction before they add it to the block, so it must be non-interactive and fast to verify. Zcash uses zk-SNARKS for this purpose, but these properties come at a cost – proof generation is extremely slow. Many users have cryptocurrency wallets on their phones, which are slower than desktop CPUs. Therefore, wallets choose not to implement shielded transactions at all.

In this thesis we take a look at using graphics cards, present on many devices today, to accelerate zk-SNARKs. In order to make our solution cover as many platforms as possible (including mobile phones), we port performance critical code (scalar multi-exponentiation over curve BLS12-381) to OpenCL [SGS10]. We benchmark different algorithms for multi-exponentiation on different devices (Intel, NVIDIA, AMD and ARM). We explain slower performance on GPUs by the requirements of the most efficient algorithm, and limitations of modern GPUs.

The main contribution of this thesis is the evaluation of performance of scalar multi-exponentiation of elliptic curve points on modern GPUs. We also discuss problems that arise while writing cross-platform code meant to run on GPUs from different

vendors. We benchmark and compare Zcash performance on both desktop and mobile CPUs. Finally, we discuss the limitations of modern hardware and compilers we encountered and advise how they could be fixed when desiging a hardware wallet for an FPGA.

The remainder of the thesis is organized as follows. The background and related research are presented in Chapter 2. In Chapter 3 we explain the anatomy of Zcash and zk-SNARKs. Chapter 4 covers the architecture of OpenCL. Implementation details of different algorithms are covered in Chapter 5. Chapter 6 describes the tests we performed and the results of our measurements, as well as the comparison of OpenCL support by different vendors. Chapter 7 discusses the results from Chapter 6. Finally, we propose possibilities for further research in Chapter 8 and provide a summary of our most important results in Chapter 9.

Chapter 2

Related Work

Wu et al [Wu+18] implemented a distributed version of zk-SNARKs. Their implementation increased the supported circuit size from 10-20 million to 1 billion. This was done by distributing work on computing clusters. Their result shows that zk-SNARK computation is quite parallelizable. However, Wu et al focused on distributing proof calculation on multiple CPUs and clusters, not GPUs.

Elliptic curve operations have been frequently benchmarked on GPUs [MC; Ber+10; ABS10]. However, all of the papers focused on throughput of scalar multiplication (exponentiation) used in cryptographic signatures. The consensus is that GPUs can lead to a significant speedup when calculating many signatures. While exponentiation algorithms can be used to implement multiexponentiation, multiexponentiation is a problem of great importance in cryptography, and should be treated separately.

Chang and Lou [CL97] and Borges et al [BLP17] took a look at multiexponentiation in a parallel setting. Chang and Lou showed that multiexponentiation can be efficiently distributed to multiple nodes. Borges et al benchmarked parallel multiexponentiation on multi-core processors in a modular group and reported a significant speedup. However, none of these papers discuss accelerating elliptic curve multiexponentiation on a GPU.

Many papers compare OpenCL performance to CUDA, or accross two different platforms – usually NVIDIA and AMD [FVS11; KDH10]. They usually focus on kernel performance and tuning the kernels to run efficiently on different platforms [Kom+10; Hen+14]. This is in line with Sorensen's paper [SD16] where they show that that developers rarely test their code on multiple platforms, and that platforms often suffer from compatibility issues and bugs. In this thesis we focus on issues programers need to overcome to develop a kernel for devices from four different vendors (NVIDIA, AMD, Intel, ARM).

Chapter 3

Zcash and zk-SNARKs

In this chapter we provide an overview of Zcash transactions and how they provide anonymity to its users. Next, we explain the origin of zero-knowledge proofs in the paper by Goldwasser et al [GMR85]. Afterward, we go through the steps of a zero-knowledge proof used in Zcash – zk-SNARKs. Finally, we discuss the changes to Zcash in the latest update (Sapling).

3.1 Zcash

Cryptocurrencies have appeared recently and have been marketed as an alternative to cash and electronic payments. The best example is the Bitcoin bubble which made people gain millions overnight, and lose them soon after. However, Bitcoin is of limited use today. It cannot be used as a general currency due to the low throughput and long waiting times for the transaction to be added to the block. Its status as an anonymous currency is also disputed, resulting in 2 different cryptocurrencies being developed to fill that void - Monero (XMR) [Mon] and Zcash (ZEC) [Zcac].

Zerocash whitepaper [Sas+14] identified the faults in Bitcoin and proposed an alternative – Zerocash. Zcash is an instantation of Zerocash built on top of Bitcoin protocol. Zcash has two different addresses. Transparent addresses behave the same way as Bitcoin addresses. Data resides in the public blockchain so it can be tracked the same way as with Bitcoin. However, shielded addresses (Zerocash addresses) reveal nothing when they are a part of the transaction. This is accomplished by using notes (**NOTE**) which contain the public key (**PK**) of the owner, some amount of Zcash (**M**), as well as a unique identifier (**N**). Every shielded transaction results in a note like this (transaction output). When this note is used in a transaction, it is spent, and its nullifier (**NULL(N)**) is published. Valid, unspent notes are ones which are in the set of all generated notes, and whose nullifier hasn't been published.

Keeping all notes in a list would result is an abysmal performance, so they are kept in a Merkle tree. Furthermore, instead of keeping notes in a public structure, where everyone can see them, we keep commitments to notes (**COMM(NOTE)**). This guarantees that the value (**M**) and the owner's identity (**PK**) are not public. However, now that we don't know **PK** of the note's owner, how do we verify the transaction?

Spending a note in Zcash involves computing a zero-knowledge proof π. This requires the **NOTE**'s spender to prove the following:

- The commitment **COMM(NOTE)** exists in a Merkle tree with all notes
- That they have the private key **SK**, corresponding to the note's public key **PK**

- The **NULL(N)** is equal to the nullifier provided (**T**)

If the proof verification passes, the node can check if the note has been spent previously by searching for **T** in a Merkle tree with published nullifiers. This results in the transaction being accepted and added to the blockchain. Along with this **T** is added to the set of spent nullifiers. For more information read [Zcah; Hop+19; Sas+14].

3.2 Zero-Knowledge Proofs

In 1985. Goldwasser, Micali, and Rackoff published a paper with an interesting twist [GMR85]. Most papers discussed methods of combating a dishonest prover, trying to deceive an honest verifier. Their work took a completely different approach: What information could a verifier obtain from a prover? They defined a new class of languages - IP (Interactive Proof). IP contains language L for which there is an interactive protocol between the prover P and a verifier V, after which P has convinced V with a non-negligible probability that the statement s is in the language L.

The interactive proof must satisfy the following properties:

- **Completeness** If the statement s is in L, there exists a prover P that can convince the poly-time verifier V of this, with probability of at least $2/3$.

$$s \in L \implies \exists P Pr[out_V\langle V, P\rangle(s) = 1] \geq 2/3$$

- **Soundness** If the statement s isn't in L, no prover P can convince the poly-time verifier V that it is with probability $1/3$ or greater.

$$s \notin L \implies \forall P Pr[out_V\langle V, P\rangle(s) = 1] < 1/3$$

The choice of constants $2/3$ and $1/3$ is arbitrary because probability amplification can be used until we are satisfied with the probability of error.

Another important property that proof may have is **zero-knowledge**. While two previous properties are tied to different provers, zero-knowledge states that all verifiers can learn only that the statement s belongs to the language L. This statement can be formalized using a simulator and a transcript. V can generate the transcript with the same distribution as the messages exchanged during the protocol, without ever communicating with P. Hence, V learns nothing from P, other than $s \in L$.

Unfortunately, zero-knowledge proofs described are interactive. They cannot be used in blockchain because every verification of the blocks would require all parties to be online and exchange messages with the verifier. Blum, Feldman and Micali [BFM88] introduced **N**on-**I**nteractive **Z**ero-**K**nowledge proofs (NIZK – pronounced *nee-zeek*). These proofs require a preparation step – the establishment of a common reference string (CRS). This is performed once and CRS is later used for proof generation and verification.

Some NIZKs, such as zk-STARKs [BS+18], don't require the establishment of CRS. In such constructions a pre-agreed hash function (*random-oracle*) plays the role of a CRS. This makes zk-STARKs harder to backdoor. However the large proof size prevents them from being used in a blockchain setting.

3.3 zk-SNARKs

zk-SNARKs are a cryptographic primitive used for zero-knowledge proofs. Their name stands for **Z**ero-**K**nowledge **S**uccinct **N**on-Interactive **Ar**guments of **K**nowledge. They cannot be considered true proofs, but arguments because they hold only computationally. The main strength lies in succinctness and non-interactivity - zk-SNARK proofs are short and can be verified offline. Furthermore, they can be used to prove any statement that can be represented as a circuit (eg. any finite computation on a modern CPU). This can be useful in a number of contexts, not only for cryptocurrencies. (eg. verifiable cloud execution). zk-SNARKs aren't interactive so they require a preparation stage to generate a common reference string (CRS). CRS generation step is the most sensitive part of zk-SNARKs. The randomness used to generate CRS needs to be deleted to prevent the creation of fake proofs.

Most of the material in this section is based on a series of articles by Vitali Buterin [But19b; But19a; But19c] and is meant to bridge the gap between Buterin's articles and full papers. The differences from Buterin's articles, as well as from the original papers are discussed at the end of this section. For further information consult original papers – Pinocchio protocol[Par+13] and Zerocash whitepaper [Sas+14], as well as Zcash blog [Zcah] and Zcash protocol [Hop+19].

3.3.1 Arithmetic Circuit

The calculation we want to perform starts as a program. To generate a proof for it, we need to convert it into an arithmetic circuit over the field $\mathbb{F}$. The gates in the circuit consist of two field operations and their inverses (eg. addition, multiplication, subtraction, and division). Different variables in the program and intermediate results are represented by different wires in the circuit. Conditional execution can be simulated by performing all branches in parallel, and selecting one of them. Finite loops are unrolled and function calls are inlined. An example arithmetic circuit depicting $(a + b) * (c/d)$ can be seen in Figure 3.1.

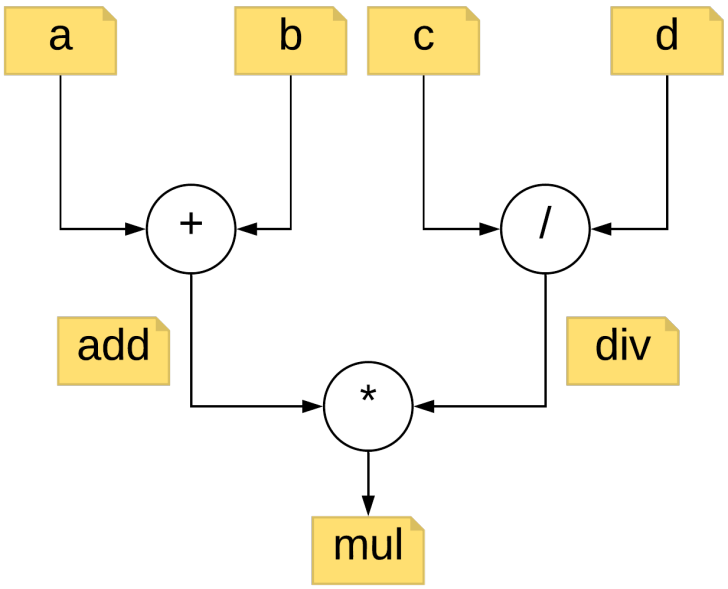

FIGURE 3.1: Example Arithmetic Circuit

3.3.2 R1CS Form

The next step of generating a proof is making sure that all constraints in the circuit are satisfied (eg. that multiplication gates actually perform multiplication). This is achieved by conversion of the circuit into R1CS form (Rank 1 Constraint System). Circuits consist of gates and wires. Gates perform operations and wires carry values. R1CS makes sure that values satisfy constraints set by the gates.

R1CS for a gate consists of 4 vectors: **S**, **A**, **B**, and **C**. Vector **S** represents the assignments of values to all wires in a circuit, and a value 1. It is also called a witness. Vectors **A**, **B**, and **C** represent the constraints. They are filled based on the operation the gate performs, in such way that the constraint in Equation 3.1 is met.

$$\mathbf{A}_i \cdot \mathbf{S} \times \mathbf{B}_i \cdot \mathbf{S} - \mathbf{C}_i \cdot \mathbf{S} = 0 \tag{3.1}$$

If we take a look at the multiplication gate in Figure 3.1, we can construct the following R1CS representation:

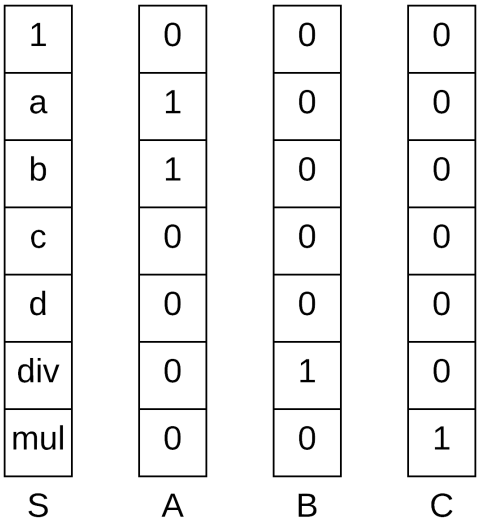

FIGURE 3.2: Multiplication Gate R1CS

The first thing that we notice in Figure 3.2 is that there is no wire marked **add**. Addition (and subtraction) can be folded into multiplication and division gates, so we can reduce the number of wires. Wire **add** has been replaced by inputs of the addition gate **a** and **b**. These values have been assigned value **1** in vector **A** – they appear once in the first argument of multiplication. Wire **div** appears once in right argument (vector **B**), while wire **mul** is marked in the output (vector **C**). Calculating the constraint shows that Equation 3.1 is satisfied.

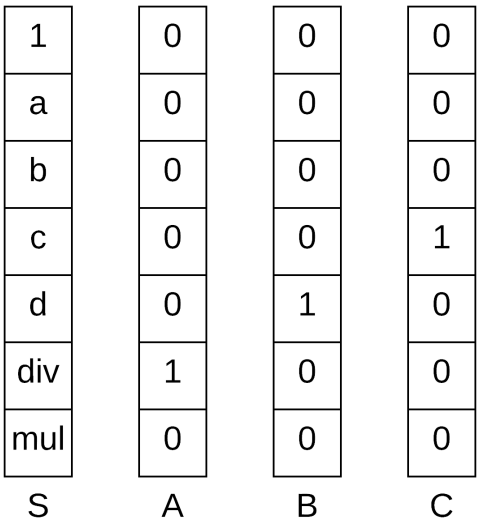

FIGURE 3.3: Division Gate R1CS

Figure 3.3 shows R1CS for the division gate in Figure 3.1. R1CS for this gate actually

guarantees that $\mathbf{div} \cdot \mathbf{d} = \mathbf{c}$, which is equivalent to $\mathbf{c}/\mathbf{d} = \mathbf{div}$. Vector **A** contains the coefficient for **div**, while vector **B** contains the value connected to the divisor **d**. The dividend **c** has been assigned a value of 1 in vector **C**. Equation 3.1 is also satisfied in this case.

3.3.3 Quadratic Arithmetic Program

Quadratic Arithmetic Program enables us to represent R1CS constraints for all gates at once. First, we enumerate all gates, and construct three polynomials (A(x), B(x), and C(x)) for every wire in the circuit. Polynomial for the j-th variable $A_j(x)$ is formed by interpolating it through the points with coordinates $(i, \mathbf{A}_j^i)$. Here, i represents the gate index and $\mathbf{A}_j$ represents the entry assigned to j^{th} wire. Polynomials can be interpolated using well-known algorithm such as Langrangian interpolation. We can recover R1CS vectors by calculating values of the polynomials at the gate's index (Example 3.3.1).

Example 3.3.1. If we want to recover the 2^{nd} gate's value assigned to 4^{th} wire in vector $\mathbf{B} - \mathbf{B}_4^2$, we can do it by computing $B_4(2)$.

Then, we form a new polynomial A(x) by arranging all A_j vectors in a new vector based on the index j, and taking a scalar product with **S**. We form polynomials B(x) and C(x) the same way. The number of wires (variables) in a circuit is denoted by w.

$$A(x) = \sum_{j=1}^{w} \mathbf{S}^j \cdot A_j(x) \tag{3.2}$$

Considering that polynomials A(x), B(x) and C(x) are sums of interpolated polynomials multiplied by the corresponding element in **S**, we can evaluate them at different indices and derive Theorem 3.3.1. The total number of gates in a circuit is denoted by g.

Theorem 3.3.1.

$$\forall t : [t \in \{1, 2, 3, \dots g\} \implies A(t) \cdot B(t) - C(t) = 0]$$

Proof. We know that Equation 3.1 holds for every gate:

$$\forall t : [t \in \{1, 2, 3, \dots g\} \implies \mathbf{A}_t \cdot \mathbf{S} \times \mathbf{B}_t \cdot \mathbf{S} - \mathbf{C}_t \cdot \mathbf{S} = 0]$$

We can expand every scalar product into a sum over all n wires:

$$\forall t : [t \in \{1, 2, 3, \dots g\} \implies (\sum_{j=1}^{n} \mathbf{A}_t^j \cdot \mathbf{S}^j) \times (\sum_{j=1}^{n} \mathbf{B}_t^j \cdot \mathbf{S}^j) - (\sum_{j=1}^{n} \mathbf{C}_t^j \cdot \mathbf{S}^j) = 0]$$

By definition, a polynomial interpolated from some points has the same values when evaluated at those points, so we can substitute them:

$$\forall t : [t \in \{1, 2, 3, \dots g\} \implies (\sum_{j=1}^{n} A_j(t) \cdot \mathbf{S}^j) \times (\sum_{j=1}^{n} B_j(t) \cdot \mathbf{S}^j) - (\sum_{j=1}^{n} C_j(t) \cdot \mathbf{S}^j) = 0]$$

We can use the definition of polynomials A(x), B(x), and C(x) (Equation 3.2) to conclude that Theorem 3.3.1 is true.

$$\forall t : [t \in \{1, 2, 3, \dots g\} \implies A(t) \cdot B(t) - C(t) = 0]$$

□

Theorem 3.3.2. Theorem 3.3.1 implies that the polynomial $A(x) \cdot B(x) - C(x)$ is divisible by the polynomial $T(x) = (x-1)(x-2)\dots(x-g)$, or more formally:

$$A(x) \cdot B(x) - C(x) = T(x) \cdot H(x)$$

Proof. From 3.3.1 we know that 1, 2, 3...g are roots of the polynomial $A(x) \cdot B(x) - C(x)$. We can now use the factor theorem. The first g factors correspond to known roots of the polynomial, while $H(x)$ represents the coefficient:

$$A(x) \cdot B(x) - C(x) = (x-1)(x-2)\dots(x-g) \cdot H(x)$$

The first part is actually just $T(x)$ so we subsitute it, finishing the proof.

$$A(x) \cdot B(x) - C(x) = T(x) \cdot H(x)$$

□

3.3.4 Pinocchio Protocol

This section presumes basic (operational) knowledge of elliptic curves and pairings. In his articles Buterin provides a decent, albeit non-technical, introduction to both elliptic curves and pairings [But19b; But19a; But19c]. For a short black-box treatment of different cryptographic pairings, consult [GPS08]. Readers looking for a more textbook introduction to pairings, can read [Cos12].

The simplest way to verify that all checks that the circuit, or QAP, performs are satisfied, would be to reveal all values. However, that would leak sensitive information such as private keys, available funds, transferred ZEC, etc. Providing just a polynomial divisible by T(x) would be meaningless, because it could have been forged. The primitive we need should provide us with a way to check that the polynomial was constructed in a specific way, while letting us verify that it is indeed divisible by T(x). Pinocchio protocol [Par+13] uses pairings (bilinear maps) to achieve that goal.

In this section, we use a type III pairing: $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$. A pairing is denoted by a two-argument function $e(P_{G_1}, P_{G_2})$. G_1, G_2 and G_T represent generators of groups $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$, respectively. All group operations are represented in the additive notation to stress that we are working with elliptic curve points (+ for the operations in $\mathbb{G}_1$ and $\mathbb{G}_2$, $\oplus$ for $\mathbb{G}_T$, and $[n]P$ for scalar multiplication for all groups).

Example 3.3.2. Pairings can be used to check whether two pairs of elliptic curve points (A and B, C and D) have the same quotients:

$$B/A \stackrel{?}{=} D/C$$

This is verified by computing two pairings:

$$e(A, D) \stackrel{?}{=} e(B, C)$$

In case of $A = [k]B; C = [k]D$ we have:

$$e([k]B, D) \stackrel{?}{=} e(B, [k]D)$$

We use property of the pairing to move the constant outside the pairings:

$$[k]e(B, D) = [k]e(B, D)$$

If the pairs have the same quotients, both pairings will be equal. □

As we mentioned in Section 3.2 that non-interactive proofs must rely either on a common reference string (CRS), or some shared randomness equivalent to CRS (eg. random oracle). Unless the CRS is created correctly, and all private data is deleted after its creation, it is vulnerable to backdooring. Any person that knows that information can create fake proofs. This is where the term trusted setup originates. In order for proofs to be secure, the party creating it must strictly follow the procedure. As we will see in Section 3.4.5, Zcash has used many different techniques to reduce the risk as much as possible.

The most important piece of information that needs to be destroyed after we create CRS, is the coordinate t at which polynomials A(x), B(x) and C(x) will be evaluated. There is also a paradox here to resolve: Although t needs to be secret, we must allow the prover to calculate the values of polynomials at t. This is achieved by multiplying the powers of t by G_1, and adding them to the CRS. These powers will be used later to compute $[H(t)]G_1$. We presume that the discrete logarithm problem (DL) is hard, so the adversary cannot extract t from the powers.

To obtain $A(t), B(t), C(t)$ we need to compute $A_j(t), B_j(t), C_j(t)$ for all gates in the circuit. By also adding these values multiplied by secret coefficients k_a, k_b and k_c to the CRS, and using the idea from Example 3.3.2, we are able to prevent the prover from using arbitrary polynomials. Multiplication by the elliptic curve generator enables us to also perform the needed pairing checks. Values $\rho_A, \rho_B, \beta, \gamma$ are also considered secret, and are used to aid subsequent pairing checks. Accordingly, we add the following values to the trusted setup:

$$\forall j \in \{1, 2, 3, \dots g\} : [\rho_A A_j(t)]G_1, [k_A \rho_A A_j(t)]G_1$$
$$\forall j \in \{1, 2, 3, \dots g\} : [\rho_B B_j(t)]G_2, [k_B \rho_B B_j(t)]G_1$$
$$\forall j \in \{1, 2, 3, \dots g\} : [\rho_A \rho_B C_j(t)]G_1, [k_C \rho_A \rho_B C_j(t)]G_1$$

The prover then needs to provide the following points to prove that they computed polynomials $A(x), B(x), C(x)$ correctly:

$$\pi_A = [\rho_A A(t)]G_1, \pi'_A = [k_A \rho_A A(t)]G_1$$
$$\pi_B = [\rho_B B(t)]G_2, \pi'_B = [k_B \rho_B B(t)]G_1$$
$$\pi_C = [\rho_A \rho_B C(t)]G_1, \pi'_C = [k_C \rho_A \rho_B C(t)]G_1$$

The check is performed by computing:

$$e(\pi_A, [k_a]G_2) \stackrel{?}{=} e(\pi'_A, G_2)$$
$$e([k_b]G_1, \pi_B) \stackrel{?}{=} e(\pi'_B, G_2)$$

$$e(\pi_C, [k_c]G_2) \stackrel{?}{=} e(\pi'_C, G_2)$$

Considering that polynomials $A(x), B(x), C(x)$ are linear combinations of polynomials $A_j(x), B_j(x), C_j(x)$, respectively, required points can be computed from the points in the trusted setup and the witness **S** (**S** represents the coefficients of linear combination). Notice that the verifier needs G_1, G_2, $[k_a]G_2$, $[k_b]G_1$, $[k_c]G_2$ to perform the check, so we also need to add them to the CRS. No one can retrieve k_a, k_b, nor k_c if DL is hard.

The prover must also convince the verifier that they have used the same coefficients for all polynomials. This is achieved by extending the trusted setup with the following points:

$$\forall j \in \{1, 2, 3, \ldots, g\} : [\beta(\rho_A A_j(t) + \rho_B B_j(t) + \rho_A \rho_B C_j(t))]G_1$$

The prover then provides the point:

$$\pi_K = [\beta(\rho_A A(t) + \rho_B B(t) + \rho_A \rho_B C(t))]G_1$$

Pairing check is performed by the verifier:

$$e(\pi_K, [\gamma]G_2) \stackrel{?}{=} e(\pi_A + \pi_C, [\gamma\beta]G_2) \oplus e([\gamma\beta]G_1, \pi_B)$$

Finally, they need to prove that the polynomial $A(x) \cdot B(x) - C(x)$ is really divisible by $T(x)$. This is achieved by using the polynomial $H(x)$ to show that multiplying it by $T(x)$ gives us $A(x) \cdot B(x) - C(x)$. If the prover provides $\pi_H = [H(t)]G_1$, we can do this using pairings:

$$e(\pi_a, \pi_b) \stackrel{?}{=} e(\pi_c, G_2) \oplus e(P_H, [\rho_A \rho_B T(t)]G_2)$$

We will now discuss the simplifications made to the proof. The proof explained works for proofs without any input wires. Adding inputs to the circuit is straightforward. It involves introducing several terms to the CRS and adding them to π_A during the linearity and divisibility checks. The reader can check the full version of Pinocchio zk-SNARKs in [Par+13], or just the review at the end of [BS+14].

There is another important place where we deviate from Buterin's articles: We use asymmetric pairing, instead of A symmetric pairing. Asymmetric pairings are more efficient and make Groth's results in the section 3.4.4 easier to understand.

The final remark has to do with the choice of groups $\mathbb{G}_1$ and $\mathbb{G}_2$ for the points provided by the prover. All points beside π_B are elements of $\mathbb{G}_1$. Elements of $\mathbb{G}_1$ are smaller than the elements of $\mathbb{G}_2$. Pinocchio protocol takes advantage of this to reduce the size of the proof by assign as many points as possible to $\mathbb{G}_1$. π_B is the only point that has to be in $\mathbb{G}_2$ due to the pairing check where we calculate $e(\pi_A, \pi_B)$. This brings us to the proof consisting of 7 $\mathbb{G}_1$ and 1 $\mathbb{G}_2$ element.

3.4 Sapling Update

3.4.1 Introduction

Zcash is undergoing continuous development. The most substantial update to date is called Sapling [Zcag]. Its goal was to overhaul the cryptography of Zcash. The

major changes are a new curve, new commitment scheme, a new proving system, as well as writing these components in Rust. The changes are meant to address the slow speed of shielded transactions, and lead to their wider adoption.

3.4.2 New Curve: BLS12-381

The old curve BN254 saw its conservative security level estimate reduced to 110 bits after a new variant of the number sieve algorithm was presented [Zcaa; KB15]. Increasing the security parameters for this curve would reduce the performance of FFT (for polynomial division) and multiexponentiation steps of proof generation. A new curve was selected - BLS12-381. The embedding degrees of this curve is 12. Other parameters of this curve are its group order $r \approx 2^{255}$ and the base field characteristic $q \approx 2^{381}$.

3.4.3 New Commitment Scheme: Pedersen Commitments and JubJub

Zcash zk-SNARK circuit needs to compute and verify commitments to notes. SHA256 compression function was used as the commitment scheme. Unfortunately, this also bloated the circuit quite a bit. The move to BLS12-381 did not only increase the security level to 128 bits, but also introduce a highly efficient commitment scheme: Pedersen commitments over a custom Edwards curve Ed255-JubJub [Zcab]. The field underlaying JubJub can fit in the scalar field of BLS12-381 and be calculated in the circuit quite efficiently. This change alone reduced shielded transaction generation time by 75% [Zcaf].

3.4.4 New Proving System

While Pinocchio is fullfulls all requirements for Zcash, its biggest problem is the proof size. Instead of using 7 $\mathbb{G}_1$ elements, and one $\mathbb{G}_2$ element, Jens Groth [Gro16] improved upon the previous work to use only 2 $\mathbb{G}_1$ elements and a $\mathbb{G}_2$ element. He also proved that at least one element needs to be in each group for linear circuit checks to be secure.

The new system has the prover provide:

$$\pi_A = [\underbrace{\alpha + A(t) + r\delta}_{X}]G_1$$

$$\pi_B = [\underbrace{\beta + B(t) + s\delta}_{Y}]G_2$$

$$\pi_C = [\frac{\beta A(t) + \alpha B(t) + C(t) + H(t)T(t)}{\delta} + Xs + Yr - rs\delta]G_1$$

The verifier needs to check:

$$e(\pi_A, \pi_B) \stackrel{?}{=} e(\alpha G_1, \beta G_2) \oplus e(\pi_C, \delta G_2)$$

Groth's zk-SNARK has been simplified here and doesn't include inputs to the circuit. Constants $\alpha, \beta, \delta, r, s, t$ are random secrets generated during the generation of CRS. Polynomials A(x), B(x), C(x), H(x), and T(x) are defined in the same way as in Pinocchio [Par+13] protocol.

3.4.5 Rust Implementation

Rust [Rus] was imagined as a memory-safe and thread-safe low-level programming language. Switching to a new curve required a rewrite of the cryptographic code. This was done in Rust to improve the security of Zcash [Zcad]. Furthermore, a new trusted setup was required to define the CRS for Groth's proving system. The program for the ceremony was also written in Rust and used secure multiparty computation to guarantee honesty. 87 people contributed in the first (circuit-agnostic) phase, and over 90 took part in the second phase [Zcae].

Chapter 4

OpenCL

In this chapter we go over the history of hardware that lead to the creation of OpenCL. Afterward, we describe the OpenCL architecture and its memory model.

4.1 History and Introduction

With the predictions of the Moore's Law [Moo+65] coming to an end for single-core processors, a new programming and hardware paradigm was needed to continue the exponential growth. Increasing the frequency of the chip to improve performance would have resulted in overheating (Power Wall). Furthermore, manufacturers continued to reduce the size of a transistor, giving them more computing elements to work with. The answer was reducing the frequency, but using extra transistors to exploit various levels of parallelism present in programs.

CPU manufacturers (Intel, AMD) needed to maintain the performance of single-threaded applications intact while expoliting thread-level parallelism. They opted to have a small number of independent cores in their processors (eg. Intel Dual Core). This was complemented by the invention of hyperthreading [Mar+02] which enabled a single core to execute multiple instruction streams simultaneously.

GPU computing took a different route [McC10]. GPU loads were already massively parallel – small programs were applied to every pixel on the screen. This resulted in GPUs having hundreds of small processor, performing the same computation on different data (SIMD – Single Instruction Multiple Data). Unfortunately, even though their computing power continued to grow, graphics cards were limited to performing only computations limited to image processing. This changed with OpenGL 2.0 standard released in 2004 [SA04]. Even though OpenGL is a graphics API, it enabled users to make use of the massively parallel architecture to solve custom tasks using GLSL (OpenGL Shading Language) [Kes19]. Soon, NVIDIA published their own general-purpose GPU solution – CUDA [Nvi07].

With so much different parallel hardware on the market, interoperability was hard to achieve. Someone having an ATI GPU could not run CUDA programs, so they were forced to buy a new graphics card, or port their software to OpenGL 2.0 which was meant to be used for graphics. In 2009. Apple published OpenCL [Mun09] to provide heterogenous computing support on as many platforms as possible. Only OpenCL architecture is standardized, so the actual, underlying implementation (organization) is left to hardware vendors. This means that OpenCL programs can take full advantage of available hardware – GPUs will execute them in SIMD fashion, FPGA compilers will create only needed SIMD hardware, while multi-core CPUs

will run the program in multiple threads. Today, OpenCL is supported on multi-core CPUs, integrated CPU graphics, GPUs, FPGAs, and by almost all major hardware manufacturers (Intel, AMD, NVIDIA, ARM).

4.2 OpenCL Architecture

Information in this section is based on [Gas+12].

4.2.1 Host and Kernel

An OpenCL program consists of two parts: a host program and a guest program (kernel) (Figure 4.1 [Mun09]). The host program is written in a language which has OpenCL bindings available (C, C++, Python, Rust...), while the kernel is written in OpenCL C or OpenCL C++ (OpenCL 2.1+). The host application interfaces with the operating system to find out which OpenCL devices are available, compiles kernels, queues them for execution and reads back the result. The kernel receives parameters from the host application, performs (usually parallel) operations and returns the result.

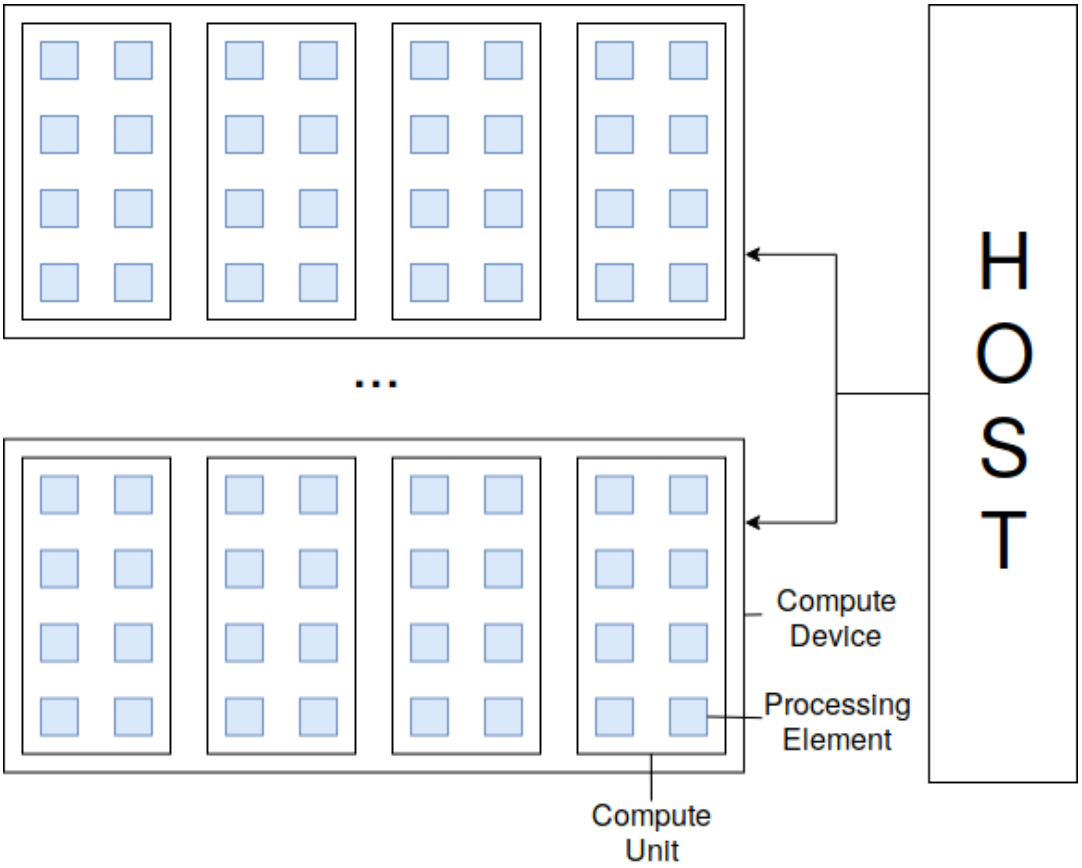

FIGURE 4.1: OpenCL Platform Model

To help parallelization, many instances of the kernel are run, each having a different ID. Based on its ID, the kernel decides which data to fetch, and which operations to execute. Several kernel instances can form a group, which can be used to partially synchronize execution of more complex kernels. The host program determines the total number of kernel instances, as well as the size of a group.

4.2.2 Memory Hierarchy

OpenCL was heavily influenced by CUDA, which is evident in the organization of memory (Figure 4.2 [Mun09]). While CPUs usually automatically handle latency hiding by caching frequent accesses, there are no guarantees that OpenCL devices have cache onboard. The developer must choose where data resides.

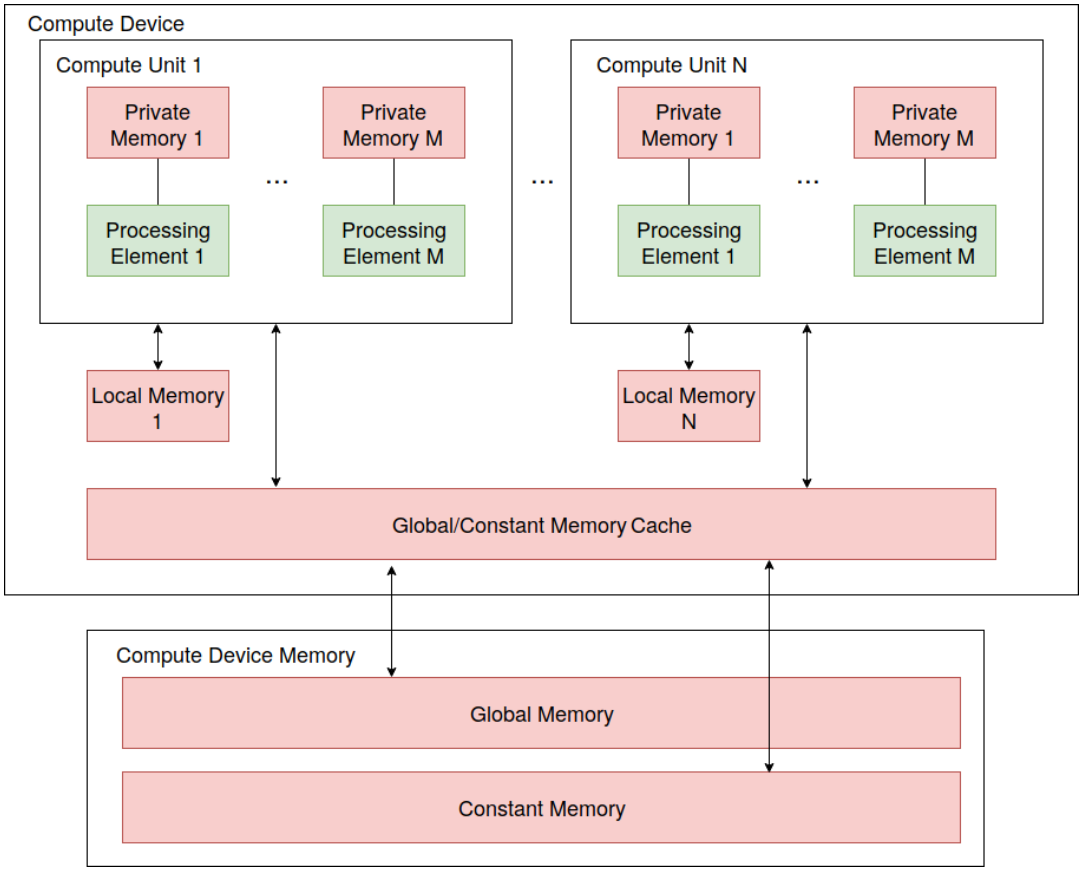

FIGURE 4.2: OpenCL Memory Model

There are three levels of memory, each smaller, faster and accessible by a smaller number of elements than the last. Devices differ in the amount of memory available at each level. If the kernel uses more memory than available, accesses will spill to a lower memory level, slowing down memory accesses.

- Global memory (`__global`) is accessible by every kernel instance, as well as by a host application (using OpenCL calls). This is where host usually writes starting data, and where kernel outputs results. It has the longest latency, but also the highest capacity. Constant memory (`__constant`) is a special part of global memory where read-only data can be stored.
- Local memory (`__local`) is accessible by the elements in the same group.
- Private memory (no keyword) is accessible only by its owner. It is the fastest, but also the smallest memory.

4.2.3 Latency Hiding

While OpenCL is indeed platform-agnostic, organizational details of GPUs affect the performance of OpenCL code running on them. GPUs consist of thousands of

compute units split into groups which execute same instructions in parallel. When a unit issues a data request, the thread blocks until it is served. If there is free space in the register file, the scheduler will start the execution of the next segment. This can happen as many times as needed, as long as there are free registers. If a kernel is well optimized, the GPU will almost never be idle.

This leads to two opposite concepts to keep in mind when writing OpenCL kernels for GPUs. Caching frequently used data in local and private memory (without overflowing to lower hierarchies) reduces latency for subsequent accesses. Unfortunately, this also reduces the number of threads whose state we can keep simultaneously. Balancing these two forces leads to the fastest execution of a kernel.

Chapter 5

Implementation Details

In this chapter we describe our OpenCL implementation of BLS12-381 operations. Afterward, we explain the problem of multiexponentiation and list related literature. Finally, we give outline of three algorithms: Pippenger's algorithm, binary method, and windowed method. We also describe their different implementations in OpenCL. We use multiplication as the group operation in this chapter to make examples easier to follow, and to justify the name of the problem - multiexponentiation.

5.1 Implementation of Elliptic Curve Operations

The Rust implementation of BLS12-381 from the pairing crate [Gitc] used by Zcash was ported to OpenCL C. This step was straight-forward and made easier by the fixed size of primitive data types in both Rust and OpenCL C. Constants were used wherever it was possible, to aid compiler optimization. Almost all functions were inlined to remove the function call overhead and open new opportunities for optimization. Due to the difficulty in debugging on GPUs and cryptographic code, almost every function is unit tested and compared to the Rust implementation.

The host code is written in Rust to interface with the librustzcash Rust library [Gita], and uses ocl crate [Gitb]. The higher level interface is used for testing on Intel, NVIDIA and AMD GPUs. However, cross-compilation for Android some tests had to be implemented using the low-level API.

Finite fields $\mathbb{F}_r$ (255-bit modulus) and $\mathbb{F}_q$ (381-bit modulus) have been ported to OpenCL, as well as an elliptic curve BLS12-381 group $\mathbb{G}_1$ over $\mathbb{F}_q$.

5.2 Multiexponentiation Algorithms

5.2.1 Pippenger's Multiexponentiation Algorithm

The most time-consuming part of proof generation is the multiplication of the elliptic curve points by the coefficients of the witness to create π_A, π_B and π_C. In the multiplicative notation, this multiplication becomes exponentiation, so the algorithms solving this problem can also be called exponentiation algorithms.

The actual problem we are trying to solve is:

Given $x_1, x_2, \ldots, x_n \in \mathbb{G}$ and $y_1, y_2, \ldots, y_n \in \mathbb{Z}$, compute $x_1^{y_1} x_2^{y_2} \ldots x_n^{y_n}$.

While square-and-multiply (Figure 2) and the sliding window method (Figure 3) are asymptotically optimal for a single exponentiation, it is possible to compute the multiexponentiation faster by grouping terms, and exponentiating them together.

Example: $2^3 5^3$ can be computed as $(2 \cdot 5)^3$.

Pippenger's multiexponentiation algorithm [Pip76] can be used to calculate the required product. The algorithm can be used to calculate several multiproducts at once ($x_1^{y_1} x_2^{y_2} \dots x_n^{y_n}$, $x_1^{z_1} x_2^{z_2} \dots x_n^{z_n}$, $x_1^{p_1} x_2^{p_2} \dots x_n^{p_n}$) and Pippenger has proven that the algorithm is asymptotically optimal. However, for zk-SNARKs we require only one multiexponentiation, so the actual implementation is slightly simpler. For background on Pippenger's algorithm consult [Hen10]. Another paper on Pippenger, as well as some alternatives (such as Yao's [Yao76] and Brauer [Bra39]) is [Ber02]. For discussion and computational comparison of several methods (including Bos-Coster algorithm [BC89]) check [BBB94]. In the rest of the section, we will discuss the version of the Pippenger from bellman Rust crate (Algorithm 1).

Algorithm 1 Simplified Pippenger

```
1: function MULTIEXPINNER(Bases[1 ... n], Exponents[1 ... n], Shift, Width)
2:     if Shift + Width < EXP_BITS then
3:         Higher ← new thread MULTIEXPINNER(Bases, Exponents, Shift+Width,
   Width)
4:     end if
                                                  ▷ Bucket creation and assignment
5:     Buckets ← new Base[1 ... 2^Width − 1]
6:     Mask ← 2^Width − 1
7:     for i ← 1 ... n do
8:         ExpPart ← (Exponents[i] ≫ Shift) & Mask
9:         if ExpPart ≠ 0 then
10:            Buckets[ExpPart] ← Buckets[ExpPart] · Bases[i]
11:        end if
12:    end for
13:    PartialSum ← 0
14:    Sum ← 0
                                                  ▷ Smart multiplication step
15:    for i ← 2^Width − 1 ... 1 do
16:        PartialSum ← PartialSum · Buckets[i]
17:        Sum ← Sum · PartialSum
18:    end for
                                                  ▷ Chunk combination step
19:    if Shift + Width < EXP_BITS then
20:        wait Higher
21:        for i ← Width ... 1 do
22:            Higher ← Higher · Higher
23:        end for
24:        Sum ← Sum · Higher
25:    end if
26:    return Sum
27: end function
28:
29: function MULTIEXP(Bases[1 ... n], Exponents[1 ... n])
30:    if n < 32 then
31:        Width ← 3
32:    else
33:        Width ← ln(n)
34:    end if
35:    return MULTIEXPINNER(Bases, Exponents, 0, Width)
36: end function
```

Instead of exponentiating bases separately, and them muliplying them together, Algorithm 1 does the opposite. It segments exponents into chunks (eg. bits 3-0) and adds the bases to corresponding buckets based on the value of the chunk. This results a bucket containing all bases that need to be exponentiated to the same power. Considering that the number of buckets is smaller than the number of bases, we should save quite a bit of time.

It is actually possible to do even better. We don't need to exponentiate buckets separately. Considering that all buckets have sequential powers we need to calculate, we can do it in the following way:

$$a \cdot b^2 \cdot c^3 = c \cdot (c \cdot b) \cdot (c \cdot b \cdot a) \tag{5.1}$$

We can run this algorithm in multiple threads – assigning a different chunk to every thread. In the end, we just merge the results by squaring a higher chunk enough times, and multiplying it with a lower one.

As we can see this algorithm is quite parallelizable. We just need to change the width of the bit chunk to distribute work over more threads. However, the hard limit to this is the number of bits in the exponent. In case that we have more processors than bits in the exponent, we can split bases we want to exponentiate in two parts. Instead of iterating though all bases, threads would iterate through bases in the segment assigned to them. There would also be an additional step of multiplying results from all segments together. Unfortunately, the smart multiplication from Equation 5.1 will also be replicated.

5.2.2 Implemented Algorithms

Square and Multiply (Binary Method)

Square-and-multiply[1] [Knu14] (Algorithm 2) multiexponentiation kernel implements multiexponentiation in a simple way. Every elliptic curve point and exponent is loaded and exponentiated separately. The exponentiation is performed by iterating through the exponent left-to-right. For every bit we square the result[2] and multiply by the base[3] if the bit is 1. After we process all bases, we call the reduction algorithm to produce the final result.

[1] Double-and-add in additive groups
[2] We double in an additive group
[3] We add the base in an additive group

Algorithm 2 Square and Multiply Algortihm

```
1:  function SAM(Base, Exponent)
2:      Result ← 1
3:      for i ← Width ... 1 do
4:          Result ← Result^2
5:          if Exponent[i] = 1 then
6:              Result ← Result · Base
7:          end if
8:      end for
9:      return Result
10: end function
11:
12: function MULTIEXP(Bases[1 ... n], Exponents[1 ... n])
13:     Result ← 1
14:     for i ← 1 ... n do
15:         Result ← Result·SAM(Bases[i], Exponents[i])
16:     end for
17:     return Result
18: end function
```

Sliding Window Method

Sliding window method [Knu14] (Algorithm 3) is a generalization of square-and-multiply. It also works with every point and exponent separately, but it processes the exponent 4 bits at a time. First, a lookup table is generated for the powers of the current base (powers 2-15). For every chunk of the exponent read, it performs the needed amount of squarings and multiplies the result by the corresponding power. This method uses more memory than ordinary square-and-multiply, but the number of operations is lower. Just like in the binary method, we perform reduction to compute the final result.

Algorithm 3 Sliding Window Method

```
1:  function SWM(Base, Exponent)
2:      PowerTable[1 ... Chunk] ← newBase[1 ... Chunk]
3:      CurrentPower ← 1
4:
5:      for i ← 1 ... Chunk do
6:          CurrentPower ← CurrentPower · Base
7:          PowerTable[i] getsCurrentPower
8:      end for
9:
10:     Result ← 1
11:     for i ← Width/Chunk ... 1 do
12:         Bits ← Get i^th Chunk From Exponent
13:         for j ← 1 ... Chunk do
14:             Result ← Result^2
15:         end for
16:
17:         if Bits ≠ 0 then
18:             Result ← Result · PowerTable[Bits]
19:         end if
20:     end for
21:
22:     return Result
23: end function
24:
25: function MULTIEXP(Bases[1 ... n], Exponents[1 ... n])
26:     Result ← 1
27:     for i ← 1 ... n do
28:         Result ← Result·SWM(Bases[i], Exponents[i])
29:     end for
30:     return Result
31: end function
```

Four-bit Pippenger's Algorithm

This kernel is the closest to the Rust implementation of Pippenger's algorithm (Algorithm 1). Points are segmented to take advantage of thousands of processors on the GPU. Every thread then iterates through the points in the segment and reads the four-bit window (chunk) in the exponent based on the thread index. Point is then added to the corresponding bucket (Algorithm 1 - Bucket assignment step). We then add the buckets together (Algorithm 1 - Smart multiplication step) write the result to shared local memory.

At this point, values corresponding to different bit offsets are reduced to the value of the segment by squaring and multiplying them enough times (Algorithm 1 - Chunk combination step). The segment value is then written to global memory, and a reduction algorithm is called to combine all segments.

One-bit Pippenger's Algorithm

In this implementation of Pippenger's algorithm (Algorithm 1), points are divided into multiple chunks. Kernel threads are grouped in blocks of 256, and every thread is assigned to a bit in the exponent. Every thread iterates through the points in its segment and adds all points where the tracked bit is one. When they are finished, threads write the results to the global memory. These sums are then reduced based on the bit in the exponent which was assigned to them. Reduction results in 256 values, which are then multiplied using the efficient multiplication algorithm (Equation 5.1).

Four-bit Pippenger's Algorithm with Separate Reduction

Four-bit Pippenger's Algorithm (Algorithm 1) kernel has an advantage that it performs fewer operations than the one-bit kernel. However, it features a local shared memory reduction step done on GPU, during which many processing units could be left without any work. This kernel starts as Four-bit Pippenger's algorithm, but writes the result of each thread to global memory, instead of reducing them. At the very end, they are reduced to 64 powers, and combined on CPU to get the final result.

Chapter 6

Results

In this chapter we first describe the hardware that was used for benchmarking. Afterward, we use the opportunity to discuss the problems involved in developing code for different OpenCL devices. Finally, we outline the tests we ran, and report the results.

6.1 Test Preparation

6.1.1 Hardware

The CPU used for testing desktop performance is an **Intel i7 7700HQ** @ 2.80 GHz with 4 cores (8 virtual cores) [Intb]. Both 32-bit and 64-bit code was tested on this CPU, compiled using Rust compiler with corresponding targets. The operating system running is Ubuntu 18.10.

For testing on mobile devices, we used **Samsung S9's Exynos** processor. It has 8 cores: 4 cores run at 2.9 GHz, and the other four at 1.9 GHz [Exy]. Both 32-bit and 64-bit code was tested on this processor, using corresponding targets for the Rust compiler, and Android NDK (`armv7a-linux-androideabi16` for 32-bit code, and `aarch64-linux-android21` for 64-bit code).

Intel HD Graphics 630 is a GPU integrated with 7700HQ. It has 24 processing units working at a frequency of 350 MHz during normal operation, and 1.1 GHz burst frequency [Inta]. It has access to system RAM. NEO Linux drivers were used.

Mali-G72 MP18 is a GPU integrated with Samsung's Exynos processor on Galaxy S9. It has 18 processing units working at 850 MHz [Mal]. It has access to system RAM.

NVIDIA GTX 1060M 6GB is a GPU with 1280 computation units working at a frequency of 1404 MHz (1670 MHz boost frequency). It houses 6 GB of global memory. 1280 CUDA cores are divided equally among 10 SMs (streaming microprocessors). Each SM has 256 KB of private memory, 96 KB of shared memory, as well as a 48 KB L1 cache [Nvic]. Proprietary NVIDIA Linux drivers were used.

AMD RX 580 is a GPU with 2304 streaming processors (SP) grouped in 36 Compute Units (CU) operating at 1257 MHz (1340 MHz boost). Every streaming processor has 256 vector registers and 512 scalar registers, each 4 bytes wide (64 KB + 8 KB total). Every compute unit also has 64 KB of shared memory and a complex cache hierarchy [Amda].

6.2 OpenCL Support on Different GPUs

Considering that only 6% of OpenCL papers test the program on 3 or more different platforms [SD16], we will take this opportunity to list the difficulties we've encountered. For all GPUs we used ocl Rust crate to run OpenCL code.

6.2.1 NVIDIA

Setup

NVIDIA driver officially supports only OpenCL 1.2. Recently, there has been some progress towards the more modern standard, with NVIDIA quietly enabling OpenCL 2.0 [Nvif]. NVIDIA has been pushing its GPGPU solution CUDA over OpenCL, and OpenCL is expected to perform slightly slower than equivalent CUDA code [FVS11; KDH10]. This is also visible in dropping support for OpenCL code profiling that was present in older driver versions [Nvia]. However, we've managed to get static kernel data and assembly by dumping the high-level assembly (PTX file) and compiling it for the architecture that our card supports (Compute Capability 6.1). CUDA occupancy calculator was then used to estimate how many threads' states could be saved at the same time.

Runtime Bugs

There was one bug in the compiler that we didn't expect. When compiling a kernel for Four-bit Pippenger, when we loaded only the needed portion of the exponent, we got `CL_OUT_OF_RESOURCES` error[1]. The same kernel worked without a problem on Intel's GPU. The error was fixed by reading the entire exponent structure from global memory, instead of only reading the integer that we needed.

6.2.2 Intel

Setup

Intel's NEO driver supports OpenCL 2.1. Running longer kernels (more than 10s) required us to manually disable the watchdog timer in the kernel. Unfortunately, because Intel was the main display device, this caused our device to freeze until kernels finished execution. Due to extremely slow execution speed, we haven't profiled code executing on this GPU.

Runtime Bugs

The biggest issue was encountered when we manually unrolled the square-and-multiply loop (255 iterations). Intel kernel kept outputting the wrong result, but adding `printf` somehow fixed this problem. NVIDIA GPU outputted the correct result for the same code without any issues. Old Intel driver didn't support `printf` for 64-bit integers on Ubuntu, but switching to NEO solved this problem.

[1] It seems that CL_OUT_OF_RESOURCES is treated as a generic error

6.2.3 Mali

Setup

Mali supports OpenCL 2.1. Compiling OpenCL code for Samsung Galaxy S9 required us to dump OpenCL.so from the phone, and link it during compilation. 32-bit code compiled properly even with `ld` loader, but 64-bit version required us to use `gold`. It is also interesting that mobile GPUs need low-level ocl crate API (a wrapper around C code). After cross-compiling the binary, which involved some basic symlinking to resolve name conflicts during Rust crate compilation, we used `adb` and `adbshell` to copy and run code on the device.

Runtime Bugs

Unfortunately, a compilation of OpenCL kernel results in a `Segmentation Fault` after a couple of minutes for the 32-bit version. Another problem that we encountered with Mali is the unexpected use of OpenCL API. All other vendors compile the kernel when we call `clBuildProgram`, while Mali does this during the kernel call. While testing our kernels, we noticed that the application may get killed if the kernel takes too long.

6.2.4 AMD

Setup

AMD has the best support for OpenCL of all vendors tested. The only problem we encountered is that profiler GUI wouldn't start, but CLI version had no issues.

Runtime Bugs

We haven't encountered any bugs with AMD cards.

6.2.5 Conclusion

OpenCL theoretically enables us to write code only once and run it on different parallel devices. Unfortunately, due to differences in vendors' implementations, it is necessary to test the code on every single device we plan to support. Even then, the code may need to be adapted just to run. Additional debugging is also frequently required, even though code runs properly on other devices. This can pose a problem if the platform provides limited debugging and profiling possibilities (e.g. NVIDIA).

6.3 Tests

6.3.1 Whole Proof Generation

This test involved dumping the data from the actual spending proof generation and running it multiple times on both CPUs with both 32 and 64-bit code. Spending proof has been chosen because it takes longer to generate than the output proof. We also performed a stress test to determine the drop of performance under heavy load. As we see in Figure 6.1, 32-bit code is significantly slower than 64-bit code. Also, 64-bit ARM code is comparable in performance with x86-64. There is also a slightly

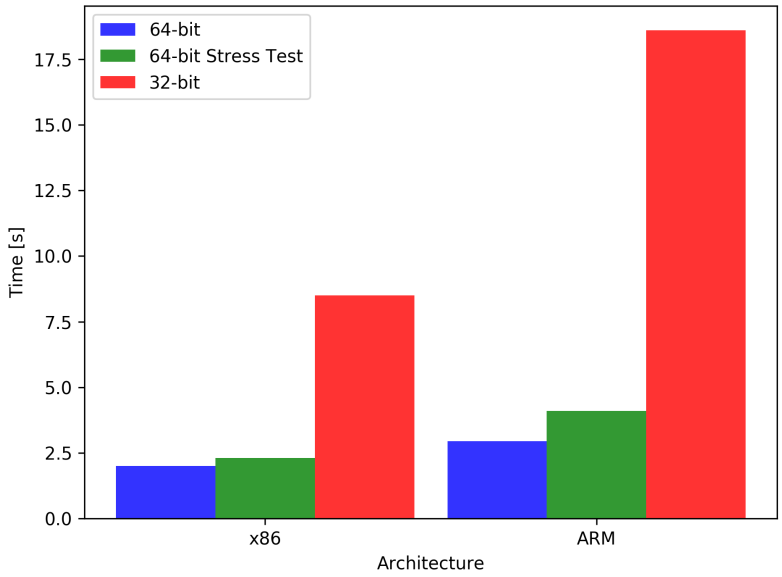

FIGURE 6.1: Benchmark of Spending Proof on Various Hardware

larger drop in average performance during the stress test for ARM than for Intel. However, Intel performance was consistent, while the worst-case ARM performance dropped down to 6.75s.

6.3.2 FFT and Multiexp

To determine the best target for optimization we benchmarked different parts of spending proof generation on Intel i7 processor. We benchmarked FFT, Multiexp and witness generation.

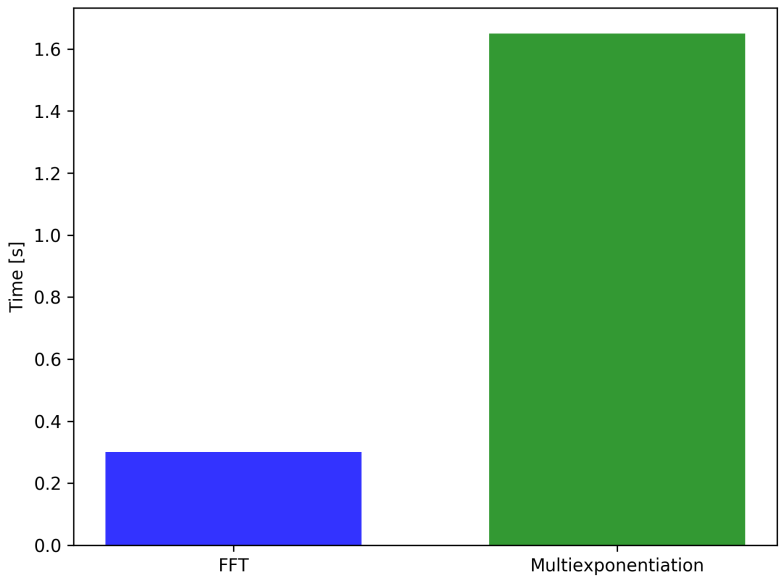

FIGURE 6.2: Benchmark of FFT and Multiexponentiation in a Spending Proof

CPU spends the most time performing FFT and multiexponentiation. Witness generation takes several milliseconds at most. In Figure 6.2 we can see that multiexponentiation takes 85% time of proof generation. This made multiexponentiation a natural target for optimization. To determine potential for parallelization, we also benchmarked single-core performance and plugged in the numbers into Amdahl's law. This gave us 88.5% of parallel code (in theory).

6.3.3 131k Test

We decided to benchmark our multiexponentiation algorithms on group G1, with 131k points and exponents. This is the size of the largest array that gets processed. $\mathbb{G}_1$ group was selected because it is simpler to implement and is used more often than $\mathbb{G}_2$. The data for integrated GPUs (Intel and Mali) shown in Figure 6.3 has been

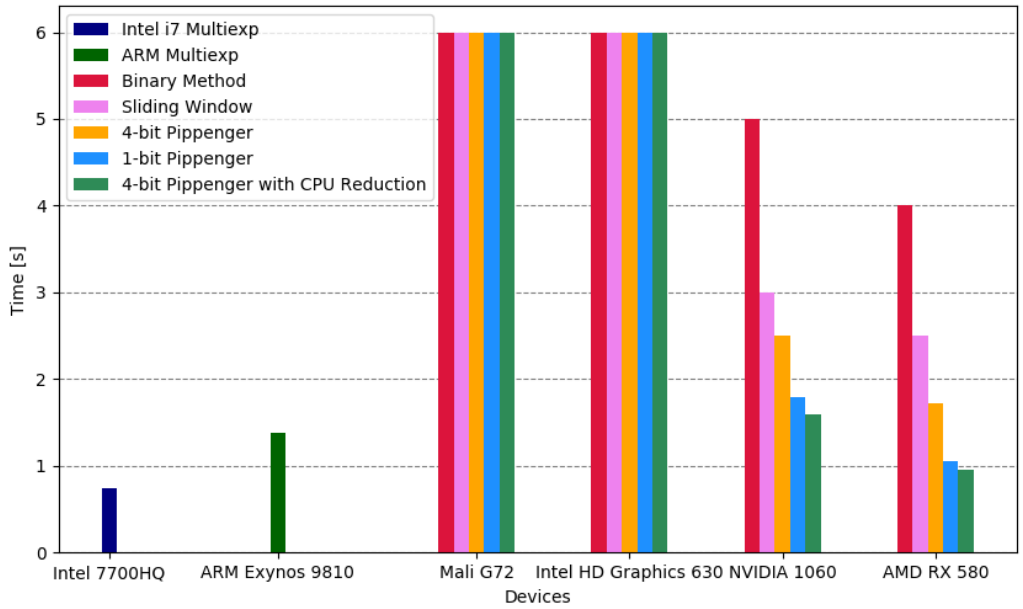

FIGURE 6.3: Comparison of Algorithms Implemented Running on Various Hardware

cut off at 6 seconds. The actual times are much higher – several minutes for Mali and 10.4 seconds for the best Intel kernel. NVIDIA and AMD perform better, but still not at the level of CPUs. Square-and-multiply takes the most time, followed by windowed-square-and-multiply. Variants of Pippenger's algorithm are the fastest, with 4-bit Pippenger with CPU reduction taking the first spot. Unfortunately, even it fails to overtake multiexponentiation on Intel i7.

Chapter 7

Discussion

In this chapter, we analyze the results of performance measurements. We discuss the effects of processor word width on performance, as well as the performance on mobile devices. Finally, we talk about the different factors which affected the execution time of multiexponentiation on GPUs.

7.1 Whole Proof Generation

We were quite surprised to discover that performance on mobile ARM processors is almost as good as as good as a full blown desktop CPUs from Intel. Performance on the i7 processor is a bit better, however, ARM's 8 cores make the performance gap really narrow.

Another interesting insight is the importance of the processor word width. 32-bit code was slower than its 64-bit counterpart on both architectures. This performance penalty is incurred because of arithmetic operations. zk-SNARKs rely on 381-bit finite prime field arithmetic. Emulating these operations requires fewer 64-bit than 32-bit operations. For 32-bit code every 64-bit addition needs to be decomposed into 2 32-bit additions. Every multiplication requires 3 32-bit multiplications (4 if we want to keep the high bits). On top of this, there are several ADD instructions needed to compute the final product (Figure 7.1).

```
push   esi
mov    ecx, dword ptr [esp + 16]
mov    esi, dword ptr [esp + 8]
mov    eax, ecx
imul   ecx, dword ptr [esp + 12]
mul    esi
imul   esi, dword ptr [esp + 20]
add    edx, ecx
add    edx, esi
pop    esi
ret
```

```
mov    rax, rdi
imul   rax, rsi
ret
```

FIGURE 7.1: Assembly of the function that multiplies two 64-bit numbers on x86 (left) and x64 (right)

The stress test showed that phones are prone to overheat, and throttle the computation of proofs if under heavy load. This happens after several proofs have been computed, even if nothing else is running on the device. Laptops don't show this

behavior due to the fans that disperse the heat. Note that the test was performed on a laptop with GPU turned off.

7.2 FFT and Multiexp

The results of this test showed no surprising facts. Multiexponentiation and FFT are the most time-consuming parts of zk-SNARKs. Parallelizing FFT would lead to minimal performance gains due to it taking only 15% of execution time. Considering that the simplified Pippenger's algorithm used for multiexponentiation is inherently parallelizable, it lent itself as the perfect candidate for GPU acceleration.

7.3 131k Test

Integrated GPUs performed poorly when compared to CPUs and discrete GPUs. Considering that mobile devices have only integrated GPUs, this eliminates GPU accelerated zk-SNARK generation on them. If we take a look at frequencies and core counts of discrete and integrated GPUs, it is clear that Intel and Mali GPUs will be at least an order of magnitude slower than NVIDIA and AMD cards. The actual surprise is that not even discrete GPUs can beat CPUs running Pippenger's algorithm. In this section, we explain multiple contributing factors.

7.3.1 Processor Word Length – 32 vs 64

As we concluded in Section 7.1, processor word length plays an important role in zk-SNARK performance. With GPUs, this effect is further amplified by the memory hierarchy. All of today's GPUs are still clusters of 32-bit RISC cores operating in unison [Nvie; Amdc]. Small word length means that these cores need to execute several times more instructions than CPUs running equivalent code. It is interesting to note that GPUs are optimized towards floating-point numbers, leading to lower throughput for integer operations [Nvid].

7.3.2 Branch Divergence

A common problem with code running on GPUs is branch divergence. When CPU encounters a branch in the code, it will execute only one of two code paths. Unfortunately, all GPU cores in a group need to execute the same instruction. To satisfy this, GPUs will execute both branches, and tell cores to ignore the instructions which don't satisfy their condition check.

However, profiling on our AMD card showed thread convergence rate of 85%. Improving this further would lead to minimal gains in performance.

7.3.3 Inlining, Loops and Constants

While functions are considered good practice on CPUs, using them on GPUs can significantly hinder performance. Function calls require a call stack. Unfortunately, this stack is stored in global memory. For small kernels, this isn't a problem because

the stack can be cached. However, our kernels have elliptic curve points as arguments, so the stack is too big to fit in the cache. Another problem is that Pippenger's algorithm requires an array of elliptic curve points with random access [Nvib]. Kernel disassembly of NVIDIA code showed that these arrays are stored on the stack (i.e. in global memory).

Inlining function calls and unwrapping loops reduces divergence and removes the stack. However, it substantially increases the size of the code. This, combined with 32-bit registers, results in code that is too large to fit in the instruction cache, incurring a performance penalty. Analysis of AMD disassembly shows that the kernels were around 90 KB in size, while the instruction cache has only 30 KB [Amdb]. On the other hand, the NVIDIA compiler actively tried to prevent code explosion by keeping some functions intact, even though they had been marked inline. Unfortunately, this resulted in an enormous call stack.

OpenCL uses global memory to store constants. Usually, they can be cached, so the performance penalty doesn't need to be high. However, if the kernel has a lot of memory traffic, constants will be pushed out of the cache and will be fetched every time they are used. To prevent this, we used macros in place of variables, hoping that compilers will use immediate load instructions instead of memory reads. Unfortunately, both NVIDIA and AMD compilers detected constants and put them in global memory.

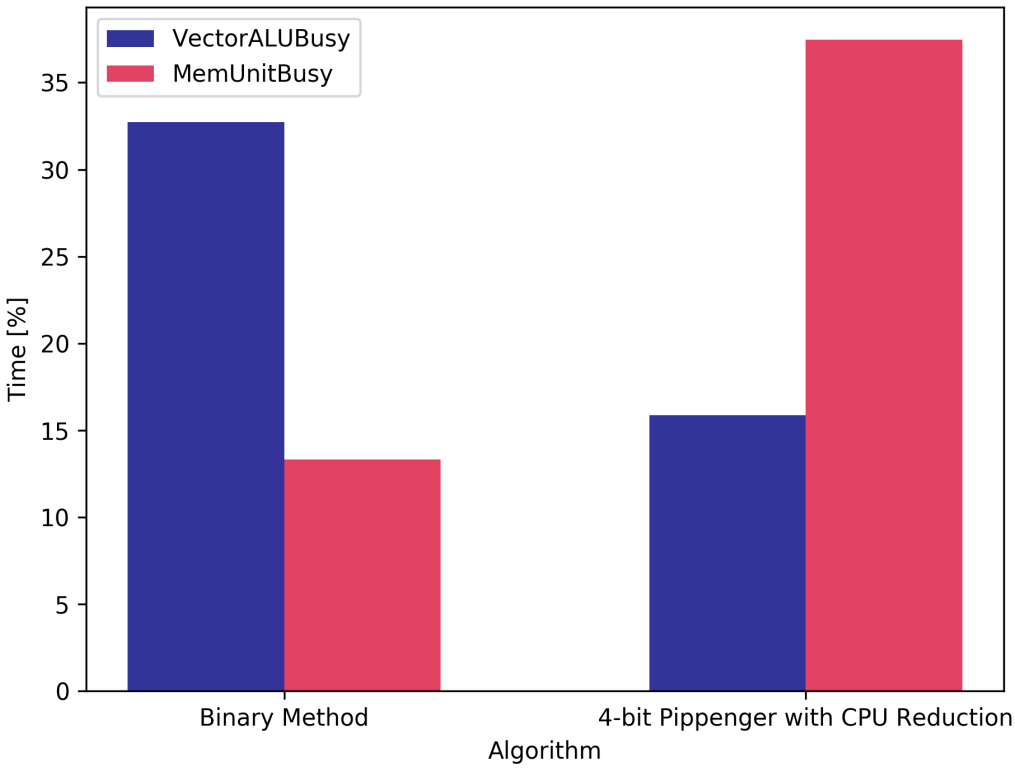

FIGURE 7.2: Memory and ALU Activity of Binary Method and 4-bit Pippenger with CPU Reduction Running on AMD RX 580

All of these things, combined with high register usage of kernels (more than 160 registers on NVIDIA, 256 on AMD), results in high memory traffic and low occupancy

for Pippenger's algorithm on GPUs. Figure 7.2 shows data from AMD profiler - vector ALU and memory unit usage for the binary method and Pippenger's algorithm. Even though Pippenger's algorithm is faster, it barely uses ALU, compared to memory. CPUs can run the same algorithm with much larger array sizes without incurring a performance penalty, leading to better performance.

7.3.4 Performance Gains

It is possible to attain a speed-up of 25-30% by running CPU and GPU together and distributing work between them. This scheme would involve putting (a smaller) part of the exponent on GPU, exponentiating the rest on CPU, and combining the results. Unfortunately, due to different ratio's of CPU and GPU power, we would need to maintain either a table for most common combinations or run a test first to determine the optimal ratio.

However, the gains are minimal even with more powerful GPUs. Graphic cards draw a lot of power, heat up the machine and increase fan noise. This scheme would require us to maintain two versions of the same code and would be economical only if we already own a strong GPU. If someone wants to buy a machine specifically for Zcash, it would make more sense to buy a processor with more cores.

Chapter 8

Further Work

Considering that OpenCL compilers generate large, memory-intensive GPU code, it may be possible to program Pippenger's algorithm in GPU assembly. [Ber+10] have shown that we would need to use device-specific assembly to take advantage of all registers and control spills. This excludes intermediate assembly such as NVIDIA's PTX and would increase the load on developers. Programming an algorithm of this size in assembly often leads to cryptic, unmaintainable, and insecure code, which would go against general security engineering principles. Code written like this would work only on a single device (or a family of devices), severely limiting its usefulness.

The more viable solution involves creating custom hardware that supports 381-bit operations. These compute units would also have enough cache to store arrays for Pippenger's algorithm, as well as intermediate values. The biggest problem here is the design of the multiplier, although there are several available and tested proposals [Bri+; Qua+05]. VHDL and Verilog can be used to implement the design on an FPGA. Using FPGAs would lead to lower frequency compared to GPU, but better allocation of silicon (eg. no need for floating-point hardware). Hardware wallet implemented like this could have higher performance and lower power consumption than commercial CPUs.

Chapter 9

Conclusion

In this thesis, we have given an overview of zk-SNARKs used in Zcash. Afterward, we described the OpenCL architecture and problems we encountered with different vendors' implementation of it, before moving on to algorithms for multiexponentiation of BLS-381 points that we implemented in OpenCL. Finally, we reported the results of tests performed and showed possibilities for further research.

In our tests, we have found out that 64-bit ARM processors offer performance comparable to Intel's i7 processors for Zcash proof generation. We have also noted that due to the lack of cooling on phones, the performance tends to oscillate.

Ultimately, we have shown that our implementation of BLS-381 G1 multiexponentiation algorithm in OpenCL is slower on all tested vendors' GPUs than the baseline CPU implementation. We have attributed this to high memory traffic caused by GPU's limited memory resources, small caches, large sizes of generated code caused by short processor word length, as well as quirks in the register and memory allocation of OpenCL compilers.

Bibliography

[ABS10] Samuel Antao, Jean-Claude Bajard, and Leonel Sousa. "Elliptic curve point multiplication on GPUs". In: *ASAP 2010-21st IEEE International Conference on Application-specific Systems, Architectures and Processors*. IEEE. 2010, pp. 192–199.

[Amda] *AMD RX 580 Specifications*. May 2019. URL: https://www.amd.com/en/products/graphics/radeon-rx-580.

[Amdb] *Graphics Core Next Architecture*. May 2019. URL: http://developer.amd.com/wordpress/media/2013/06/2620_final.pdf.

[Amdc] *Graphics Core Next Architecture, Generation 3 Reference Guide*. May 2019. URL: http://developer.amd.com/wordpress/media/2013/12/AMD_GCN3_Instruction_Set_Architecture_rev1.1.pdf.

[BBB94] F Bergeron, Jean Berstel, and S Brlek. "Efficient computation of addition chains". In: *Journal de théorie des nombres de Bordeaux* 6.1 (1994), pp. 21–38.

[BC89] Jurjen Bos and Matthijs Coster. "Addition chain heuristics". In: *Conference on the Theory and Application of Cryptology*. Springer. 1989, pp. 400–407.

[Ber02] Daniel J Bernstein. "Pippenger's Exponentiation Algorithm". In: (2002).

[Ber+10] Daniel J Bernstein et al. "ECC2K-130 on Nvidia GPUs". In: *International Conference on Cryptology in India*. Springer. 2010, pp. 328–346.

[BFM88] Manuel Blum, Paul Feldman, and Silvio Micali. "Non-interactive zero-knowledge and its applications". In: *Proceedings of the twentieth annual ACM symposium on Theory of computing*. ACM. 1988, pp. 103–112.

[BHC17] Thibault de Balthasar and Julio Hernandez-Castro. "An analysis of bitcoin laundry services". In: *Nordic Conference on Secure IT Systems*. Springer. 2017, pp. 297–312.

[BKP14] Alex Biryukov, Dmitry Khovratovich, and Ivan Pustogarov. "Deanonymisation of clients in Bitcoin P2P network". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2014, pp. 15–29.

[BLP17] Fábio Borges, Pedro Lara, and Renato Portugal. "Parallel algorithms for modular multi-exponentiation". In: *Applied Mathematics and Computation* 292 (2017), pp. 406–416.

[Bra39] Alfred Brauer. "On addition chains". In: *Bulletin of the American mathematical Society* 45.10 (1939), pp. 736–739.

[Bri+] Riadh Brinci et al. "Efficient Hardware Design for Computing Pairings Using Few FPGA In-built DSPs." In: ().

[BS+14] Eli Ben-Sasson et al. "Succinct non-interactive zero knowledge for a von Neumann architecture". In: *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 2014, pp. 781–796.

[BS+18] Eli Ben-Sasson et al. "Scalable, transparent, and post-quantum secure computational integrity." In: *IACR Cryptology ePrint Archive* 2018 (2018), p. 46.

[But19a] Vitalik Buterin. *Exploring Elliptic Curve Pairings*. May 2019. URL: https://medium.com/@VitalikButerin/exploring-elliptic-curve-pairings-c73c1864e627.

[But19b] Vitalik Buterin. *Quadratic Arithmetic Programs: from Zero to Hero*. May 2019. URL: https://medium.com/@VitalikButerin/quadratic-arithmetic-programs-from-zero-to-hero-f6d558cea649.

[But19c] Vitalik Buterin. *Zk-SNARKs: Under the Hood*. May 2019. URL: https://medium.com/@VitalikButerin/zk-snarks-under-the-hood-b33151a013f6.

[CL97] Chin-Chen Chang and Der-Chyuan Lou. "Parallel computation of the multi-exponentiation for cryptosystems". In: *International Journal of Computer Mathematics* 63.1-2 (1997), pp. 9–26.

[Cos12] Craig Costello. *Pairings for beginners*. 2012.

[Exy] *Exynos 9810 SoC Specifications*. May 2019. URL: https://www.samsung.com/semiconductor/minisite/exynos/products/mobileprocessor/exynos-9-series-9810/.

[FVS11] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. "A comprehensive performance comparison of CUDA and OpenCL". In: *2011 International Conference on Parallel Processing*. IEEE. 2011, pp. 216–225.

[Gas+12] Benedict Gaster et al. *Heterogeneous computing with OpenCL: revised openCL 1.2 Edition*. Newnes, 2012.

[Gita] *librustzcash Rust Crate Github Repository*. May 2019. URL: https://github.com/zcash/librustzcash.

[Gitb] *ocl Rust Crate Github Repository*. May 2019. URL: https://github.com/cogciprocate/ocl.

[Gitc] *pairing Rust Crate Github Repository*. May 2019. URL: https://github.com/zcash/librustzcash/tree/master/pairing.

[GMR85] S Goldwasser, S Micali, and C Rackoff. "The knowledge complexity of interactive proof-systems". In: *Proceedings of the seventeenth annual ACM symposium on Theory of computing*. ACM. 1985, pp. 291–304.

[GPS08] Steven D Galbraith, Kenneth G Paterson, and Nigel P Smart. "Pairings for cryptographers". In: *Discrete Applied Mathematics* 156.16 (2008), pp. 3113–3121.

[Gro16] Jens Groth. "On the size of pairing-based non-interactive arguments". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2016, pp. 305–326.

[Hen10] Ryan Henry. *Pippenger's Multiproduct and Multiexponentiation Algorithms*. Tech. rep. Citeseer, 2010.

[Hen+14] Sylvain Henry et al. "Toward OpenCL automatic multi-device support". In: *European Conference on Parallel Processing*. Springer. 2014, pp. 776–787.

[Hop+19] Daira Hopwood et al. "Zcash Protocol Specification". In: (May 2019). URL: https://github.com/zcash/zips/blob/master/protocol/protocol.pdf.

[Inta] *Intel HD Graphics 630 Specifications*. May 2019. URL: https://www.intel.com/content/www/us/en/support/products/98909/graphics-drivers/graphics-for-7th-generation-intel-processors/intel-hd-graphics-630.html.

[Intb] *Intel i7 7700HQ Specifications*. May 2019. URL: https://ark.intel.com/content/www/us/en/ark/products/97185/intel-core-i7-7700hq-processor-6m-cache-up-to-3-80-ghz.html.

[KB15] Taechan Kim and Razvan Barbulescu. *Extended Tower Number Field Sieve: A New Complexity for the Medium Prime Case*. Cryptology ePrint Archive, Report 2015/1027. https://eprint.iacr.org/2015/1027. 2015.

[KDH10] Kamran Karimi, Neil G Dickson, and Firas Hamze. "A performance comparison of CUDA and OpenCL". In: *arXiv preprint arXiv:1005.2581* (2010).

[Kes19] John Kessenich. "The OpenGL shading language version 1.10. 59". In: (May 2019). URL: http://www.opengl.org/documentation/oglsl.html.

[Knu14] Donald E Knuth. *Art of computer programming, volume 2: Seminumerical algorithms*. Addison-Wesley Professional, 2014.

[Kom+10] Kazuhiko Komatsu et al. "Evaluating performance and portability of OpenCL programs". In: *The fifth international workshop on automatic performance tuning*. Vol. 66. 2010, p. 1.

[Mal] *ARM Mali G72 Specifications*. May 2019. URL: https://developer.arm.com/ip-products/graphics-and-multimedia/mali-gpus/mali-g72-gpu.

[Mar+02] Deborah T Marr et al. "Hyper-Threading Technology Architecture and Microarchitecture." In: *Intel Technology Journal* 6.1 (2002).

[MC] Eric M Mahé and Jean-Marie Chauvet. "Fast GPGPU-Based Elliptic Curve Scalar Multiplication". In: ().

[McC10] Chris McClanahan. "History and evolution of gpu architecture". In: *A Survey Paper* (2010), p. 9.

[Mon] *Monero: Home*. May 2019. URL: https://www.getmonero.org/.

[Moo+65] Gordon E Moore et al. *Cramming more components onto integrated circuits*. 1965.

[Mun09] Aaftab Munshi. "The OpencCL specification". In: *2009 IEEE Hot Chips 21 Symposium (HCS)*. IEEE. 2009, pp. 1–314.

[Nak+08] Satoshi Nakamoto et al. "Bitcoin: A peer-to-peer electronic cash system". In: (2008).

[Nvia] *CUDA Toolkit Documentation*. May 2019. URL: https://docs.nvidia.com/cuda/profiler-users-guide/index.html#import-csv-session.

[Nvib] "GPU Pro Tip: Fast Dynamic Indexing of Private Arrays in CUDA". In: (May 2019). URL: https://devblogs.nvidia.com/fast-dynamic-indexing-private-arrays-cuda/.

[Nvic] *NVIDIA 1060 Specifications*. May 2019. URL: https://www.nvidia.com/en-us/geforce/products/10series/geforce-gtx-1060/.

[Nvid] *NVIDIA Maximize Instruction Throughput*. May 2019. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#maximize-instruction-throughput.

[Nvie] *NVIDIA Pascal Architecture Instruction Set*. May 2019. URL: https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html#maxwell-pascal.

[Nvif] "Release 378 Graphics Drivers for Windows, Version 378.66". In: (Feb. 2017). URL: http://us.download.nvidia.com/Windows/378.66/378.66-win10-win8-win7-desktop-release-notes.pdf.

[Nvi07] CUDA Nvidia. "NVIDIA CUDA programming guide (version 1.0)". In: *NVIDIA: Santa Clara, CA* (2007).

[Par+13] Bryan Parno et al. "Pinocchio: Nearly practical verifiable computation". In: *2013 IEEE Symposium on Security and Privacy*. IEEE. 2013, pp. 238–252.

[Pip76] Nicholas Pippenger. "On the evaluation of powers and related problems". In: *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*. IEEE. 1976, pp. 258–263.

[Qua+05] Gang Quan et al. "High-level synthesis for large bit-width multipliers on FPGAs: a case study". In: *Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. ACM. 2005, pp. 213–218.

[Rus] *Rust Programming Language*. May 2019. URL: https://www.rust-lang.org/.

[SA04] Mark Segal and Kurt Akeley. "The OpenGL® Graphics System: A Specification (Version 2.0 - October 22, 2004)". In: *The Khronos Group Inc* (2004).

[Sas+14] Eli Ben Sasson et al. "Zerocash: Decentralized anonymous payments from bitcoin". In: *2014 IEEE Symposium on Security and Privacy*. IEEE. 2014, pp. 459–474.

[SD16] Tyler Sorensen and Alastair F Donaldson. "The Hitchhiker's Guide to Cross-Platform OpenCL Application Development". In: *Proceedings of the 4th International Workshop on OpenCL*. ACM. 2016, p. 2.

[SGS10] John E Stone, David Gohara, and Guochun Shi. "OpenCL: A parallel programming standard for heterogeneous computing systems". In: *Computing in science & engineering* 12.3 (2010), p. 66.

[Wu+18] Howard Wu et al. "DIZK: A Distributed Zero-Knowledge Proof System". In: (2018).

[Yao76] Andrew Chi-Chih Yao. "On the evaluation of powers". In: *SIAM Journal on computing* 5.1 (1976), pp. 100–103.

[Zcaa] *Zcash BLS12-381*. May 2019. URL: https://z.cash/blog/new-snark-curve/.

[Zcab] *Zcash JubJub*. May 2019. URL: https://z.cash/technology/jubjub/.

[Zcac] *Zcash Main Page*. May 2019. URL: https://z.cash/.

[Zcad] *Zcash Parameter Generation*. May 2019. URL: https://z.cash/blog/bellman-zksnarks-in-rust/.

[Zcae] *Zcash Parameter Generation*. May 2019. URL: https://z.cash/technology/paramgen/.

[Zcaf] *Zcash Reducing Shielded Proving Time in Sapling*. May 2019. URL: https://z.cash/blog/reducing-shielded-proving-time-in-sapling/.

[Zcag] *Zcash Sapling Update*. May 2019. URL: https://z.cash/upgrade/sapling/.

[Zcah] *Zcash zk-SNARKs*. May 2019. URL: https://z.cash/technology/zksnarks/.