ZKBoo on the GPU: Better soundness errors for a little extra

(ZKBoo på GPU'en: Bedre sikkerhed for lidt ekstra)

Nina Andrup Pedersen, 202006703
Jakob Schneider Villumsen, 202004222

Bachelor Report (15 ECTS) in Computer Science

Advisor: Jesper Buus Nielsen
Department of Computer Science, Aarhus University

July 2023

AARHUS UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

We present ZKG, a parallel implementation of ZKBoo's SHA-256 $(2,3)$-decomposition on the GPU using the Futhark Programming Language. ZKBoo builds on the *MPC-in-the-head* paradigm from Ishai et al. (IKOS construction) to produce Zero-Knowledge proofs from MPC protocols.

Although ZKG is generally slower in reaching soundness error 2^{-80} than ZKBoo on the CPU, we achieve soundness error 2^{-584} significantly faster on the GPU. Experimental results show that ZKG is up to 40 times faster in achieving soundness error 2^{-584}, hence more secure against quantum computers.

Nina Andrup Pedersen and Jakob Schneider Villumsen,
Aarhus, July 2023.

Contents

Abstract **ii**

1 Introduction **1**

2 Preliminaries **2**
2.1 Multi-Party Computation 2
2.1.1 Ways to Cheat 2
2.1.2 Properties 2
2.1.3 Secret Sharing 3
2.2 Zero-Knowledge 3
2.2.1 Properties for a Zero-Knowledge Proof 4
2.2.2 Σ-Protocols 5
2.2.3 Proof of Knowledge 6
2.2.4 Commitment Schemes 6

3 MPC-in-the-head **8**
3.1 IKOS construction 8
3.1.1 Preliminaries 8
3.1.2 Protocol 8
3.1.3 MPC to Zero-Knowledge Properties 9
3.2 ZKBoo 10
3.2.1 (2, 3)-function decomposition 10
3.2.2 The protocol 11

4 ZKG **13**
4.1 Implementation 13
4.1.1 Approach 13
4.1.2 (2, 3)-Function Decomposition for SHA-256 14
4.1.3 Properties of the Function Decompositions 16
4.2 Benchmarks 17
4.2.1 Overview 17
4.2.2 Differences from ZKBoo 18
4.2.3 Experimental Setup 18
4.2.4 Experimental Results 19
4.3 Future Work 23

5 Conclusion 25

Acknowledgments 26

Bibliography 27

A Benchmark data 30

A.1 ZKBoo results 30
A.2 ZKG results 30
A.3 SHA-256 results 31

Chapter 1

Introduction

Zero-Knowledge proofs (ZK-proofs) allow parties to agree on the truthfulness of a statement without revealing the fact that makes the statement true. This paradigm has the potential to tackle many issues in the digital realm. For example, a ZK-proof allows one party to convince another that they know a password without ever revealing it.

Independent of ZK-proofs, Multi-Party Computation (MPC) protocols allow mutually distrustful parties to correctly compute a function despite errors or deliberate manipulation without revealing the individual inputs. Thus, MPC allows for privacy-preserving general-purpose computation.

However, *Ishai et al.* found in 2007 that any perfectly correct and t-private MPC protocol was reducible to a ZK-proof, assuming only the existence of one-way functions [1]. This newfound connection was later named *MPC-in-the-head*. Specifically, given an MPC protocol that computes a function f, and a public value y, a prover can prove that they know a secret w where $y = f(w)$.

Building upon this discovery, the ZKBoo protocol, published in 2016, introduced the $(2,3)$-decomposition, which allows for the transformation of any function into an MPC protocol, which, in turn, can be used for a ZK-proof [2]. Contrary to other Zero-Knowledge Proof systems, which allow for quick verification at the expense of longer proof times [3, 4], ZKBoo achieved low running times for both the prover and verifier, albeit at a higher offset. However, they must repeat the protocol 137 times to achieve sufficient soundness error.

In this report, we aim to improve upon the running times of ZKBoo, by spreading protocol repetitions onto parallel tasks on the GPU. GPUs excel at data-parallel computation tasks, where the goal is to apply the same function to lots of data. We aim to replicate the experiments of ZKBoo as closely as possible such that we can easily compare our results with theirs.

First, we in explain the theoretical aspects of MPC and Zero-Knowledge proofs in Chapter 2, after which we introduce the IKOS construction [1] and the ZKBoo protocol in Chapter 3. We then finally introduce our experiments and findings in Chapter 4.

Chapter 2

Preliminaries

2.1 Multi-Party Computation

Multi-Party Computation (MPC) is a technique used in cryptography where multiple parties collectively compute a function on their secret inputs. One key advantage of this technique is that the involved parties never need to trust a single intermediary. Additionally, an MPC protocol is resistant to cheating parties. This chapter will introduce ways to cheat, the properties of an MPC protocol, and techniques for sharing a secret.

2.1.1 Ways to Cheat

In an MPC protocol, a party is allowed to cheat in several different ways, and it is the job of the protocol designer to create a protocol that can tolerate some cheating parties. When a party cheats, we call them corrupted. A corrupted party can cheat by giving incorrect inputs or simply not following the protocol. Why a party would want to do that may seem counterintuitive. However, it may be the case that giving incorrect inputs reveals something about other parties' inputs. Not following the protocol could break correctness, privacy, or the protocol's progress. Finally, if multiple corrupted parties exist, we assume the same adversary controls them so that they can pool their knowledge. In the rest of the report, we only consider MPC protocols in the semi-honest model, where corrupted parties follow the protocol and provide correct inputs but can pool their knowledge. In constrast, in the malicious model, parties can deviate from the protocol [5].

2.1.2 Properties

An MPC protocol is for n parties $P_1, P_2, \ldots, P_n$. Each P_i has an input x_i. The protocol should compute the result y of some function f, such that $y = f(x_1, x_2, \ldots, x_n)$ [5].

In the semi-honest model, we want the MPC protocol to have properties of *correctness* and *privacy* [1].

Definition 2.1.1. (**Perfect Correctness**). *An MPC protocol is correct if, for any input* $(x_1, x_2, \ldots, x_n)$*, the protocol always outputs the correct value* $y = f(x_1, x_2, \ldots, x_n)$*.* [1]

Definition 2.1.2. *(t-**Privacy**). An MPC protocol is private if, for any input* $(x_1, x_2, \ldots, x_n)$*, no party learns any new knowledge other than y from the protocol.*
Particularly, assume we have a probabilistic polynomial time (PPT) Turing machine, S, called a simulator. The simulator has the same inputs as t corrupted parties who have pooled their knowledge. Then the MPC protocol is private if the resulting view of S has the same distribution as the combined views of the t corrupted parties.

2.1.3 Secret Sharing

The goal of secret sharing is to split some secret x into n shares, such that any $(t+1)$ shares can reconstruct x. Here, t is the threshold with $t < n$, and we can tolerate t corrupted parties.[2] Any number of shares t or less should not reveal any knowledge about x. We can differentiate between two cases of secret sharing: (1) $t = (n-1)$ and (2) $t < n$. [3]

For case (1), we want to share secret x with $t = (n-1)$. We will start by defining the group $\mathbb{Z}_p$, where p is a prime number. Then we choose $n-1$ random numbers $x_1, \ldots, x_{n-1} \in \mathbb{Z}_p$, and let

$$x_n = \left(x - \sum_{i=1}^{n-1} x_i \right) \mod p$$

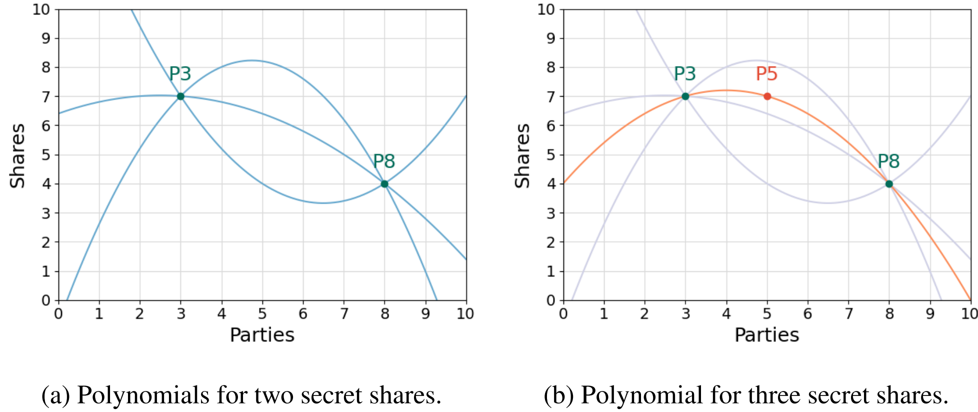
For n parties, P_i will receive the shares x_i. Then $(t+1)$ parties can reconstruct x only if $t = (n-1)$ [5].

For case (2), we want to share some secret x with $t < n$, then we can use what is known as Shamir's secret sharing scheme [6]. Here, any secret, x, is shared with a polynomial f of degree t, where $f(0) = x$. For n parties, P_i will receive the share $f(i)$. Then $(t+1)$ parties can reconstruct x, since $(t+1)$ distinct points can deterministically define a function of degree t using Lagrange interpolation [7]. Figure 2.1 shows an example of this construction.

2.2 Zero-Knowledge

A Zero-Knowledge Proof is a technique used in cryptography where one party (the prover) will try to convince another party (the verifier) that some statement is true. The Zero-Knowledge Proof should not reveal any knowledge other than the statement's validity. The technique is relevant if the prover possesses some secret knowledge, which makes the statement true, and no one else knows this knowledge. This chapter will

[1] There is also statistical and computational correctness, but we will only focus on perfect correctness.
[2] When $t = n$ all parties are corrupted, and secret sharing does not make sense.
[3] Actually, case (1) is just a special case of case (2).



(a) Polynomials for two secret shares. (b) Polynomial for three secret shares.

Figure 2.1: Shamir's secret sharing scheme for a polynomial of degree 2. Two secret shares cannot deterministically determine the polynomial. But if one more distinct point is added (the dark orange point, P_5), then the only possible polynomial is the orange one.

introduce the properties for Zero-Knowledge Proofs and relevant techniques used in these protocols, such as Σ-protocols, Proof of Knowledge and Commitment schemes.

2.2.1 Properties for a Zero-Knowledge Proof

Every Zero-Knowledge Proof must have the following three properties, *completeness*, *soundness*, and *zero knowledge* [8]. First, however, we will need formal definitions to argue for the properties.

Assume the prover, P, and the verifier, V, are both probabilistic polynomial time (PPT) Turing machines with input y. And P knows some witness w. Let R be the relation for any computational problem. Furthermore, assume a probabilistic Turing machine can decide if any pair (y,w) belongs to R in polynomial time. This assumption means R is an NP-relation. Let L be the language of yes-instances for R. Then P wants to convince V that $(y,w) \in L$ [8].

All the definitions in the following are based on [8]. These first two properties should hold for every proof system.

Definition 2.2.1. ***(Completeness)****An honest prover should be able to convince an honest verifier of a true statement. Formally, the following should hold:*

$$\forall y \in L, Pr[\langle P,V\rangle(y) = Accept] = 1$$

Where $\langle P,V\rangle(y)$ is the interaction between P and V on y.

Definition 2.2.2. ***(Soundness)*** *The verifier should not accept false statements (except with negligible probability). Formally, the following should hold:*

$$\forall y' \notin L, Pr\left[V(y') = Accept\right] \leq \varepsilon$$

Where ε is called the soundness error.

Specifically for Zero-Knowledge Proofs, the following property should hold:

Definition 2.2.3. ***(Zero-Knowledge)*** *The verifier learns nothing beyond the truth value of the given statement. Formally, the following should hold:*
Assume V^* is a corrupted verifier, and we have a PPT algorithm M^* (called a simulator*). Then for every y where $\exists w$ such that $(y,w) \in L$, it should hold that the following two results have the same distribution:*

1. *The actual interaction between P and V^* on y.*

2. *The simulated interaction by M^*, with access to V^* and y.*

V^* does not gain knowledge from P because M^* can simulate the interaction without access to P and their knowledge, w [8].

Using the soundness property specified in definition 2.2.2 only makes sense when we want to prove the membership for $y \in L$, but sometimes this property is trivial. For example, consider a verifier who knows the hash value, y, such that $y = H(w')$ (for some hash function H). Then the proof that $y \in L$ is trivial, and the prover should instead convince the verifier that they know w'. Then we need to define Σ-protocols, which guarantee Proof of Knowledge.

2.2.2 Σ-Protocols

A Σ-Protocol is a form of Zero-Knowledge proof, and in this section we give the definition and the essential properties (without proof). Here, we use Damgård's definition [9].

Definition 2.2.4. ***(Σ-Protocol)*** *A protocol $\mathscr{P}$ is a Σ-protocol for binary relation R if the following properties hold:*

1. *The communication pattern looks as follows*

   (a) *Prover P sends a message a.*

   (b) *Verifier V sends a random t-bit string e*

   (c) *P replies with z, and V decides to accept or reject based on y, a, e, z.*

2. *$\mathscr{P}$ has completeness (same definition as 2.2.1).*

3. *s-*Special soundness*: From any y and any set of s accepting conversations on input y:*

$$\{(a, e_i, z_i)\} \quad i \in \{1, \ldots, s\}$$

   where $e_i \neq e_j$ $\forall i, j$ when $i \neq j$.
   Then the witness w is effeciently computable such that $(y,w) \in R$.

4. Special honest-verifier Zero-Knowledge*: There exists a polynomial-time simulator M which, given y and e, outputs accepting conversations of the form (a, e, z) with the same probability distribution as real conversations between honest parties P, V with input y.*

Finally, Σ-Protocols have two valuable properties. First, they are invariant under parallel composition, and secondly, any Σ-protocol is also a Proof of Knowledge [9]. Note that it is not always the case that soundness is invariant under parallel composition [8].

2.2.3 Proof of Knowledge

Proof of Knowledge captures the setting where the prover can prove the claim that they know some witness w. For instance, for a hash function, a prover must prove that they know the input that maps to the hash value. We will use Damgård's notation [9] based on Bellare and Goldreich's definition [10]. Let κ be the *knowledge error* which denotes the probability that the verifier accepts a proof without a correct witness w. Then a protocol $\langle P, V \rangle$ is a Proof of Knowledge for the relation R if the following two conditions hold:

Definition 2.2.5. ***(Completeness)*** *On common input y, if the prover is honest and receives an extra input w such that $(y, w) \in R$, then the verifier always accepts* [4].

Definition 2.2.6. ***(Knowledge Soundness)*** *Assume there exists a PPT Turing Machine called M. It receives as input y and black-box rewindable access to the prover and tries to compute w, where $(y, w) \in R$. For any prover P^*, let $p(y)$ be the probability that the verifier V accepts on input y. Then there exists constant c such that when $p(y) > \kappa(y)$, then M will output a correct w in at most*

$$\frac{|y|^c}{p(y) - \kappa(y)}$$

steps, where access to P^* counts as one step.

Knowledge soundness states that when the prover is good at convincing the verifier, M outputs a correct w in few steps. When p and κ are close to each other, the expression increases in value, and the further apart they are, the closer it gets to $|y|^c$.

2.2.4 Commitment Schemes

A commitment scheme is a two-phased protocol. The sender will commit to some value in the first phase (the commit phase) and reveal the value in the second phase (the reveal phase). The commitment should be unforgeable by ensuring the sender cannot change the committed value later. In addition, the receiver should be able to validate that the revealed value is the same as the committed value [8].

A practical method for committing a value is to apply a cryptographic hash function to the given value. The receiver can easily verify a hash value. Moreover, the sender

[4] 2.2.5 is similar to 2.2.1

cannot modify the committed value because cryptographic hash functions are collision-resistant.

Zero-Knowledge proofs frequently incorporate a commitment scheme. For instance, in a Σ-protocol, the verifier's challenge could be asking about the value of a commitment, as we will see later.

Chapter 3

MPC-in-the-head

3.1 IKOS construction

Combining Zero-Knowledge Proofs and Secure Multi-Party Computation, we get what is known as MPC-in-the-head, first described by [1]. The general idea is to generate a Zero-Knowledge proof, by simulating an n-party MPC protocol "in the head". Notably, the Zero-Knowledge Proof properties follow from the MPC properties.

3.1.1 Preliminaries

The following is a simplified version of the notation and explanations in [1].

Assume we have an NP-relation R, the language L of yes-instances in R, and a function f corresponding to the relation R. Also, assume we can transform the function f into an n-party MPC protocol Π_f with perfect correctness and t-privacy (in the semi-honest model). As described in section 2.2.1, both the prover and verifier know some $y \in L$, and the prover knows a w such that $(y, w) \in L$.

Whenever we execute the protocol, each party will store their computed values in a so-called view. For example, for party P_i, their view, V_i, will store their input share, w_i, the randomness used throughout the execution, r_i, and the messages they receive from other parties. The purpose of these views is for the verifier to check and recompute the execution of the protocol. The verifier will examine two views, V_i and V_j, by checking that the incoming messages at V_j are the same as the outgoing messages from V_i. If this holds, then the views are *consistent*.

3.1.2 Protocol

The following describes the steps in the protocol from [1]. This version is in the semi-honest model and assumes perfect correctness and two-privacy for the MPC protocol Π_f. Additionally, we have a commitment scheme as defined in section 2.2.4.

1. **In-the-head execution**

(a) The prover secret shares the witness w such that $w = w_1 \oplus w_2 \oplus \ldots \oplus w_n$[1].

(b) The prover executes the MPC-protocol Π_f on the inputs $w_1, w_2, \ldots, w_n$ "in-the-head" by simulating each party.

(c) From the execution, the prover has obtained the views $V_1, V_2, \ldots, V_n$ which are transcripts of the MPC protocol as seen by party $P_1, P_2, \ldots, P_n$. The prover commits to each of these views and the commitments are sent to the verifier.

2. **Challenge**

(a) The verifier picks two distinct numbers $i, j \in \{1, 2, \ldots, n\}$.

(b) The verifier challenges the prover to open the corresponding two views.

3. **Response**

(a) The prover opens the two views V_i and V_j and sends them to the verifier.

4. **Verification**

(a) The verifier checks that the opened views are consistent with each other, the committed views, output shares, and the public value y.

(b) If the views are consistent, the verifier accepts. Otherwise, reject.

3.1.3 MPC to Zero-Knowledge Properties

The Zero-Knowledge proof should have the properties introduced in section 2.2.1. This section will argue how to get the ZK-proof properties from the MPC properties (*perfect correnctness* and *t-privacy*). All the argumentations are based on the work of [1].

Completeness

This should only hold when both the prover and verifier are honest. Assuming honest prover and verifier, we get completenes from perfect correctness of the MPC protocol. An honest prover will always output the correct value, y, given the inputs, $x_1, \ldots, x_n$, and all views will be consistent, which makes the verifier accept.

Soundness

Assume that $\forall w, (y, w) \notin R$ [2]. The soundness depends on whether the prover executes the protocol honestly or not. If the prover honestly executes the protocol, the result of the protocol is y'. Since the MPC protocol is correct, we get $y' \neq y$, and the verifier will always reject. On the other hand, if the prover does not execute the protocol honestly, then at least two views are inconsistent. In this case, the verifier will reject with the same probability that it opens the two inconsistent views.

[1] This secret sharing technique is just a special case of addition which is introduced in section 2.1.3

[2] This describes Proof of Membership, which is sometimes trivial. See discussion in 2.2.1.

Zero-Knowledge

The proof will have property Zero-Knowledge when the MPC protocol is t-private. Assume we have a PPT simulator, M^*, which has access to y, a corrupted verifier, V^*, and an MPC simulator, S. [3] The prover chooses the inputs, $w_1, \ldots, w_n$, to be uniform and independent. If M^* chooses t random values, $w_1^*, \ldots, w_t^*$, these will have the same distribution as the provers inputs. Then let M^* obtain t views, $V_1, \ldots, V_t$, by running S on $(w_1^*, \ldots, w_t^*)$. It follows from the MPC property of t-privacy that these obtained views have the same distribution as those from P in the interaction with V^*. Correspondingly, the interaction between P and V^* has the same distribution as when M^* simulates S and V^*. This equivalence is precisely the definition of Zero-Knowledge.

3.2 ZKBoo

The ZKBoo protocol [2] is a practical instance of the IKOS construction [1], which works for statements on the form:

$$\text{"I know } w \text{ such that } y = f(w)\text{"}. \tag{3.1}$$

Additionally, ZKBoo includes an implementation of the protocol with SHA-1 [11] and SHA-256 [11], which demonstrates the feasibility of a real library implementation.

3.2.1 (2, 3)-function decomposition

The original IKOS construction treats the MPC functionality as a black box. ZK-Boo introduces an MPC circuit construction for three parties[4] with correctness and 2-privacy (in the semi-honest model) called $(2,3)$-function decomposition. This decomposition works for any function f on the form 3.1. Thus, the $(2,3)$-decomposition gives the flexibility to decompose any function to an MPC protocol with the desired properties, and the IKOS construction ensures that it can be reduced to a valid ZK-proof.

The method allows each party P_i to receive intermediate computations from P_{i+1}, where $i+1 \equiv (i \mod 3)+1$. Whenever a party receives an intermediate computation, they store it in their view. In addition to the computations, each party also stores their secret share w_i and their output share y_i in their view. The views allow the verifier to recompute and verify the computations for party P_i but keep the view of P_{i+2} secret (since the recomputation of one view only requires the prover to reveal two views).

Specifically, a $(2,3)$-function decomposition, D, includes several components:

- `Share`: the method for secret sharing the input, w (like the ones in section 2.1.3).
- $\texttt{Output}_i$: computes the output share y_i from view V_i and random tape k_i.

[3] The MPC simulator is defined in definition 2.1.2

[4] There are some performance considerations underpinning the choice of 3 parties, but we refer the curious reader to [2].

- `Reconstruct`: the method for reconstructing the output, y, from the outputs shares, y_1, y_2, and y_3.
- Π_f: the MPC protocol for three parties with 2-privacy corresponding to f.

The definitions for the decomposition's MPC properties (correctness and privacy) are similar to the formal definitions in section 2.1.2.

3.2.2 The protocol

The prover knows w, such that $y = f(w)$. The prover and verifier know y and D (a $(2,3)$-decomposition of f). The protocol consists of three phases, the commitment phase, the proving phase, and the verification phase [2]. These phases realize the communication pattern for a Σ-protocol.

Commitment phase

1. The prover secret shares the input w with `Share` and generates three random tapes, k_1, k_2, and k_3, one for each party.[5]
2. Then the prover runs Π_f in its head with the secret shares as input. The result is three views, V_1, V_2, and V_3.
3. The prover commits to the view at each party, $c_i = \texttt{Commit}(k_i, V_i)$.
4. Lastly, the prover sends the commitments to the verifier along with the output shares, (y_1, y_2, y_3), where $y_i = \texttt{Output}_i(k_i, V_i)$.

Proving phase

1. The verifier picks some number $e \in \{1, 2, 3\}$ and sends it to the prover.
2. The prover opens the commitments with index e and $e+1$[6] and sends the random tapes, k_e, and k_{e+1}, and views, V_e, and V_{e+1}, to the verifier.

Verification phase

1. The verifier checks that the information from the prover is consistent. This can be done by answering the following questions:
   (a) Is the reconstruction of the output shares the same as the public input, y?
   $\texttt{Reconstruct}(y_1, y_2, y_3) \stackrel{?}{=} y$
   (b) Are all the output shares the same as the last value of the corresponding opened views?
   $y_i \stackrel{?}{=} \texttt{Output}_i(k_i, V_i), \quad \forall i \in \{e, e+1\}$
   (c) Are all the committed views the same as commiting the opened views? [7]
   $\texttt{Commit}(k_i, V_i) \stackrel{?}{=} c_i, \quad \forall i \in \{e, e+1\}$

[5]These tapes provide the randomness to D.

[6]Where $e + 1 \equiv (e \mod 3) + 1$

[7]This step is not in the original ZKBoo paper [2], but they did include it in the paper for ZKB++ [12]. If the verifier omitted this check, the prover could cheat with the commitments.

(d) Are all the values in the opened view, V_e, the same as recomputing Π_f for party e?

2. If the verifier can answer "yes" to all the questions, the verifier accepts; otherwise, the verifier rejects.

The properties for the Zero-Knowledge proof follow from the fact that the protocol is a Σ-protocol and the IKOS construction. We assume the MPC protocol is perfectly correct and perfectly 2-private in the semi-honest model. The IKOS construction proves these MPC properties give Completeness and Zero-Knowledge [1]. The Σ-protocol gives us 3-Special Soundness and Special honest-verifier Zero-Knowledge.

However, we must argue 3-Special soundness. Assume we have three accepting conversations, $\{(a,e_1,z_1),(a,e_2,z_2),(a,e_3,z_3)\}$. If $e_1 \neq e_2$, $e_1 \neq e_3$, and $e_2 \neq e_3$, we must have overlapping views (which are equal) in all conversations because the commitment function is assumed to be collision resistant. Then we know that there must exist some simulator W, which from the opened views, can backtrack the protocol and compute w' such that $y = f(w')$ [2].

Finally, the Σ-protocol is a Proof of Knowledge [9], which means the prover must know w such that $y = f(w)$. Additionally, because it is a Σ-protocol, running several instances in parallel does not reduce soundness [2] [9].

Remark: As demonstrated by the *Picnic* signature scheme based on ZKBoo [12], the primitives in MPC-in-the-head do not have to rely on asymmetric cryptography– easily broken in a post-quantum world [13]. Instead, the signature scheme can use symmetric-key primitives, which are harder to break. This fact is one of the reasons why MPC-in-the-head is useful.

Chapter 4

ZKG

The ZKBoo protocol, introduced in Chapter 3, enables efficient Zero-Knowledge proofs using techniques from Multi-Party Computation. However, we must run the ZKBoo protocol repeatedly to achieve a sufficient soundness error. This chapter presents our contribution, where we aim to improve the speed by implementing ZKBoo on GPUs. A GPU can perform many more parallel computations simultaneously than a CPU. Furthermore, by utilizing the Futhark Programming Language [14], intended for parallel programming, which compiles to GPU code, we aim to gain some additional performance from the Futhark compiler, as opposed implementing it directly in an OpenCL C dialect [15].

As a proof-of-concept, we have implemented the SHA-256 hash function [11] within the ZKBoo framework, showcasing the potential advantages of GPU-based Zero-Knowledge proofs.

Lastly, we have named our project ZKG–an abbreviation of ZKBoo on the GPU.[1] Our implementation is available at `https://gitlab.au.dk/jaschdoc/zkg`.

4.1 Implementation

This section briefly argues why we implemented ZKBoo, the way we did it, and what trade-offs we considered. We then prove that the SHA-256 $(2,3)$-decomposition has privacy and correctness, thus yielding the required properties for Zero-Knowledge proofs via the IKOS construction.[2]

4.1.1 Approach

Our primary hypothesis was that we could improve the ZKBoo protocol's performance by splitting the 137 instances into parallel jobs on the GPU instead of being bound by

[1] ZKGoo had an odd ring to it.

[2] Our proof is naturally valid because the $(2,3)$-decomposition of ZKBoo is valid for all functions, but our proof may be easier to understand in the context of this report.

the maximum number of threads available on a CPU. In order to make it easy for us, we opted to implement it in the Futhark Programming Language.

Futhark is a functional and pure language, and because of its purity, the compiler can be aggressive in its optimizations. Furthermore, since Futhark is syntactically similar to the ML family of languages, it has high-level abstractions that allow the programmer to express complicated concepts while emitting efficient GPU code. The language also has several desirable features. Firstly, it can compile a program into several different backends: sequential C, multi-threaded C, CUDA[3], and OpenCL[4]—the latter two run on the GPU. The different backends ensured the possibility of comparing the ZKBoo implementation with ours. If we had chosen to implement it directly in CUDA or OpenCL, we would not have easily been able to compare our CPU performance with the ZKBoo implementation. Secondly, Futhark has integrated benchmarking features with the possibility of automatically estimating parameters for when to parallelize tasks through the autotune feature [16]. Moreover, since autotune is a dynamic optimization technique, it compliments its static optimization, yielding even more performance.

Initially, we opted to write the control code in the Go programming language.[5] However, after observing a considerable slow-down compared to the ZKBoo C implementation, we wrote the control code in C instead. The Go runtime, especially the garbage collector, likely incurred this overhead. Additionally, since we were trying to compare our results with the original ZKBoo results, we needed to reduce the changes we made from the original implementation. Thus, after seeing the poor performance of Go, it was an easy choice to rewrite it in C. Luckily, we had written a test suite for our Go implementation, so it was easy to port things over to C and remain confident that it would work.

4.1.2 (2, 3)-Function Decomposition for SHA-256

[2] introduces the technique $(2,3)$-function decomposition to construct an MPC protocol from any function. Since our implementation uses the function SHA-256, we will demonstrate the course of action in this section. The basic arithmetic operations in SHA-256 are bitwise XOR, bitwise AND, and integer addition [11]; we must construct these as three-party MPC operations.

Lemma 4.1.1. *Let a and b be two bits, which are secret shared as $a = a_1 \oplus a_2 \oplus a_3$ and $b = b_1 \oplus b_2 \oplus b_3$. And let a_i and b_i be the shares of a and b held by party P_i. Then XOR and AND are defined as:*

$$(a \oplus b)_i = a_i \oplus b_i \tag{4.1}$$

$$(a \wedge b)_i = (a_i \wedge b_i) \oplus (a_j \wedge b_i) \oplus (a_i \wedge b_j) \oplus r_i \oplus r_j \tag{4.2}$$

Where $j = (i \mod 3) + 1$. And r_i is a random bit for party P_i.

We will do integer addition as specified in [17]. Figure 4.1 illustrates the bristol circuit for integer addition, which only uses XOR and AND.

[3] `https://docs.nvidia.com/cuda/`

[4] `https://www.khronos.org/opencl/`

[5] `https://go.dev/`

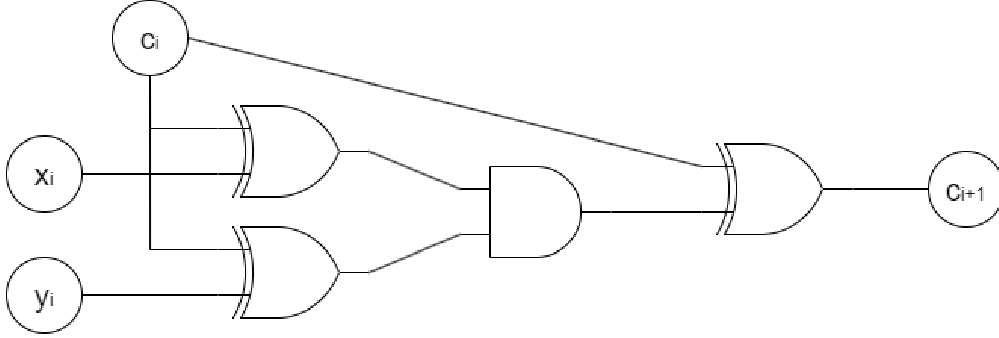


Figure 4.1: A bristol sub-circuit [17] for integer addition. The inputs x_i and y_i are two bits from the input integers x and y; c_i is the carry bit from the previous sub-circuit; and c_{i+1} is the carry bit for the next sub-circuit.

Theorem 4.1.1. *If the prover did not use randomness to calculate the AND (eq 4.2), then from the opened commitments, there is a 50% chance that an adversary can guess the inputs to AND.*

Proof. Let us assume the prover did not use randomness to calculate AND, and the opened views are from party P_1 and party P_2. Then the opened views will reveal the following knowledge:

$$a_1, b_1, a_2, b_2, \tag{4.3}$$

$$w_1 = (a_1 \wedge b_1) \oplus (a_2 \wedge b_1) \oplus (a_1 \wedge b_2), \text{ and} \tag{4.4}$$

$$w_2 = (a_2 \wedge b_2) \oplus (a_3 \wedge b_2) \oplus (a_2 \wedge b_3) \tag{4.5}$$

The calculation 4.4 does not include any new knowledge not known by P_1 and P_2. But the calculation 4.5 includes new knowledge a_3 and b_3. Let's rewrite this equation:

$$w_2 \oplus (a_2 \wedge b_2) = (a_3 \wedge b_2) \oplus (a_2 \wedge b_3) \tag{4.6}$$

Then we have four cases for leaking knowledge, depending on the values of a_2 and b_2.

a_2	b_2	Eq 4.6 result	Revealed knowledge
0	0	0	-
0	1	a_3	$a = a_1 \oplus a_2 \oplus a_3$.
1	0	b_3	$b = b_1 \oplus b_2 \oplus b_3$.
1	1	$a_3 \oplus b_3$	-

Note: '-' means nothing is revealed.

From these cases, there is a 50% chance that an adversary can get hold of either a or b. And this breaks the property of *privacy* (definition 2.1.2), which means we must use randomness for AND. ■

4.1.3 Properties of the Function Decompositions

Theorem 4.1.2. *The function decompositions of XOR and AND have the property* Perfect Correctness *(as defined in 2.1.1).*

Proof. If the arbitrary operation is $\boxplus$, then the following should hold:

$$a \boxplus b = (a \boxplus b)_1 \oplus (a \boxplus b)_2 \oplus (a \boxplus b)_3$$

for the operation to be correct.

For XOR, we get:

$$\begin{aligned}
&(a \oplus b)_1 \oplus (a \oplus b)_2 \oplus (a \oplus b)_3 \\
&= (a_1 \oplus b_1) \oplus (a_2 \oplus b_2) \oplus (a_3 \oplus b_3) \\
&= (a_1 \oplus a_2 \oplus a_3) \oplus (b_1 \oplus b_2 \oplus b_3) \\
&= a \oplus b
\end{aligned}$$

For AND, we get:

$$\begin{aligned}
&(a \wedge b)_1 \oplus (a \wedge b)_2 \oplus (a \wedge b)_3 \\
&= (a_1 \wedge b_1) \oplus (a_2 \wedge b_1) \oplus (a_1 \wedge b_2) \oplus r_1 \oplus r_2 \oplus \\
&\quad (a_2 \wedge b_2) \oplus (a_3 \wedge b_2) \oplus (a_2 \wedge b_3) \oplus r_2 \oplus r_3 \oplus \\
&\quad (a_3 \wedge b_3) \oplus (a_1 \wedge b_3) \oplus (a_3 \wedge b_1) \oplus r_3 \oplus r_1 \\
&= (a_1 \wedge b_1) \oplus (a_2 \wedge b_1) \oplus (a_1 \wedge b_2) \oplus \\
&\quad (a_2 \wedge b_2) \oplus (a_3 \wedge b_2) \oplus (a_2 \wedge b_3) \oplus \\
&\quad (a_3 \wedge b_3) \oplus (a_1 \wedge b_3) \oplus (a_3 \wedge b_1) \\
&= (a_1 \wedge (b_1 \oplus b_2 \oplus b_3)) \oplus (a_2 \wedge (b_1 \oplus b_2 \oplus b_3)) \oplus (a_3 \wedge (b_1 \oplus b_2 \oplus b_3)) \\
&= (a_1 \wedge b) \oplus (a_2 \wedge b) \oplus (a_3 \wedge b) \\
&= (a_1 \oplus a_2 \oplus a_3) \wedge b \\
&= a \wedge b
\end{aligned}$$

Both the equations for XOR and AND hold, which means they have the property *Perfect Correctness* (as defined in 2.1.1). ■

Theorem 4.1.3. *The function decomposition of XOR and AND (from lemma 4.1.1) have the property* 2-Privacy *(as defined in 2.1.2).*

Proof. For the property of 2-privacy for a $(2,3)$-function decomposition, it should hold that the knowledge received by two parties in the protocol should have the same distribution as if a PPT simulator, S, simulated the protocol. Formally, we can encapsulate the knowledge received by two parties as

$$\{k_e, V_e, k_{e+1}, V_{e+1}, y_{e+2}\}$$

Then the simulator can do as follows:

1. The simulator, S, knows the public input y and D. It simulates the protocol for two imaginary parties, P_i and P_j, where $j = i + 1$.

2. Let S sample two random tapes, k_i and k_j, and two input shares, w_i and w_j. This randomness will have the same distribution as the randomness in the actual protocol.

3. Then S can simulate the functionality of D, where it calculates the operations as defined in lemma 4.1.1.

   (a) The XOR gate will always have the same distribution as in the actual protocol.

   (b) For an AND operation in P_j, S should generate random values for a_k, b_k, and r_k, where $k = j + 1$. Since an AND operation uses a randomly sampled r in the actual protocol, the result distributes uniformly like the result for S.

4. Lastly, S can use the output shares from D, y_i and y_j, to calculate $y_k = y \oplus y_i \oplus y_j$. Here, y_k will have the same distribution as y_{e+2}.

This simulation gives us the 2-privacy property since the knowledge for party P_i and P_j in the simulation has the same distribution as the knowledge for party P_e and P_{e+1} in the actual MPC protocol. ■

The MPC protocol has both properties of *Perfect Correctness* and *Perfect 2-Privacy*. We know from ZKBoo, that the Zero-Knowledge proof has properties of *Soundness*, *Completeness* and *Zero-Knowledge* (see section 3.2.2).

4.2 Benchmarks

4.2.1 Overview

We split the benchmarks into two major categories:

1. End-to-end benchmarks

2. Futhark-only benchmarks

The end-to-end benchmarks are directly comparable to the original implementation of the SHA-256 decomposition demonstrated in ZKBoo. These benchmarks wrap our SHA-256 decomposition into a binary that reads input, generates randomness, and has other quality-of-life improvements, like setting the number of times to repeat the benchmark. Naturally, this is comparable to a real-world setting, where a program would call this API to generate a transferable ZK-proof and to verify with a separate API call.

The Futhark-only benchmarks do not have wrapper code. Instead, we generated inputs in advance and directly ran custom entry points in Futhark, using the built-in benchmarking feature. This approach has less overhead than our end-to-end benchmarks and allows us to examine one part of the algorithm at a time closely. This suite consists of five benchmarks:

1. Proof and verification total time
2. Proof-only time
3. SHA-256 time
4. Parallel SHA-256 Mock time
5. Proof-only time using parallel SHA-256 mock

We crafted the latter two benchmarks to investigate the slow-down that the sequential nature of SHA-256 incurred on execution times when used as commitment function and how much we could reduce the execution time of a single protocol phase. The parallel SHA-256 Mock is not a secure hash function, as we have replaced the sequential loops with data-parallel operations map/reduce using mock constants to see how much performance we could gain from removing dependencies on previous computations. We only experimented with the commitment function in this regard.

4.2.2 Differences from ZKBoo

The most significant difference is that we ran the verifier within the same program without writing to disk, so we did not have to worry about noise from I/O operations [2]. Of course, the verifier ran in a separate API call so that we could benchmark separately.

We also used a different function to time function calls, as the clock function on Linux did not measure parallel programs correctly, and we updated the original ZKBoo code, making timings comparable.

Finally, we have larger proof sizes as our prover sends the fully materialized random tapes to the verifier instead of just the keys to AES. This change is an oversight on our part, as it made it easier to test when writing the code.

4.2.3 Experimental Setup

We ran our benchmarks on a computer installed with Fedora 38, equipped with an Intel I7-4790k CPU with four cores and eight threads, clocked at 4 GHz, boosting to 4.4 GHz. For the GPU, the computer had a NVIDIA GeForce GTX 970 clocked at 1114 MHz, boosting to 1253 MHz, with 1664 CUDA cores.

The wrapper program uses OpenSSL[6] to generate randomness and a hash value of the input that the verifier can compare to the hash value in the ZK-proof. For our end-to-end benchmarks, we report on the results of running the program on four different inputs using 137 rounds. This experiment was repeated 1000 times for each input. We then separately measured the proving and verification times using the `omp_get_wtime` function in OpenMP.[7]

[6] https://www.openssl.org/

[7] https://www.openmp.org/

We ran the Futhark-only prover and parallel prover with increasing numbers of rounds. We also repeated the benchmark for the total time of proof and verification 1000 times; for the remaining benchmarks in this suite, we ran 500 times. For the SHA-256 benchmarks, we ran it on varying input sizes up to 50 MB. However, only the parallel SHA-256 mock version could handle larger input sizes. We ran the Futhark-only benchmarks with Futhark's four backends: Sequential C, Multicore C, CUDA, and OpenCL. Additionally, we ran everything for the GPU backends without using Futhark's autotune feature and after having tuned GPU parameters with autotune.

Finally, we generated an interactive proof and the indices the verifier should choose for its challenge in advance. Of course, it is not entirely correct, but the most important thing is that the prover does not get to choose the indices; thus, for benchmarking and correctness of the protocol, it does not matter if the verifier picks these numbers or receives them from an oracle.

4.2.4 Experimental Results

All the data used in this section can be found in Appendix A. All color schemes are from [18].

Total running times of all implementations

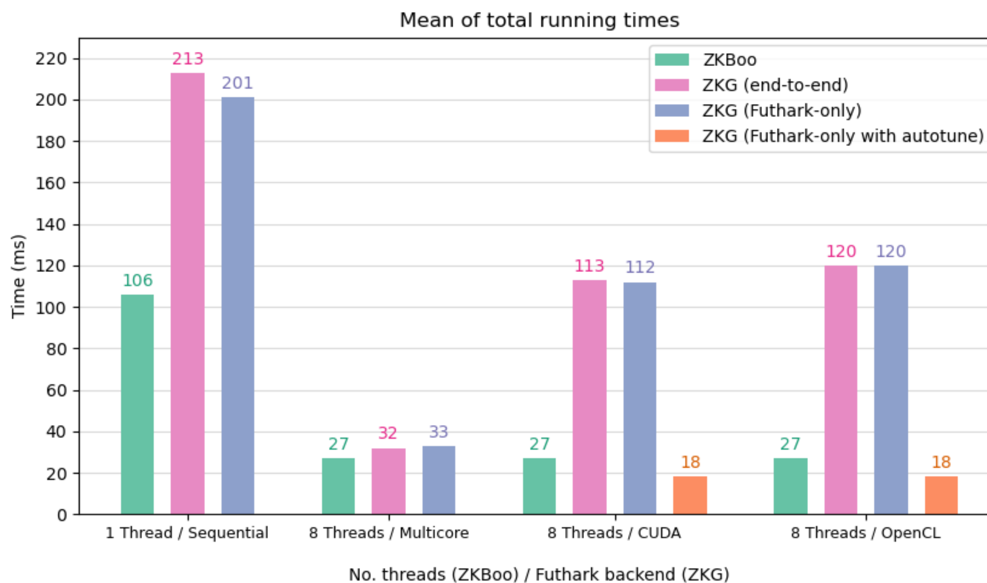


Figure 4.2: Total running times of proof and verification (not including random tape generation), average of 1000 repetitions over 4 different inputs, using 137 protocol rounds. Since the ZKBoo implementation does not compile to GPU, we compare the CPU runtime using the maximum number of threads with the ZKG running on the GPU.

Figure 4.2 shows the mean of all implementations' total running times. Here, we can compare how our implementation, ZKG, performs against the original ZKBoo implementation. The second batch of bars (multicore) shows that the parallel backend

of Futhark is clearly on par with ZKBoo, both for the end-to-end and the Futhark-only benchmarks. The pink and blue bars for the GPU backends (CUDA and OpenCL) show that the non-tuned backends are significantly slower than ZKBoo. The story changes slightly after tuning GPU parameters, as we see an increase of 9 ms over ZKBoo (the orange bar for CUDA and OpenCL). Of course, this varies from system to system as some systems may have newer GPUs than CPUs and vice versa. We leave it as future work to estimate the parameters for deciding when to run on the GPU and CPU. Nevertheless, the results indicate that despite the tension between the sequential nature of SHA-256, the ZKBoo protocol benefits from GPU parallelization in some cases.

Normal SHA-256 versus Parallel SHA-256 Mock

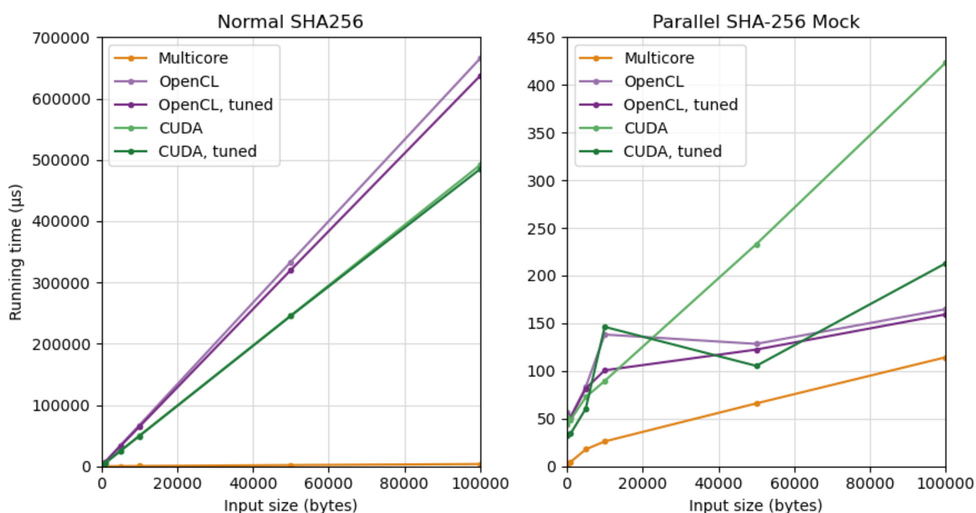


Figure 4.3: The running times of SHA-256 on the GPU and multicore on the CPU. These were Futhark-only benchmarks.

Figure 4.3 shows the running time of SHA-256 and the mock version on different backends. It is easy to see that the GPU backends suffer from the sequential loops in SHA-256, and the hash times benefit significantly from a parallel implementation. Moreover, the autotune functionality in Futhark did not improve the running time as it did for the entire protocol (see figure 4.2), which indicates that significantly increasing speed using SHA-256 is difficult. However, if we use the parallel mock version of SHA-256, the running time improves dramatically, and the autotune in Futhark makes it even faster. The hash function now only takes a few hundred microseconds, making it over 1000 times faster than the sequential implementation. As mentioned previously, the parallel mock version is not secure, but it merely demonstrates that GPUs work better with data-parallel algorithms. Despite the much faster running times of the parallel SHA-256 mock, the multicore backend still beats the GPU backends. This finding is likely because the clock rate of the CUDA cores is lower than the clock rate of the CPU cores. However, discounting the memory limit, the graphs indicate that eventually, the GPU would be just as fast as the CPU, so a natural next step would be to investigate when this happens.

Proof times, Normal SHA-256 versus Parallel SHA-256 Mock

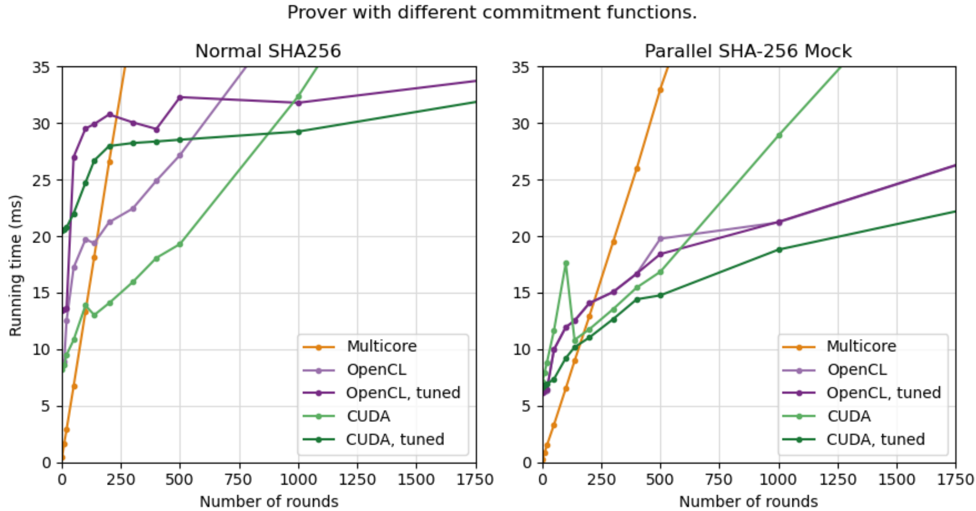


Figure 4.4: The running times of the Prover on the GPU and multicore on the CPU. These were Futhark-only benchmarks.

Figure 4.4 compares the running time of the prover when we use normal SHA-256 against the parallel mock version of SHA-256. The proof times also benefit from a parallel implementation, but less drastically than only SHA-256 (figure 4.3). However, since the GPU swiftly executes data-parallel programs, SHA-256 significantly impacts the total running time. Furthermore, figure 4.4 shows that the GPU backends are faster than the CPU multicore backend when running the prover with the parallel SHA-256 mock over 200 rounds. The figure shows that running both several hundred rounds in parallel with a data-parallel commitment scale better on the GPU than on the CPU.

Lastly, figure 4.5 indicates that the average running time per round decreases further than the multicore backend as the number of rounds grows, suggesting that the GPU backend has more headroom for many instances of the protocol. The multicore backend stabilizes on a running time per round of around 120 and 60 (SHA-256 and mock version respectively) on smaller inputs than the GPU backends. Figure 4.5 also supports the finding in figure 4.2 that the GPU backend can sometimes outperform the CPU backend. However, it is worth noticing that the parallel SHA-256 mock helps decrease the running time per round at fewer rounds faster than the normal SHA-256.

To summarize, all backends benefit from a data-parallel commitment function, and the GPU backends, in particular, see a massive increase in performance compared to the multicore backend.

Soundness Errors

Figure 4.6 shows the running time of the prover for 137 rounds (orange bar) and 1000 rounds (full bar). When we run the protocol for 137 rounds, we get a soundness error

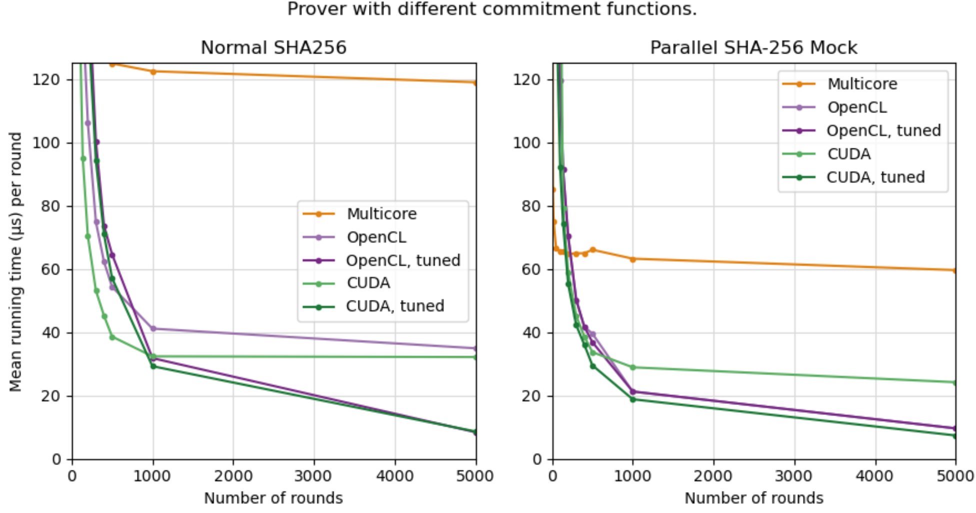


Figure 4.5: Running time per round of the Prover on the GPU and multicore CPU. These were Futhark-only benchmarks. For the parallel SHA-256 mock, we can run up to 1000 rounds for a little additional execution time.

of 2^{-80}, whereas 1000 rounds yield a soundness error of 2^{-584}.[8] The figure shows the smaller soundness error for a slight increase in execution times when using the GPU backend. The multicore backend is faster at running 137 rounds than the CUDA backend within each group. However, the CUDA backend is consistently faster on 1000 rounds.

For the normal SHA-256, the multicore CPU backend runs 137 rounds faster than the tuned CUDA backend. However, the tuned CUDA backend only takes an additional 2.6 ms to run 1000 rounds, whereas the CPU is 40.73 times slower. Furthermore, the tuned CUDA backend improves when we use the parallel SHA-256 mock compared to the normal SHA-256. However, although the multicore backend with the parallel SHA-256 mock is still slightly faster when running 137 rounds, it is much slower in reaching the second milestone. Thus, in the parallel SHA-256 mock case, the tuned CUDA backend reaches the same soundness error as the multicore CPU backend at nearly the same time and much improves the soundness error for just a few milliseconds more.

One explanation for this result could be that the GPU will run all 1000 rounds in parallel since it has more than 1000 CUDA cores. In contrast, the number of cores bounds the CPU, slowing it down even though each CPU core is faster than a CUDA core. We tested the protocol with 5000 rounds in the prover-only benchmark, this did not result in a huge increase in running time.

In a post-quantum signature scheme like *Picnic* [12], the lesser soundness errors further future proofs signatures against Grover's algorithm [19].

[8] $\sigma((\log_2 3) - 1)^{-1}$ rounds yield a soundness error of $2^{-\sigma}$ [2]. Thus, running n rounds yields a soundness error of $2^{-\sigma}$ where $\sigma = \left\lfloor \frac{n}{((\log_2 3)-1)^{-1}} \right\rfloor$.

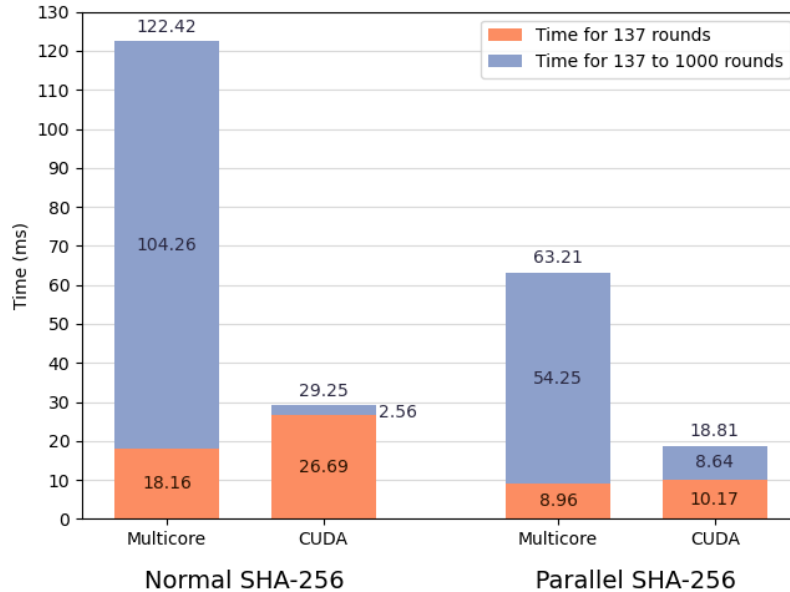


Figure 4.6: The prover running times. Note that the blue bar alone is from 137 rounds to 1000 rounds, and the time at the top of the bar is the total time. The "Parallel SHA-256" refers to the parallel mock version.

4.3 Future Work

In this section, we have collected some of our ideas for future projects building upon the findings in this report.

The proof size in our current implementation is much larger than in ZKBoo, containing indices, random tapes, and two opened views. Hence, our proof size is 1610.29 KiB. The size was not a problem as our proofs never left the GPU, so there was no extra overhead attached to this method. A natural course of action to reduce the size would be only to send the AES keys so the verifier can generate the random tapes. It would also be interesting to try the same GPU-based approach with the optimization introduced in ZKB++ [12]. An example of an optimization would be to let the prover only send one view instead of two when opening the commitment.

Our results show that the sequential nature of SHA-256 incurred a major bound on performance. Thus, a Merkle Tree-based hash function [20] would improve commitment time as we can parallelize the tree structure more easily. One way to do this would be to implement a parallel hash algorithm like `BLAKE3` [21] or `ParallelHash` from SHA-3 [22]. `BLAKE3` is particularly interesting as the backing data structure is a Merkle Tree, directly allowing us to test our hypothesis above.

We chose SHA-256 in order to reproduce and compare the results of the original ZKBoo implementation. However, it would be interesting to exploit the parallel nature of `BLAKE3` in a $(2,3)$-decomposition to see if this also yields a performance increase when computing the ZK-proof.

Of course, data-parallel hash functions and cryptographic operations are worth investigating as they would run faster than sequential algorithms on modern hardware. However, this claim is very general and applies to various computer science problems.

For example, it would be interesting to investigate whether there is a tension between parallelism and security (in this case, collision resistance and one-way property of hash functions).

As a final remark, the ZKBoo and ZKB++ function decomposition is a flexible tool for in tandem with the IKOS construction, and the SHA-256 decomposition is just an example decomposition. Furthermore, compared to other Zero-Knowledge proof protocols, ZKBoo allows online parties to communicate synchronously, which is the case for password-based sign-in. It is already a substantial improvement that ZKBoo allows a user never to reveal their password to another party, but it is much easier to find the preimage of a hash function than encryption schemes; thus, it would be interesting to create a $(2,3)$-decomposition of LowMC [23] as done in Picnic [12], on the GPU and investigate the performance of it. A server storing an encrypted password has better security guarantees than a server storing a hashed password.

Chapter 5

Conclusion

This report provided the reader with the necessary building blocks for understanding *MPC-in-the-head* [1]. We showed how $(2,3)$-decompositions from ZKBoo [2] worked as an example of the IKOS construction [1].

We also showed that implementation of the $(2,3)$-decomposition of SHA-256 on the GPU yielded significantly lesser soundness error, 2^{-584}, for little extra running time, despite being slower at reaching soundness error 2^{-80} which [2] used as goal. The lesser soundness error provides better future proofing for signature schemes against quantum algorithms such as Grover's algorithm [19].

Additionally, our findings indicate that the sequential nature of SHA-256 bottlenecked performance. Thus, using a less sequential function for commitments would significantly increase performance. We also hypothesize that a less sequential function in the $(2,3)$-decomposition would further increase performance, for instance, with `BLAKE3` [21]. However, we leave this as future work, alongside implementing the optimizations from ZKB++ [12] and implementing a $(2,3)$-decomposition of LowMC [23].

Acknowledgments

We want to thank our supervisor Jesper Buus Nielsen for his guidance and feedback and Claudio Orlandi for clarifying our questions regarding ZKBoo.

Finally, we also want to thank Troels Henriksen, author of the Futhark Programming Language, for his swift and helpful answers on the Futhark Gitter channel whenever we encountered Futhark-related issues.

Bibliography

[1] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In *Proceedings of the Thirty-Ninth Annual ACM Symposium on Theory of Computing*, STOC '07, pages 21–30, New York, NY, USA, 2007. Association for Computing Machinery.

[2] Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. Zkboo: Faster zero-knowledge for boolean circuits. In *USENIX Security Symposium*, volume 16, 2016.

[3] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. Snarks for c: Verifying program executions succinctly and in zero knowledge. In *Advances in Cryptology–CRYPTO 2013: 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II*, pages 90–108. Springer, 2013.

[4] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. *Communications of the ACM*, 59(2):103–112, 2016.

[5] Ronald Cramer, Ivan Bjerre Damgård, and Jesper Buus Nielsen. *Secure Multiparty Computation and Secret Sharing*. Cambridge University Press, West Nyack, 2015.

[6] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[7] Joseph-Louis Lagrange. *Lectures on Elementary Mathematics*. Open court publishing Company, 1795. Translated by: Thomas John McCormack.

[8] Oded Goldreich. *Foundations of Cryptography: Volume 1, Basic Tools*. Cambridge University Press, Cambridge, 2001.

[9] Ivan Damgård. On σ-protocols. 2010. `https://www.cs.au.dk/~ivan/Sigma.pdf` [Online; accessed 05-06-2023].

[10] Mihir Bellare and Oded Goldreich. On defining proofs of knowledge. In *Advances in Cryptology–CRYPTO'92: 12th Annual International Cryptology Conference Santa Barbara, California, USA August 16–20, 1992 Proceedings*, pages 390–420. Springer, 2001.

[11] National Institute of Standards and Technology. Security requirements for cryptographic modules. Technical Report Federal Information Processing Standards

Publications (FIPS PUBS) 180-4, U.S. Department of Commerce, Gaithersburg, Maryland, 2015.

[12] Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. Post-quantum zero-knowledge and signatures from symmetric-key primitives. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 1825–1842, New York, NY, USA, 2017. Association for Computing Machinery.

[13] Peter W Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*, pages 124–134. Ieee, 1994.

[14] Troels Henriksen. *Design and implementation of the Futhark programming language*. PhD thesis, University of Copenhagen, Faculty of Science [Department of Computer Science], 2017.

[15] Aaftab Munshi. The opencl specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)*, pages 1–314. IEEE, 2009.

[16] Philip Munksgaard, Svend Lund Breddam, Troels Henriksen, Fabian Cristian Gieseke, and Cosmin Oancea. Dataset sensitive autotuning of multi-versioned code based on monotonic properties: Autotuning in futhark. In *Trends in Functional Programming: 22nd International Symposium, TFP 2021, Virtual Event, February 17–19, 2021, Revised Selected Papers*, pages 3–23. Springer, 2021.

[17] Stefan Tillich and Nigel Smart. (bristol format) circuits of basic functions suitable for mpc and fhe. `https://homes.esat.kuleuven.be/~nsmart/MPC/old-circuits.html`, 2020. [Online; accessed 07-05-2023].

[18] Cynthia A Brewer, Geoffrey W Hatchard, and Mark A Harrower. Colorbrewer in print: a catalog of color schemes for maps. *Cartography and geographic information science*, 30(1):5–32, 2003.

[19] Lov K Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 212–219, 1996.

[20] Ralph C Merkle. A digital signature based on a conventional encryption function. In *Advances in Cryptology–CRYPTO'87: Proceedings 7*, pages 369–378. Springer, 1988.

[21] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and Raphael C-W Phan. Sha-3 proposal blake.

[22] John Kelsey, Shu-jen Chang, and Ray Perlner. Sha-3 derived functions: cshake, kmac, tuplehash, and parallelhash. *NIST special publication*, 800:185, 2016.

[23] Martin R Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for mpc and fhe. In *Advances in Cryptology–EUROCRYPT 2015: 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I 34*, pages 430–454. Springer, 2015.

Appendix A

Benchmark data

A.1 ZKBoo results

The mean for 1000 runs with 137 rounds of the protocol.

No. threads	Proof times (ms)	Verify times (ms)	Total times (ms)
1 thread	69	37	106
8 threads	17	10	27

A.2 ZKG results

End-to-end benchmarks

The mean for 4000 runs with 137 rounds of the protocol (1000 rounds on 4 distinct inputs).

Backend	Proof times (ms)	Verify times (ms)	Total times (ms)
c	139.06	74.41	213.47
Multicore	20.58	12.31	33.12
CUDA	105.56	7.86	113.41
OpenCL	108.82	11.77	120.60

Futhark

The mean for 500 runs with 137 rounds of the protocol.

Backend	Proof times (ms)	Total times (ms)
c	115.25	201.75
Multicore	18.16	33.12
CUDA	13.02	112.91
CUDA, tuned	26.69	18.70
OpenCL	19.38	120.10
OpenCL, tuned	29.92	18.83

Futhark, proof only

Here, all measurements are in microseconds (μs).

Rounds	C	Multicore	CUDA	CUDA, tuned	OpenCL	OpenCL, tuned
1	0.88	0.42	8.23	20.51	8.48	13.48
10	8.58	1.68	8.6	20.57	8.94	13.53
20	16.94	2.9	9.51	20.78	12.52	13.58
50	42.6	6.75	10.83	22.02	17.23	27.02
100	84.61	13.33	13.88	24.73	19.72	29.51
137	115.25	18.16	13.02	26.69	19.38	29.92
200	167.63	26.56	14.08	27.97	21.24	30.76
300	251.67	39.05	15.94	28.24	22.44	30.07
400	335.62	50.68	18.08	28.37	24.9	29.48
500	418.79	62.43	19.3	28.53	27.17	32.29
1000	837.99	122.42	32.36	29.25	41.12	31.8
5000	4186.7	594.68	160.7	43.17	174.56	42.06

A.3 SHA-256 results

Size is the input size in bytes to the SHA-256 function. All measurements are in microseconds (μs) and B is the abbreviation of bytes (mean of 500 runs per input).

Normal SHA-256

Size	C	Multicore	CUDA	CUDA, tuned	OpenCL	OpenCL, tuned
50B	5.7	6.16	707.69	671.36	952.01	908.09
100B	7.45	5.82	709.38	672.78	943.68	906.3
1000B	48.35	44.0	5087.94	5083.65	6920.56	6637.97
5000B	207.42	207.34	24892.68	24695.25	33931.97	32426.67
10000B	386.57	415.84	49436.34	49260.96	66719.32	64234.47
50000B	1888.25	1949.74	245912.58	245303.88	333414.26	320157.56
100000B	3720.89	3659.29	491777.53	485071.18	665595.12	637263.31

Parallel SHA-256 Mock

Size	C	Multicore	CUDA	CUDA, tuned	OpenCL	OpenCL, tuned
50B	1.5	1.18	51.62	32.47	56.95	52.94
100B	1.48	1.93	44.03	33.2	52.75	51.05
1000B	5.05	4.55	48.7	34.24	52.14	50.79
5000B	21.54	17.76	72.56	60.41	83.18	81.58
10000B	41.51	26.07	89.48	146.23	138.19	100.67
50000B	196.28	65.95	232.96	105.32	128.38	122.41
100000B	382.27	114.36	423.45	212.96	164.84	159.36

Prover with Parallel SHA-256 Mock

Rounds	C	Multicore	CUDA	CUDA, tuned	OpenCL	OpenCL, tuned
1	0.6	0.23	7.12	6.66	6.16	6.17
10	3.75	0.85	7.94	6.84	6.3	6.32
20	7.51	1.5	8.82	6.91	6.39	6.41
50	18.84	3.33	11.66	7.37	9.95	9.99
100	37.62	6.54	17.61	9.22	11.95	11.95
137	51.5	8.96	10.82	10.17	12.52	12.52
200	75.46	12.94	11.77	11.04	14.07	14.08
300	112.89	19.48	13.54	12.66	15.06	15.06
400	150.7	25.97	15.45	14.41	16.68	16.69
500	188.11	33.01	16.85	14.76	19.77	18.41
1000	376.8	63.21	28.89	18.81	21.22	21.26
5000	1882.33	298.01	120.93	36.84	48.05	48.03