Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Parallel Golden Collision Search on GPUs

Bachelor Thesis

Oliver Leo Dudler

September 7, 2022

Advisors: Prof. Dr. Kenneth G. Paterson, Dr. Fernando Virdia

Applied Cryptography Group
Institute of Information Security
Department of Computer Science, ETH Zürich

Abstract

The potential development of large-scale quantum computers threatens modern-day public-key encrypted communication methods. As a response, the National Institute of Standards and Technology (NIST) has initiated a standardization process for post-quantum cryptography. We take a look at SIKE, which is one of the Key Encapsulation Mechanisms submitted to NIST, and discuss how the underlying protocol (SIDH) works. Up until recently, it was believed that the best approach to break SIKE is to use the state-of-the-art collision search algorithm (vOW) by van Oorschot and Wiener (J. Cryptology, 1999). The vOW algorithm benefits fully from parallelism. Therefore, we investigate implementing vOW on GPUs. After providing an in-depth explanation of the vOW algorithm, we present an implementation of vOW for GPUs using CUDA and the C language. The collision search attack is run against a toy function based on SHA256. We discuss the problems and performance issues that arise when implementing such an algorithm for GPUs and compare the execution speed with an already existing CPU implementation. An attempt to run an attack on SIKE using our implementation of vOW and existing code for SIDH was made but was unsuccessful.

Contents

Contents iii

1 Introduction 1

2 Background 3
- 2.1 SIKE . . . 3
  - 2.1.1 Mathematical Foundations . . . 3
  - 2.1.2 SIDH . . . 6
- 2.2 vOW Algorithm . . . 8
  - 2.2.1 Runtime Analysis . . . 11
  - 2.2.2 Attacking SIKE Using vOW . . . 12
- 2.3 CUDA . . . 14
  - 2.3.1 GPU Programming in General . . . 15
  - 2.3.2 Programming Model . . . 15
  - 2.3.3 CUDA streams . . . 21
  - 2.3.4 CUDA Events . . . 21
  - 2.3.5 Synchronization . . . 21
  - 2.3.6 Profiling . . . 22

3 General vOW Implementation 25
- 3.1 Implementation for CPUs . . . 25
- 3.2 Concept . . . 25
  - 3.2.1 Datatypes . . . 29
- 3.3 The Function f . . . 31
- 3.4 $setupState$ Kernel . . . 32
- 3.5 $findDistinguishedPoint$ Kernel . . . 32
- 3.6 $locateCollision$ Kernel . . . 37
- 3.7 Distinguishedness Property . . . 38
- 3.8 Core Synchronization . . . 39
- 3.9 Evaluation . . . 39

3.10 Comparison with a CPU Implementation 41
3.11 Possible Improvements 44
3.11.1 Improve *locateCollision* Kernel 44
3.11.2 Tailor Implementation to Specific Parameters 45
3.11.3 Approach Using Streams and Events 45

4 SIKE vOW Implementation 47

5 Conclusion 49

Bibliography 53

Chapter 1

Introduction

There has been much research on developing large-scale quantum computers in the last few years. Such a device could perform calculations that would overwhelm any regular supercomputer. More specifically, using Shor's algorithm [27], quantum computers can factor large integers and solve the discrete logarithm problem in polynomial time. Since these two mathematical problems were assumed to be challenging to solve, most of today's public-key cryptography relies on their hardness. An attacker equipped with the immense computing power of such quantum technology can easily break current security protocols. Quantum computers are, therefore, a real threat to the security of modern public-key cryptosystems.

As a response, cryptographers worldwide have been actively researching methods to develop public-key cryptographic schemes secure even with the existence of quantum computers. In 2016 NIST initiated a standardization process for post-quantum cryptography. In total, 23 signature and 59 encryption schemes were submitted, out of which only 4 made it to the fourth and final round of evaluation and four are already up for standardization (one KEM and three signature schemes). One of the remaining protocols is SIKE. It is the only submitted protocol based on supersingular elliptic curves. Compared with other submissions, SIKE has the advantage of a fairly small key size but the disadvantage of a rather high computational burden on the protocol users.

Remark 1.1 (Efficient attack on SIDH) *In early August, a few weeks before the hand-in of this thesis, Wouter Castryck and Thomas Decru released a paper introducing an efficient key recovery attack on SIDH [4]. Their attack relies on the auxiliary torsion point information that Alice and Bob share during the execution of the protocol (see Section 2.1). This attack breaks SIKE and (at least for the time being) renders it useless. However, the vOW algorithm existed way before SIKE and is therefore still a relevant topic for efficiently finding collisions. Apart from this remark, the thesis is written without knowledge of this attack.*

Analyzing the security of SIKE by determining the cost of the best possible (known) attack against it is crucial to determine the necessary key size for the protocol to reach a certain security level. A larger key size implies better security but also more computation and larger overhead. Cryptanalysis of SIKE [15] suggests that the best known attack is a classical secret-key recovery using the van Oorschot and Wiener parallel golden collision search algorithm (vOW) [30].

Cost estimations of the attack assume a fully parallel implementation with instant synchronization of data across any number of cores. It also assumes that memory is free to access and available in constant time. In practice, however, fully synchronizing cores can be difficult, especially on a larger scale, and memory accesses might be a performance bottleneck.

Attacks on SIKE using the vOW algorithm have already been implemented using CPUs [8] however to the best of our knowledge there exists no implementation of the vOW algorithm on GPUs with available source code.

The contribution of this thesis is a implementation of the vOW algorithm on GPUs using CUDA and C. We analyze different challenges of implementing the algorithm on GPUs, such as core synchronization, memory accesses, and communication between the host and the device. We analyze the cost of running an attack against a SHA256 based toy function using our implementation of the vOW algorithm. We also attempted to run the algorithm against SIKE, but failed for reasons explained in Chapter 4. We compare our results with the expected runtime of vOW [30] and other implementations of vOW on CPUs.

The remainder of this paper is organized as follows. Chapter 2 provides relevant background on SIKE, the vOW algorithm and CUDA. Chapter 3 presents the implementation of the vOW algorithm against an arbitrary hash function. An attempt of implementing an attack against SIKE using our vOW implementation is described in Chapter 4. Finally, Chapter 5 makes some concluding remarks.

Chapter 2

Background

This chapter is meant to give the reader all the necessary background required to follow the implementation details and results. It starts by introducing the SIKE protocol in Section 2.1. Afterward, in Section 2.2 the parallel golden collision search algorithm by van Oorschot and Wiener (vOW) is presented. The chapter concludes by providing an introduction to CUDA in Section 2.3.

2.1 SIKE

SIKE (Supersingular Isogeny Key Encapsulation) is a Key Encapsulation Mechanism (KEM) which is based on the Supersingular Isogeny Diffie-Hellman (SIDH) protocol which was first introduced by de Feo, Jao, and Plût in 2011 [11]. SIKE is one of the 15 remaining third-round candidate PKE proposals submitted to NIST's post-quantum cryptography standardization process. Recently NIST announced that SIKE would advance to the fourth round of evaluation [2]. SIKE is the sole proposal whose security relies on the difficulty of the computational supersingular isogeny (CSSI) problem (see Section 2.1.2). Compared to other proposals, the SIKE protocol has the advantage that it has a relatively small key size, hence a small communication overhead. A disadvantage is the rather high computational burden the SIKE protocol has on both users.

A implementation of SIKE developed by Microsoft Research in 2017 can be found in [25].

2.1.1 Mathematical Foundations

This section is meant to introduce some of the mathematical concepts used in the SIKE or SIDH protocol. A reader familiar with these concepts is welcome to jump to Section 2.1.2.

Finite Fields

A finite field is a field consisting of a finite number of elements. For any positive integer r, there exists a finite field $\mathbb{F}_r$ of order r if and only if r is a power of a prime. SIKE makes use of quadratic extension fields of a finite field $\mathbb{F}_p$ where the extension field is denoted as $\mathbb{F}_{p^2}$.

Elliptic Curves

There exist multiple forms of elliptic curves. The form which we are interested in is the Montgomery curve. Let $A, B \in \mathbb{F}_p$ be elements satisfying $B(A^2 - 4) \neq 0$. A Montgomery curve E over a finite field $\mathbb{F}_p$ is defined as the set S such that all elements $(x, y) \in S$ where $x, y \in \mathbb{F}_p$ satisfy the equation

$$By^2 = x^3 + Ax^2 + x \tag{2.1}$$

Additionally S contains the "point at infinity" $\mathcal{O}$. E forms a group under group addition, and $\mathcal{O}$ is the identity element. The main reason SIDH uses Montgomery curves is that with them, it is possible to compute using efficient x-only arithmetic, i.e., when computing isogenies, we only need to consider the x-component.

There exist ordinary and supersingular elliptic curves [28]. For us, it is not essential to know the difference. SIDH uses supersingular elliptic curves because they have a specific property that makes it harder for an attacker to break SIDH, i.e., there exist subexponential attacks against SIDH using ordinary elliptic curves [5]. However, no such attack exists against SIDH using supersingular elliptic curves.

Another important mathematical construct used in SIDH is the j-invariant of an elliptic curve. Given an elliptic curve E in Montgomery form, its j-invariant is defined by

$$j(E) = \frac{256(A^2 - 3)^3}{(A^2 - 4)} \tag{2.2}$$

The j-invariant of an elliptic curve defined over a field $\mathbb{F}_p$ is unique (up to isomorphisms between two elliptic curves, which preserve the j-invariant).

Torsion Groups

Let E be an elliptic curve defined over a finite field $\mathbb{F}_p$. For any positive integer d, we define the torsion group $E[d]$ to be the set of points $\{P \in E | ord(P) = d\}$, i.e., it holds that $[d]P = \mathcal{O}$

Isogenies

An isogeny $\phi : E_1 \to E_2$ is a non-constant map defined over a finite field $\mathbb{F}_q$, where E_1 and E_2 are two elliptic curves defined over $\mathbb{F}_q$. It is also a group homomorphism from E_1 to E_2, i.e., for two points $A, B \in E_1$ we have $\phi(A + B) = \phi(A) + \phi(B)$.

There exist separable and non-separable isogenies. Knowing the difference is out of scope for this thesis. A interested reader can find an explanation in [19, §2.2]. SIKE uses separable isogenies for the following reason: For every subgroup $G \subseteq E$ of an elliptic curve E, there exists a unique (up to isomorphisms) separable isogeny $\phi : E \to E/G$ whose kernel is G. The kernel of an isogeny consists of all elements $e \in E$ such that $\phi(e) = \mathcal{O}$. Another important aspect is that the degree of such an isogeny matches the number of elements in the kernel, i.e., $deg(\phi) = |G|$. For the rest of this thesis, when talking about isogenies, we always refer to separable isogenies.

Given an elliptic curve E and a subgroup $G \subseteq E$, one can use Vèlu's formula to calculate the isogeny $\phi : E \to E/G$. It is important to note that Vèlu's formula is polynomial in the degree of the isogeny ϕ. Therefore it is not feasible to calculate high-degree isogenies using Vèlu's formula. In SIKE, we solve this problem by using the following property of isogenies:

The composition of two isogenies $\phi_A : E \to E'$ and $\phi_B : E' \to E''$ is another isogeny $(\phi_B \circ \phi_A) : E \to E''$ and it holds that $deg(\phi_B \circ \phi_A) = deg(\phi_A) \cdot deg(\phi_B)$. Therefore the computation of a high-degree isogeny can be broken down into a series of computations of low-degree isogenies.

Lastly, unlike isomorphisms, isogenies generally do not preserve the j-invariant of an elliptic curve. This property gives rise to isogeny graphs, a fundamental part of the SIKE protocol.

Isogeny Graphs

Given a finite field $\mathbb{F}_{p^2}$ where p is some large prime, we are interested in a subset of about size $\lfloor \frac{p}{12} \rfloor$, namely the set of all supersingular j-invariants in $\mathbb{F}_{p^2}$. Every elliptic curve has a unique j-invariant (up to isomorphisms). The nodes of an isogeny graph are exactly these j-invariants.

We now define the d-isogeny graph as the graph with nodes as mentioned above. An edge exists between two nodes if and only if there exists a d-isogeny (an isogeny of degree d) between the two elliptic curves represented by the j-invariants of the two nodes. Such isogeny graphs have some interesting properties: They are undirected graphs, as for each isogeny $\phi : E \to E'$ there exists a dual isogeny $\hat{\phi} : E' \to E$ of the same degree. The d-isogeny graph is $(d + 1)$-regular (apart from a few exceptions) and it has

an extremely short diameter ($\mathcal{O}(\log p)$) [11, §2]. Recall that a graph is called d-regular if all nodes in the graph have degree d.

In SIKE, we only utilize two isogeny graphs, namely the 2-isogeny and the 3-isogeny graph (It is not necessary to choose these two graphs for the protocol to work, but choosing the two smallest primes currently provides the most efficient version of SIKE).

2.1.2 SIDH

SIDH is one of the first isogeny-based key-agreement protocols. It is called Supersingular Isogeny Diffie-Hellman because it uses supersingular isogenies as a basis. Its core idea is the same as in the well-known Diffie-Hellman protocol: Each user creates their private key. Using their private key and globally defined parameters, they compute their public key. Both users proceed by exchanging their public keys. Lastly, they compute the final shared secret key using their private and the other's public key.

High-Level Overview

As usual, we will let Alice and Bob be the two participants in the protocol.

Let $p = 2^{e_A}3^{e_B} - 1$ be a prime and let $\mathbb{F}_{p^2}$ be the respective quadratic extension field. Let E be an elliptic curve in Mongomery form defined over $\mathbb{F}_{p^2}$. e_A, e_B, p and E are fixed and publicly known. For example in $SIKEp434$ we have $e_A = 216$ and $e_B = 137$ such that $2^{216} \approx 3^{137}$ and p is a 434 bit long prime (hence the name). The reason we want $2^{216} \approx 3^{137}$ is that both Alice and Bob have about the same computational expense.

Let $P_A, Q_A \in E$ be two points such that the set $\{P_A, Q_A\}$ is a basis for the torsion group $E[2^{e_A}]$. Similarly, the set $\{P_B, Q_B\}$ is a basis for $E[2^{e_B}]$. P_A, Q_A, P_B, Q_B are also publicly known and fixed.

In the first step of the protocol, Alice computes a secret linear combination R_A of her two public basis points, i.e., Alice computes

$$R_A = [m_A]P_A + [n_A]Q_A \text{ with } m_A, n_A \in_R [0, 2^{e_A}) \tag{2.3}$$

Similarly Bob computes

$$R_B = [m_B]P_B + [n_B]Q_B \text{ with } m_B, n_B \in_R [0, 3^{e_B}) \tag{2.4}$$

Let $G_A = \langle R_A \rangle$ be the subgroup generated by R_A. Alice now computes her secret isogeny $\phi_A : E \to E_A$ where $E_A = E/G_A$. As mentioned, computing a isogeny of degree d using Vélu's formula is polynomial in d. Since ϕ_A has degree 2^{e_A}, directly computing ϕ_A is not feasible. Instead, Alice computes

e_A isogenies of degree 2 by taking e_A steps (defined by R_A) in the respective 2-isogeny graph. Alice's public key is then defined by

$$PK_A = (E_A, P'_B, Q'_B) = (\phi_A(E), \phi_A(P_B), \phi_A(Q_B)). \tag{2.5}$$

Bob follows the same steps, except that he computes an isogeny of degree 3^{e_B} by computing e_B isogenies of degree 3 by taking e_B steps (defined by R_B) in the respective 3-isogeny graph. Bob's public key is similarly defined as

$$PK_B = (E_B, P'_A, Q'_A) = (\phi_B(E), \phi_B(P_A), \phi_B(Q_A)). \tag{2.6}$$

The second and third part of Alice's public key contains the images of Bob's basis points P_B, Q_B under Alice's secret isogeny ϕ_A (and vice versa for Bob's public key). This information is essential so that in the next step of the protocol, both parties can do the same computation again on their "new" basis points so that, in the end, both end up with the same shared secret.

After exchanging their public keys, Alice proceeds by computing a second linear combination $R'_A = [m_A]P'_A + [n_A]Q'_A$ using the same secret values m_A, n_A. Let $G'_A = \langle R'_A \rangle$ and let $E_{AB} = E_B/G'_A$. Alice computes another secret isogeny $\phi'_A : E_B \to E_{AB}$ in the same way as before. Bob performs analogous computations. Using $R'_B = [m_B]P'_B + [n_B]Q'_B$ and $G'_B = \langle R'_B \rangle$ he computes his second secret isogeny $\phi'_B : E_A \to E_{BA}$ where $E_{BA} = E_A/G'_B$.

In a final step both Alice and Bob compute their shared secret $ss = j(E_{AB}) = j(E_{BA})$ by taking the j-invariant of the respective elliptic curve. Alice and Bob compute the same secret ss since $\phi''_A := \phi'_A \circ \phi_A$ and $\phi''_B := \phi'_B \circ \phi_B$ both have kernel $\langle R_A, R_B \rangle$, hence ϕ''_A and ϕ''_B are the same isogeny over $\mathbb{F}_{p^2}$ (up to composition with an isomorphism over $E/\langle R_A, R_B \rangle$). It follows that E_{AB} and E_{BA} are isomorphic over $\mathbb{F}_{p^2}$ resulting in them having the same j-invariant.

A more detailed description of the SIDH, including toy examples, can be found in [7].

CSSI

In the same way that the classical Diffie-Hellman scheme relies on the hardness of the discrete logarithm problem, SIDH relies on the hardness of the Computational SuperSingular Isogeny problem (CSSI). One of the main assumptions made in the SIDH protocol is that the extra information given in the public keys $(\phi_A(P_B), \phi_A(Q_B), \phi_A(P_B), \phi_A(Q_B))$ do not help a passive adversary to solve the CSSI problem. Hence CSSI is defined as follows:

Definition 2.1 (CSSI) *Given the SIDH parameters $e_A, e_B, p, E, P_A, Q_A, P_B, Q_B$ and E/A, compute a degree-2^{e_A} isogeny $\phi_A : E \to E/A$. Or equivalently: determine a generator for A.*

That assumption still holds today, and no known passive attacks exist that use the extra information to break CSSI. Some potential active attacks which use this "torsion-point information" are mentioned in [16].

SIDH vs SIKE

One can build a PKE scheme based on SIDH (for example by using a modification of Hashed ElGamal [10]), that is secure against passive adversaries, i.e., IND-CPA secure. However, when using static (long-term) keys, attacks exist such that an active adversary can figure out the secret values and break the protocol. An example of such an attack is mentioned in [7].

Hence, in the first round of the NIST PQC standardization process, SIKE has been proposed as a secure key encapsulation mechanism to eliminate this static-key vulnerability. SIKE is an optimized, IND-CCA secure version of SIDH. The Fujisaki-Okamoto Transformation [12] [14], along with some other minor technical changes, was used to transform SIDH into SIKE. One technical change for example, is that the secret generator R_A is of the form $R_A = P_A + [n_A]Q_A$ for some (almost) randomly selected $n_A \in_R [0, 2^{e_A})$ (and similarly for R_B).

2.2 vOW Algorithm

The vOW algorithm is a parallel collision search algorithm introduced by Paul C. van Oorschot, and Micheal J. Wiener in 1996 [30].

Let S be a finite set with $|S| = n$ finite, and let $f : S \to S$ be a function where finding collisions is of interest. A collision is a pair of values $x_a, x_b \in S$ such that $x_a \neq x_b$ and $f(x_a) = f(x_b)$. In cryptography we usually work with random functions which sometimes might not be of the form $f : S \to S$. When applying the vOW algorithm in practice, one would have to adapt the function for which one wants to find collisions in order to create a matching f for the algorithm. For simplicity, we assume f to be an arbitrary random function in this section.

There exist various kinds of algorithms to find collisions for a function f (such as Meet-In-The-Middle (MITM) attacks), altough they tend to require at least $\mathcal{O}(\sqrt{n})$ memory [13, §5.4]. In any realistic scenario, n is such a massive value that it is infeasible to have that amount of storage.

The vOW algorithm solves this issue by only using a variable amount of storage. It uses so-called distinguished points and only stores these in memory. A point $x_d \in S$ is a distinguished point if it fulfills some easily verifiable property. This property could, for example, be a certain number of leading zeros. Depending on the amount of available memory, one can choose the probability of a point being distinguished to be higher or lower.

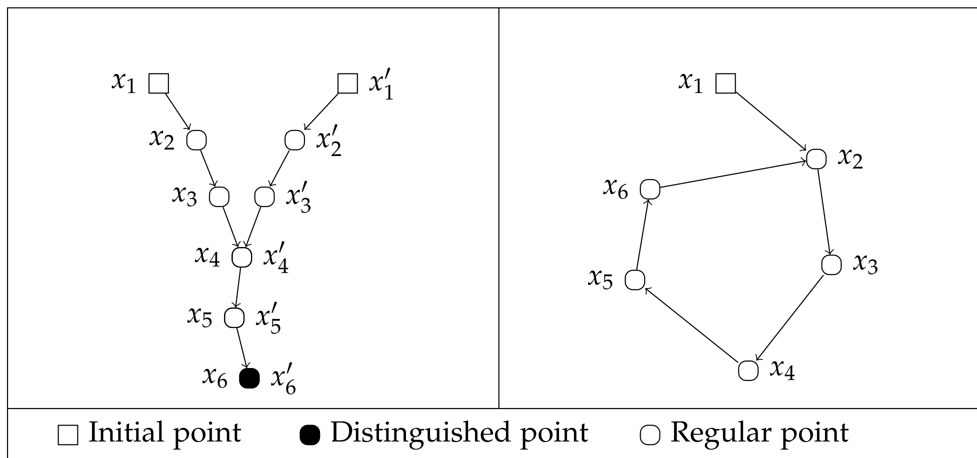

Figure 2.1: Two trails colliding, reaching the same distinguished point (left) and a trail entering a cycle (right). An edge denotes one iteration of the function f

Denote θ as the probability of a point being distinguished. The algorithm now proceeds by performing multiple walks in parallel as follows: Each processor starts its walk at a randomly selected point $x_0 \in_R S$ and continues by repeatedly applying f such that we get a sequence $x_i = f(x_{i-1})$ for $i = 1, 2, \ldots$ until a distinguished point x_d is found. The algorithm then stores the triple (x_0, x_d, d) in memory.

It is important to note that while generating a trail of points, a processor can enter a cycle of points that do not contain a distinguished point (See Figure 2.1). van Oorschot and Wiener handle this in their paper by setting a maximum path length of $\frac{20}{\theta}$. Whenever a path exceeds this length without finding a distinguished point, the processor aborts and restarts at a new randomly selected starting point.

Let w be the number of triples that fit in memory. To simplify memory access, it makes sense to store the triple at a memory location which is a fixed function of the distinguished point.

When storing the triple (x_0, x_d, d) in memory, one of the following three cases can occur:

1. The location we want to store the triple in is empty: In this case, we store the triple and proceed.
2. The location we want to store the triple in already contains another triple (x'_0, x'_d, d') with $x_d \neq x'_d$: This can be the case since we may have more distinguished points than fit into memory, i.e., multiple distinguished points map to the same memory address. Since the distinguished points are not equal, we found no collision. We replace the old triple with the new one.

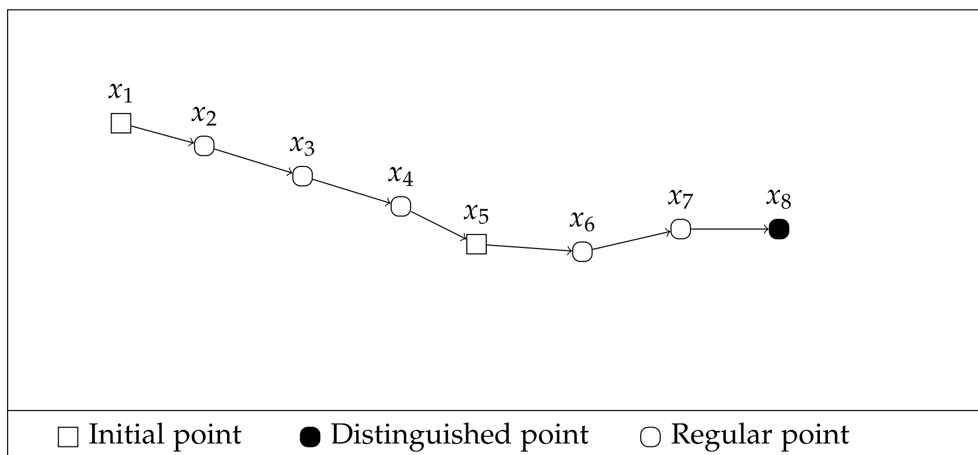

Figure 2.2: Example of a "Robin-Hood"

3. The location we want to store the triple in already contains another triple (x'_0, x'_d, d') with $x_d = x'_d$: In this case we detected a collision (assuming $x_0 \neq x'_0$). We can now use these two triples to locate the collision.

We detected a collision w.r.t. the two triples (x_0, x_d, d) and (x'_0, x'_d, d') with $x_d = x'_d$, $x_0 \neq x'_0$. Assume w.l.o.g. that $d > d'$ (if $d = d'$ we can skip this next step). We can locate the collision by iteratively applying $x_0 \leftarrow f(x_0)$ and $d \leftarrow d - 1$ until $d = d'$. If at this point we have $x_0 = x'_0$ than a so-called "Robin Hood" occured (see Figure 2.2). A "Robin Hood" is when one sequence collides with the starting point of another sequence, such that both end up at the same distinguished point. Trivially this isn't a real collision that we care about, but van Oorschot and Wiener mention that in practice θ is such a small value that, since these point sequences have expected length $\frac{1}{\theta}$, this happens rarely. Whenever it does happen, we ignore the found "collision" and move on.

However, if at this point we have $x_0 \neq x'_0$ since both points are now equally far away from the same distinguished point ($d = d'$), we can "walk along" both trails step-by-step simultaneously until we end up at the same value, which means we found our collision.

In the simple case, one is only interested in finding any collision of the function f, in which case the algorithm terminates at this point.

The more challenging case is if one is interested in finding a specific "golden collision". A random function $f : S \rightarrow S$ is expected to have roughly $\frac{n}{2}$ collisions[1]. Therefore, assuming all collisions are equally likely to occur,

[1] This is the case because there are $\frac{n(n-1)}{2}$ pairs of points in S and the chance of two points colliding is $\frac{1}{n}$. Multiplying these two terms gives approximately $\frac{n}{2}$

we need to find $\frac{n}{2}$ collisions in expectancy before succeeding, and we also require an efficient way to test if a located collision is the golden one. The algorithm continues to generate distinguished points and locate collisions until the golden collision is found.

One key aspect of the vOW algorithm, if we are looking for a "golden-collision", is that we must repeat the process described above multiple times (finding and locating a certain number of distinguished points and collisions), often having to exchange the function f with a new version that still preserves the "golden collision". The reason is that for a specific function version f_i, where the i denotes the i-th version, the probability of detecting the golden collision may be very low. This might be because one or both of the two points which form the golden collision have very few (or no) pre-images under f_i or because the trails leading up to these points are very short. Hence we have a small chance of finding them.

According to van Oorschot and Wiener, the best average runtime is achieved by finding a fixed number of distinguished points for every version of f, restarting the algorithm each time we reach that number without locating the golden collision.

Some details must be considered when implementing this algorithm in practice (especially on GPU). We will talk more about those in Chapter 3.

2.2.1 Runtime Analysis

Van Oorschot and Wiener provide an approximate runtime analysis of their algorithm to get a general idea of the runtime formula. They then examine the performance using various simulations to obtain the unknown constants in their formula. We will still include their approximate analysis for completeness:

Assume the available memory is fully occupied, holding w distinguished points. Each path leading up to a distinguished point has expected length $\frac{1}{\theta}$, hence the total amount of points on these paths is about $\frac{w}{\theta}$. Each time any processor finds a new point (either by starting a trail at a new randomly selected point, or by performing an iteration of the function f on the previous point), the chance of that point lying on any of the trails stored in memory is $\frac{1}{n} \cdot \frac{w}{\theta} = \frac{w}{n\theta}$. It follows that in expectancy $\frac{n\theta}{w}$ new points need to be generated in order to find a collision. Locating a collision requires $\frac{2}{\theta}$ steps on average, as we need to walk along both paths up to the colliding point. The total cost of detecting and locating a collision is therefore $\frac{n\theta}{w} + \frac{2}{\theta}$ function iterations. Setting $\theta = \sqrt{2w/n}$ minimizes this term to $\sqrt{8n/w}$. As previously mentioned, in expectancy we need to find $\frac{n}{2}$ collisions before finding the golden collision, leading to a total cost of $\frac{n}{2} \cdot \sqrt{8n/w} = \sqrt{2n^3/w}$ function iterations.

As mentioned, this analysis has a few flaws. For example, the analysis does not consider the time the algorithm needs to fill the memory. During that phase, we have fewer points in memory; hence, the odds of finding a collision are worse than when the memory is full. Other flaws will not be discussed here but can be found in their paper [30, §4.2].

Let $\alpha, \beta > 0$ be some constants and let $\theta = \alpha\sqrt{w/n}$. In their experiments, they let each function version run for the amount of time it takes to find βw distinguished points. For each function version, they counted the number of function iterations (i) and the amount of distinct collisions found (c). Using these variables, the expected number of function versions before finding the golden collision is then $\frac{n}{2c}$ implying a runtime of $\frac{in}{2c}$ function iterations. Running simulations for multiple values of α, β revealed that the runtime is minimized for $\alpha = 2.25$ and $\beta = 10$. Using these values in further simulations, their analysis concludes by stating that for $w \geq 2^{10}$ the runtime T can be overestimated by

$$T = \frac{2.5t}{m}\sqrt{\frac{n^3}{w}} \tag{2.7}$$

where the 2.5 is a constant which they determined heuristically and was verified multiple times (for example by Adj et al. [1] or by Craig Costello et al. [8] using an AES-based XOF as f). m denotes the number of processors and t the time needed for one iteration of the function f. Note how in theory, this algorithm parallelizes perfectly, while in practice, this is not the case (especially for large m and n) due to memory access and synchronization issues.

2.2.2 Attacking SIKE Using vOW

Let E denote the initial elliptic curve in SIKE and let E/A denote the curve onto which Alice's secret isogeny $\phi_A : E \to E/A$ maps. The length of the path between E and E/A in the 2-isogeny graph (which is e_A) is much smaller than the average distance between two nodes in the graph; hence it is incredibly likely that the path Alice took to get from E to E/A (by computing 2-isogenies) is the shortest path connecting these two nodes [7, §7]. Therefore, the most efficient attacks for computing ϕ_A are meet-in-the-middle (MITM) attacks, where we compute paths of length $\frac{e_A}{2}$ starting from E and E/A until two of these paths "meet in the middle". It is then exceedingly likely that these two connecting paths of length $\frac{e_A}{2}$ form the exact path of length e_A that Alice took to compute ϕ_A. By finding that path, the attack is completed as we can quickly compute Alice's secret isogeny ϕ_A.

The standard version of a MITM attack works as follows: The algorithm works on the 2-isogeny graph defined over the field $\mathbb{F}_{p^2}$ for a given prime p. In the first step, we compute all elliptic curves, which are $2^{\frac{e_A}{2}}$-isogenous to

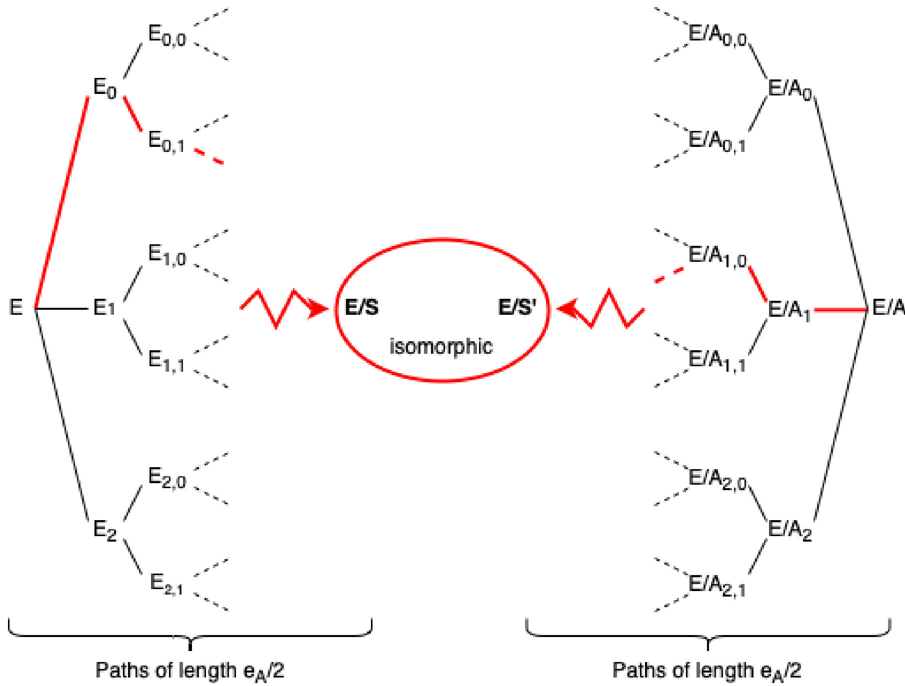

Figure 2.3: MITM attack on a 2-isogeny graph. The red path symbolizes the collision we are looking for

E and store them in memory. We do this by computing all possible paths of length $\frac{e_A}{2}$ starting from E in the 2-isogeny graph. Next, we compute paths of length $\frac{e_A}{2}$ starting from E/A and compare the elliptic curves they end on with those stored in memory. As soon as we found a collision, i.e., we found a curve E/S' which is isomorphic to another curve E/S stored in memory, we found the path connecting E and E/A and hence we found $\phi_A = \phi_{S'} \circ \psi \circ \phi_S$, where $\phi_S : E \to E/S$ and $\phi_{S'} : E/S' \to E/A$ are $2^{\frac{e_A}{2}}$-isogenies and $\psi : E/S \to E/S'$ denotes the isomorphism between the two "colliding" elliptic curves E/S and E/S'. Figure 2.3 illustrates the MITM attack described above. Recall that the 2-isogeny graph is (almost) 3-regular. Hence, computing a path of length $\frac{e_A}{2}$ starting from E (or E/A) is like following a random path down a binary tree of depth $\frac{e_A}{2}$ (except for the first step), where each step corresponds to computing one possible 2-isogeny.

In a paper from 2018 [1], Adj et al. showed that this generic version of a MITM attack is not the best algorithm for attacking SIKE. Although it has a reasonably good runtime complexity, its exponential memory requirements are unrealistic and impractical. They proposed to set a limit of $w = 2^{80}$ (which corresponds to multiple yottabytes) for the amount of available memory in order to be able to analyze the security of SIKE correctly. Their anal-

ysis concluded that, although it has a worse runtime complexity, the vOW algorithm is the best-known algorithm for finding Alice's secret isogeny ϕ_A due to its smaller (and variable) memory usage.

Defining the Function f

In order to apply the vOW algorithm, we need a set of functions $\{f_v : S \to S\}$ where $v \in \mathbb{N}$ such that they all have the "golden collision" and they all act like random functions.

For the simplicity of notation let $E_0 := E$ and $E_1 := E/A$. Let $\mathcal{G}_i$ denote the set of all order-$2^{\frac{e_A}{2}}$ subgroups of $E_i[2^{e_A}]$ for $i \in \{0,1\}$. Let $I = \{1,2,\dots,N\}$ and $S = \{0,1\} \times I$, where $N = |\mathcal{G}_0| = |\mathcal{G}_1|$. Intuitively, N is the number of elliptic curves which are $2^{\frac{e_A}{2}}$-isogenous to E_0 (E_1) and therefore N is also the number of possible paths of length $\frac{e_A}{2}$ starting from E_0 (E_1). Let $\mathcal{J}_i$ denote the set of all j-invariants of elliptic curves that are $2^{\frac{e_A}{2}}$-isogenous to E_i and let $\mathcal{J} = \mathcal{J}_0 \cup \mathcal{J}_1$.

Let $g_n : \mathcal{J} \to S$ for $n \in \mathbb{N}$ be a set of random functions, let $k_i : \mathcal{G}_i \to \mathcal{J}_i$ be defined by $k_i(G) = j(E_i/G)$ where $G \in \mathcal{G}_i$ and finally let $h_i : I \to \mathcal{G}_i$ be two bijections. We can now define our set of random-acting functions $f_v : S \to S$ as

$$f_v(i,x) = g_v(k_i(h_i(x))) \tag{2.8}$$

where f_v denotes the v-th function version. By construction, each function version f_v has multiple individual collisions, which occur due to the randomness of the function g_v (and because $|\mathcal{J}| > |S|$), but they all share the same "golden collision" that is of interest, which occurs due to $k_i \circ h_i$. We now expect to have two unique subgroups $G_0 \in \mathcal{G}_0$ and $G_1 \in \mathcal{G}_1$ such that $j(E_0/G_0) = j(E_1/G_1)$. Let $z_i = h_i^{-1}(G_i)$ where h_i^{-1} exists since h_i are bijections for $i \in \{0,1\}$. The "golden collision" we want to locate is then defined by the points $(0, z_0)$ and $(1, z_1)$.

2.3 CUDA

CUDA (Compute Unified Device Architecture) is a computing platform and programming model introduced by NVIDIA in 2006. Its purpose is to enable general-purpose programming on GPUs. CUDA can be used with many programming languages, including C, C++, or Python. It is a heterogeneous programming language, as its code runs on the CPU (referred to as the host) and the GPU (referred to as the device).

This section provides an introduction to CUDA and its programming model. It is based on the CUDA toolkit documentation by NVIDIA [23]. An interested reader can find more detailed descriptions there.

2.3.1 GPU Programming in General

As mentioned above, CUDA is supposed to enable general-purpose programming on GPUs. However, why would we want that? Originally, GPUs were designed as specialized processors to accelerate graphics and image rendering in, e.g., gaming or video editing. They had a relatively narrow range of applications. This changed quite a bit over the last few years, partially due to the shift towards parallel programming. The most famous example is probably the extensive use of GPUs for mining cryptocurrency [3]. In general, GPUs seem to become more in use in several scientific fields, one of which is cryptography [21] [20] [24].

The CPU and GPU were designed with entirely different goals in mind. The CPU was modeled to handle various tasks quickly but suffers from lousy parallelizability. This is where the GPU comes in. GPUs are perfect for monotonous (few branches) and highly-parallel computation. They provide way higher instruction throughput and memory bandwidth than a CPU with a similar price. Hence, using GPUs might be desirable for well parallelizable applications.

2.3.2 Programming Model

Kernels

Kernels are the part of CUDA code that executes on the GPU. They are a special kind of function that the host can call. A kernel is defined by using the __global__ keyword. Kernels run on the device using so-called CUDA threads. The main difference between CUDA threads and CPU threads is that the former are incredibly lightweight, and kernels can run on thousands of CUDA threads simultaneously. In contrast, multi-core CPUs usually only run a few threads simultaneously. In the CUDA programming model, threads organize into groups of threads called thread blocks.

When calling a kernel, one has to specify the number of blocks B and threads per block T to execute the kernel on. Each thread (block) running the kernel is then given a unique thread-id (block-id) within the kernel, which can be used to, e.g., access memory locations with a specific offset.

```
1 // Kernel definition
2 __global__ void VecMult(int* a, int* b, int* r) {
3     int i = threadIdx.x;
4     r[i] = a[i] * b[i];
5 }
```

```
6
7  int main() {
8      ...
9      // Kernel invocation with 1 block and n threads
10     VecMult<<<1, n>>>(a, b, r);
11     ...
12 }
```

Listing 2.1: Kernel example

Listing 2.1 shows a code example of a small kernel, which given two vectors a, b of length n calculates the coefficient-wise product r. The two values inside the triple-brackets denote B and T.

Kernels are the only function type that allows device code to be called from the host code. Of course, there also exist "normal" functions being called and executed on the host or device directly. These must be declared with the __host__ or __device__ keywords, respectively. These keywords instruct the compiler to compile the function as host or device code. Using both keywords simultaneously is also possible, causing the compiler to compile the functions as both. A function is compiled as host code per default when no keyword is used.

```
1  //device-callable function
2  __device__ void multBy2(int* a, int i) {
3      a[i] = 2*a[i];
4  }
5
6  //host-callable function
7  void hostFunction() {
8      ...
9  }
10
11 //Kernel definition
12 __global__ void VecMult(int* a, int* b, int* r) {
13     int i = threadIdx.x;
14     r[i] = a[i] * b[i];
15     multBy2(a, i);
16 }
17
18 int main() {
19     ...
20     // Kernel invocation with 1 block and n threads
21     VecMult<<<1, n>>>(a, b, r);
22     hostFunction();
23     ...
24 }
```

Listing 2.2: Device-callable and host-callable functions

Listing 2.2 shows a code example including one device-callable function invoked by the kernel and one host-callable function invoked by the main

function. Notice how the host-callable function does not necessarily require the __host__ tag.

Usually a kernel launch is divided into three main steps:

- Transfer data from host memory to device memory
- Load and execute the kernel code
- Transfer data from device memory back to host memory

Memory must first be allocated both on the host and the device. Allocating memory on the device can be achieved using the *cudaMalloc* function. The *cudaFree* function can free the allocated memory afterward. The *cudaMemcpy* function is used to copy memory from the host to the device or vice versa. *cudaMemcpy* is a synchronous function, so the CPU halts execution until the function returns. Listing 2.3 shows an example of a complete CUDA program, multiplying coefficient-wise two vectors of length N.

```
1  #include <stdio.h>
2
3  // Kernel definition
4  __global__ void VecMult(int *a, int *b, int *r) {
5      int i = threadIdx.x;
6      r[i] = a[i] * b[i];
7  }
8
9  int main() {
10     int n = 1000;
11     //allocate host memory
12     int* a = (int*) malloc(n*sizeof(int));
13     int* b = (int*) malloc(n*sizeof(int));
14     int* r = (int*) malloc(n*sizeof(int));
15
16     for (int i = 0; i < n; i++)
17         a[i] = b[i] = 2;
18
19     //allocate device memory
20     int *a_d, *b_d, *r_d;
21     cudaMalloc(&a_d, n*sizeof(int));
22     cudaMalloc(&b_d, n*sizeof(int));
23     cudaMalloc(&r_d, n*sizeof(int));
24
25     //copy memory from host to device
26     cudaMemcpy(a_d, a, n*sizeof(int), cudaMemcpyHostToDevice);
27     cudaMemcpy(b_d, b, n*sizeof(int), cudaMemcpyHostToDevice);
28
29     // Kernel invocation with 1 block and N threads
30     VecMult<<<1, n>>>(a_d, b_d, r_d);
31
32     //copy memory from device to host
33     cudaMemcpy(r, r_d, n*sizeof(int), cudaMemcpyDeviceToHost);
34
35     //prints out "4" n times
```

```
36     for (int i = 0; i < n; i++) {
37         printf("%d", r[i]);
38     }
39 
40     //free allocated memory
41     free(a);
42     free(b);
43     free(r);
44     cudaFree(a_d);
45     cudaFree(b_d);
46     cudaFree(r_d);
47 }
```

Listing 2.3: Complete CUDA program (without error handling)

Another essential aspect of kernels is that they are asynchronous to the CPU. If one were to call multiple Kernels in succession, they would be called immediately without waiting for the prior ones to finish. On the GPU side, however, the invoked kernels will be executed serially in the order the host called them. However, running multiple kernels in parallel using CUDA streams is possible, as we will see in Section 2.3.3.

Thread Hierarchy

As previously mentioned, threads organize into groups of threads called thread blocks. These blocks can be one, two, or three-dimensional, meaning the respective thread indices are 3-component vectors. In Listing 2.3, we only used one dimension for the thread indices (threadIdx.x). The thread index being a 3-component vector provides an easy and more natural way of performing computations using vectors, matrices, or volumes. Thread blocks are similarly grouped into a grid and can be addressed using their block-id.

```
1  //Kernel definition
2  __global__ void MatrixAdd(int* a, int* b, int* r, int m, int n) {
3      int i = blockIdx.x * blockDim.x + threadIdx.x;
4      int j = blockIdx.y * blockDim.y + threadIdx.y;
5      if (i < m && j < n)
6          r[i*n+j] = a[i*n+j] + b[i*n+j];
7  }
8 
9  int main() {
10     ...
11     // Kernel invocation using 2-dimensional indices
12     dim3 threadsPerBlock(16,16);
13     dim3 blocks(m/threadsPerBlock.x, n/threadsPerBlock.y);
14     MatrixAdd<<<blocks, threadsPerBlock>>>(a_d, b_d, r_d, m, n);
15     ...
16 }
```

Listing 2.4: Matrix addition

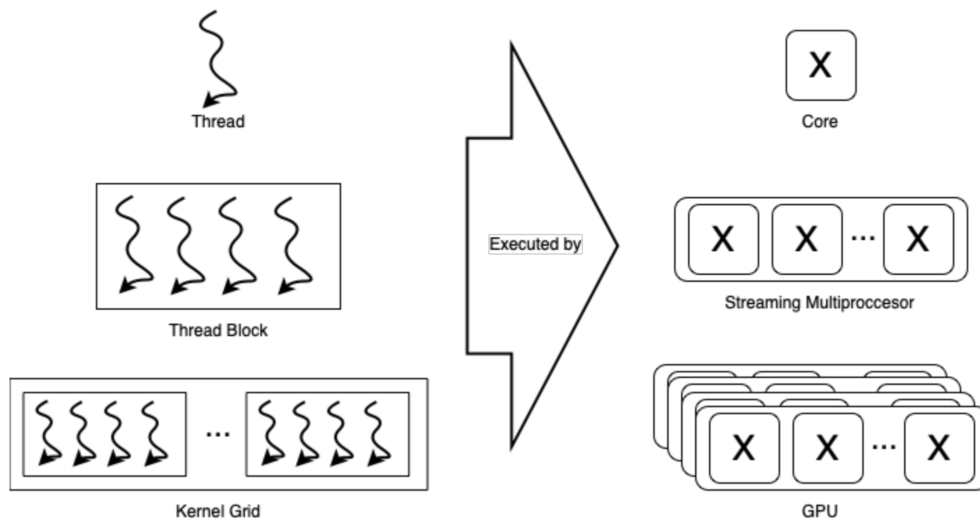

Figure 2.4: Illustrating the programmer's perspective (left) and the hardware perspective (right) of the GPU.

Listing 2.4 shows an example of a kernel computing the addition of two mxn-matrices using thread and block indices such that each thread only needs to calculate one element of the resulting matrix.

Threads within the same block share some resources of the GPU; hence limiting thread block sizes is crucial for performance. The CUDA toolkit documentation [23] recommends using block sizes of 128 or 256, though it might depend on the application. Current GPUs support thread blocks containing up to 1024 threads. A GPU contains multiple streaming multiprocessors (SM). A SM consists of multiple general purpose processors and it's main task is to execute several thread blocks in parallel. When starting a CUDA program, all thread blocks are distributed among available SMs. Threads within a thread block execute concurrently only on their given SM. Within a thread block, threads are grouped into warps (typically of size 32). Threads within a warp are then scheduled and executed as a warp unit, executing the exact instructions simultaneously. Suppose, for some reason, threads within a warp unit end up executing different instructions at the same time. In that case, a warp-divergence occurs, and the execution becomes serialized, drastically decreasing the performance. Hence it is crucial that threads within a warp unit execute the same instructions on different data to maximize performance.

Figure 2.4 shows the correlation between the programmer's perspective and the hardware perspective of the GPU.

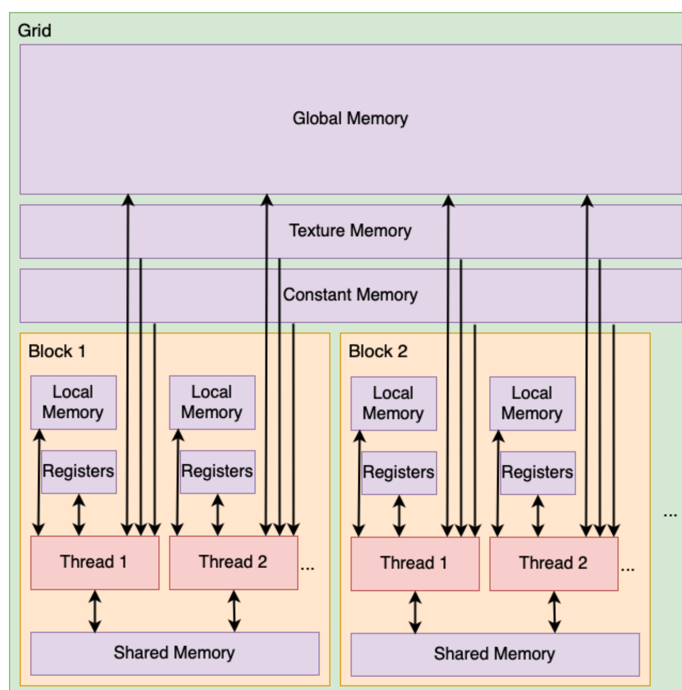

Figure 2.5: Illustration of the CUDA memory model

Memory Hierarchy

CUDA threads can read or write to various memory spaces during their lifetime. Figure 2.5 illustrates these memory spaces and their scope.

Each thread has its dedicated registers and local memory to which it can read and write. Threads within the same thread block have access to common shared memory, while all threads have access to the global memory of the GPU. There are also two globally scoped read-only memory spaces called texture memory and constant memory. The texture memory is optimized for spatial access patterns (i.e. it improves performance in graphics applications where memory access patterns have a lot of spatial locality, for example when loading textures), while the constant memory is where constants and kernel arguments are stored.

Registers and shared memory have the fastest access time by far, implying that threads within the same block can communicate more efficiently than threads from different blocks, as they would be required to use global memory. It is also important to note that local memory might have a misleading name. Even though it is local, it has about the same throughput and latency as global memory (it is also off-chip). It is called local memory because it is only accessible by one thread. Local memory is mainly used for register spilling. Register spilling occurs during compilation (specifically during the

register allocation phase) whenever there are more live variables than available registers. Therefore, the compiler has to ”spill” some variables from registers into memory (in our case, into the local memory).

2.3.3 CUDA streams

It is possible to run multiple kernels in parallel using CUDA streams. Each stream represents a queue of device work; the host can add work to these queues asynchronously. The device then schedules work from these streams whenever resources become available. CUDA operations in different streams are unordered and may overlap. Unless otherwise specified, all calls to the device, including kernels and memory copies, are placed into a default stream. In order to run parallel kernels or any other device calls, one can create and destroy non-default streams using *cudaStreamCreate* and *cudaStreamDestroy*.

In order to execute kernels in parallel, they must be in different non-default streams. This is because the default stream is always synchronous to all other streams, i.e., it can not overlap with others. Also, enough resources on the device must be available for all kernels to run simultaneously. For memory copies to execute in parallel, the same conditions as above apply, but additionally, we need to use the asynchronous function *cudaMemcpyAsync*. Also, we need to ensure there is no other memory copy happening in the same direction (i.e., host to device) contemporaneously and that the copy uses pinned memory on the host. Pinned memory can not be paged out by the operating system and can be allocated using *cudaMallocHost* or *cudaHostAlloc*.

2.3.4 CUDA Events

CUDA Events are synchronization markers and can be used to notify the host when certain operations have happened in a stream. They help measure timings or synchronize CUDA streams, for example. Events have a boolean state, meaning they are either “occurred” or “not-occurred”. *cudaEventCreate* creates an event and sets it to “occurred” by default. *cudaEventDestroy* can be used to destroy events, and *cudaEventRecord* adds an event to the specified stream and sets it to “not-occurred”. The event gets set to “occurred” as soon as it reaches the front of the stream.

2.3.5 Synchronization

There are many ways to perform synchronization in CUDA. Some functions are implicitly synchronized like for example *cudaMemcpy* or *cudaMalloc*.

There are 3 ways to explicitely synchronize the device with the host:

- *cudaDeviceSynchronize* blocks the host until all threads are done executing on the device.
- *cudaStreamSynchronize* blocks the host until all threads of a specific stream are done executing.
- *cudaEventSynchronize* blocks the host until the specified event"occured"

Synchronization within the device between threads of the same block can be achieved by calling *__syncthreads*. Synchronization between non-default streams is done using *cudaStreamWaitEvent*.

2.3.6 Profiling

Since CUDA programming is highly parallel, getting code to work can be frustrating. Fortunately, there exist profiling tools that help analyze and optimize the program.

One such profiling tool is The NVIDIA Visual Profiler. The NVIDIA Visual Profiler was introduced in 2008 and is part of the CUDA Toolkit [23]. It is a GUI-based profiling tool and provides information like a unified CPU and GPU timeline of the program, a guided application analysis that advises on possible optimizations, and much more. A complete overview can be found on the NVIDIA webpage [22].

Figure 2.6 shows a screenshot of the Visual Profiler GUI showing the unified CPU and GPU timeline and the guided application analysis.

Another profiling tool is nvprof, which allows users to collect and view profiling data directly from the command line. Nvprof is extremely useful for fast sanity checks during programming or short summaries to get an overview of the program's GPU kernels and memory copies as displayed in Figure 2.7.

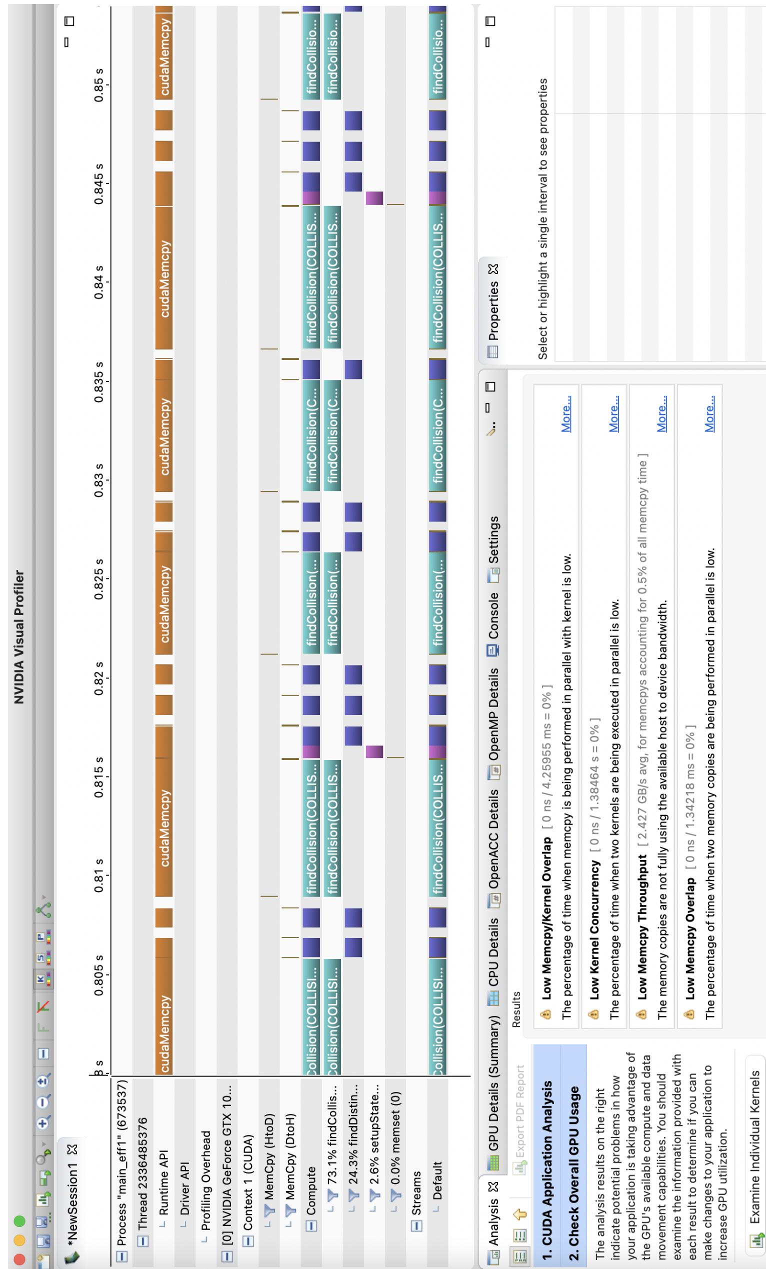

Figure 2.6: Screenshot of the NVIDIA Visual Profiler GUI

```
==674025== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   72.68%  971.22ms       150  6.4748ms  3.9953ms  10.276ms  findCollision(COLLISION*, int, HASHCOLLISI
                   24.31%  324.80ms       321  1.0118ms  958.17us  1.2348ms  findDistinguishedPoint(curandStateXORWOW*,
                    2.68%  35.837ms        50  716.74us  521.97us  994.72us  setupStateKernel(curandStateXORWOW*, unsig
                    0.21%  2.8214ms       942  2.9950us  2.3040us  10.081us  [CUDA memcpy DtoH]
                    0.10%  1.3360ms       151  8.8470us  3.0720us  9.1850us  [CUDA memcpy HtoD]
                    0.02%  245.22us       150  1.6340us  1.1200us  2.5280us  [CUDA memset]
      API calls:   88.11%  1.31130s      1093  1.1997ms  8.1140us  10.304ms  cudaMemcpy
                    7.56%  112.54ms         5  22.508ms  1.3040us  112.53ms  cudaMalloc
                    2.43%  36.232ms        50  724.65us  532.02us  994.48us  cudaDeviceSynchronize
                    1.68%  24.977ms         1  24.977ms  24.977ms  24.977ms  cudaDeviceReset
                    0.15%  2.2208ms       521  4.2620us  2.4230us  19.284us  cudaLaunchKernel
                    0.04%  534.57us       150  3.5630us  1.4790us  20.084us  cudaMemset
                    0.02%  262.97us       101  2.6030us     192ns  179.62us  cuDeviceGetAttribute
                    0.01%  125.50us         5  25.100us  2.4330us  107.72us  cudaFree
                    0.00%  67.812us       521     130ns     105ns     308ns  cudaGetLastError
                    0.00%  27.665us         1  27.665us  27.665us  27.665us  cuDeviceGetName
                    0.00%  7.8300us         1  7.8300us  7.8300us  7.8300us  cuDeviceGetPCIBusId
                    0.00%  1.6860us         3     562ns     284ns  1.0810us  cuDeviceGetCount
                    0.00%  1.2270us         2     613ns     207ns  1.0200us  cuDeviceGet
                    0.00%     615ns         1     615ns     615ns     615ns  cuDeviceTotalMem
                    0.00%     330ns         1     330ns     330ns     330ns  cuDeviceGetUuid
```

Figure 2.7: Screenshot of an output given by nvprof

Chapter 3

General vOW Implementation

In this chapter, we provide an implementation of the vOW algorithm using CUDA and C. We discuss the challenges and pitfalls that arise while implementing vOW for GPUs. After discussing the specific implementation details, we evaluate the implementation by providing some benchmarks. We also compare our algorithm with the run-time of a vOW implementation for CPUs [18]. In the last section, we discuss possible improvements of our code, as it is by no means perfect.

3.1 Implementation for CPUs

Before we discuss how to implement vOW for GPUs, let us look at how one would implement the vOW algorithm for CPUs on a high level. As mentioned in Section 2.2, the vOW algorithm parallelizes well. If we have N CPU cores at our disposal, we can let each core run the algorithm (i.e., find distinguished points and locate potential collisions) more or less independently. Cores would not be running fully independently, as synchronization on the currently used function version is required. Managing access to the memory containing the (at most w) stored distinguished points is also necessary.

Figure 3.1 illustrates the general concept of a CPU implementation. Each core more or less independently runs the algorithm until one of the cores finds the golden collision.

3.2 Concept

We have seen how one could implement the vOW algorithm for CPUs. Now let us look at what we are interested in: Implementing vOW for GPUs.

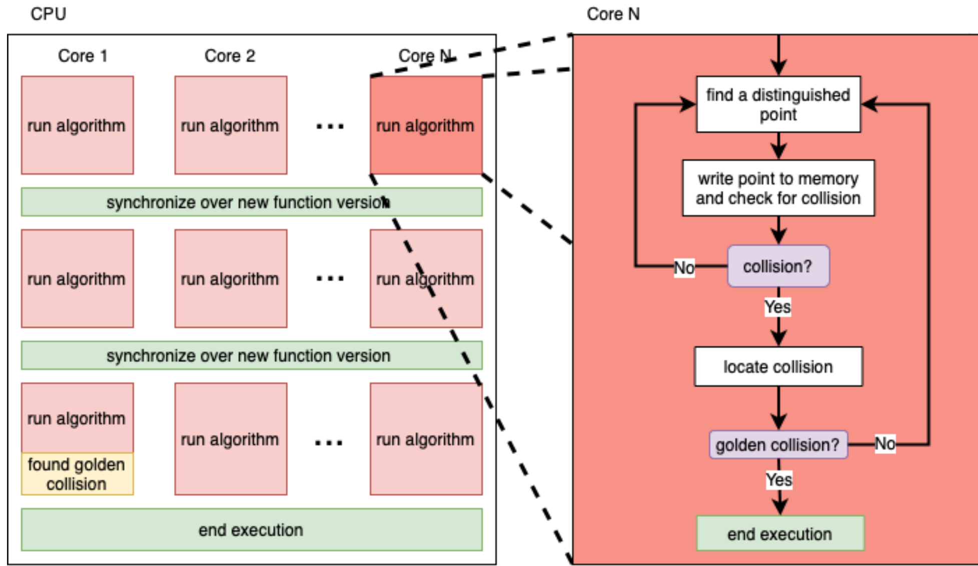

Figure 3.1: Illustration showing the concept of a CPU implementation of the vOW algorithm.

The first idea that comes to mind is to try and use the same concept for CPUs: Write one huge kernel where each thread executes the algorithm independently, similar to Figure 3.1. One immediately runs into several complications and performance issues when attempting to implement it this way. First, as we have seen in Section 2.3, the GPU has a completely different architecture than the CPU. Hence code gets executed in entirely different ways. On a GPU, threads can not simply execute code independently (unlike CPU threads) since they are executed in warps. Hence if one were to implement the algorithm described above, every single warp would experience huge warp divergence. This is the case because different threads will execute different instructions simultaneously (by the nature of the algorithm, as some threads will find distinguished points earlier than others, for example). Due to this divergence, barely any work will be done in parallel, and the execution will boil down to essentially a serial execution. Another problem with this implementation is that the memory model in a GPU setting differs. The above implementation would require us to store all our found distinguished points in the global memory located on the device. This is the case because it needs to be accessible by all threads executing the kernel. However, typically a GPU only has 1-16 GB of global memory, which is certainly not enough for large w (and n). It follows that we can not simply implement the algorithm for GPUs in the same way as for CPUs.

We apply two key changes in order to make our GPU implementation feasible and more efficient:

- Split up the algorithm into multiple kernel calls: Instead of running

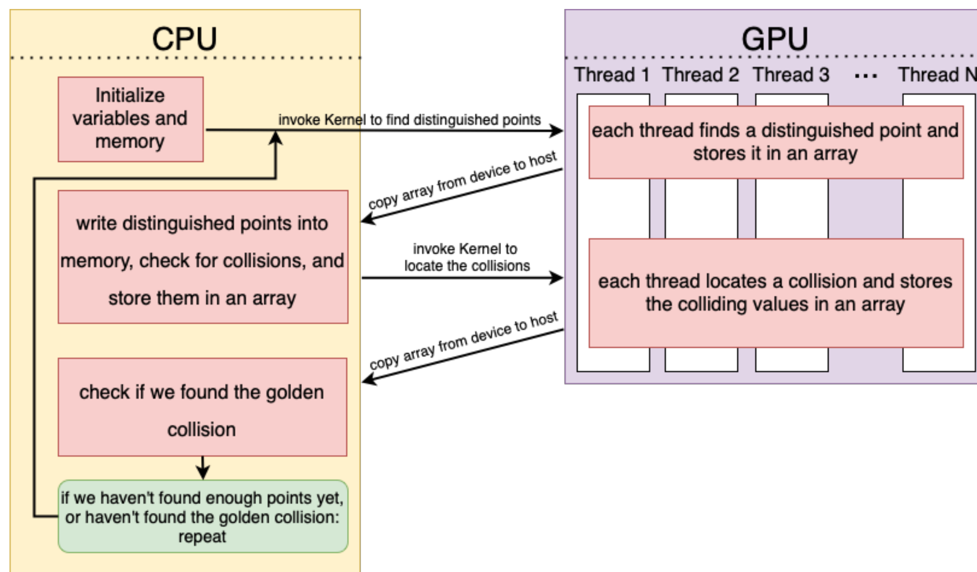

Figure 3.2: Illustration showing the concept of a GPU implementation of a single function version of the vOW algorithm.

the entire algorithm inside a huge kernel, we split the execution into smaller kernels and invoke these multiple times. This change mitigates warp divergence as we can minimize the number of branches inside a kernel, and threads are less likely to diverge.

- Store the distinguished points in host memory: Host memory is generally much more extensive than device memory. Hence we have a better chance of possessing enough memory to store our distinguished points. We make the CPU responsible for writing distinguished points into host memory and checking if a collision occurs. The GPU is responsible for finding distinguished points and locating the (potential) collisions found by the CPU.

Figure 3.2 illustrates the general concept of a GPU implementation. It only shows the execution for one function version, i.e., until $\beta \cdot w$ distinguished points are found. This process will be repeated until the golden collision is discovered. Notice how the threads do not execute the algorithm independently anymore. Instead, they execute the same part of the algorithm simultaneously. Also, note how the execution is split up into multiple kernel calls and how the CPU is responsible for managing the memory containing the distinguished points and checking for collisions.

Listing 3.1 shows the implementation of the *main* function following the concept shown in Figure 3.2. Note that any error handling and collecting statistics have been removed to make the code more readable. The full code can be inspected on GitLab [9].

```
1  int main () {
2
3      srand(time(NULL));
4      //generate random golden collision
5      GOLDENCOLLISION gc = getGoldenCollision();
6
7      //initialize functionVersion to zero
8      unsigned int functionVersion = 0;
9
10     //allocate memory
11     curandState* d_states;
12     DIST_POINT* memory;
13     DIST_POINT* points;
14     DIST_POINT* d_points;
15     DPCOLLISION* collisions;
16     DPCOLLISION* d_collisions;
17     VALUECOLLISION* collision_results;
18     VALUECOLLISION* d_collision_results;
19
20     memory = (DIST_POINT*) calloc(W, sizeof(DIST_POINT));
21     points = (DIST_POINT*) calloc(THREADS, sizeof(DIST_POINT));
22     collisions = (DPCOLLISION*) calloc(THREADS*2, sizeof(DPCOLLISION));
23     collision_results = (VALUECOLLISION*) calloc(THREADS, sizeof(VALUECOLLISION));
24
25     cudaMalloc(&d_states, THREADS*sizeof(curandState));
26     cudaMalloc(&d_points, THREADS*sizeof(DIST_POINT));
27     cudaMalloc(&d_collisions, THREADS*sizeof(DPCOLLISION));
28     cudaMalloc(&d_collision_results, THREADS*sizeof(VALUECOLLISION));
29
30     //run the algorithm for NUMBER_OF_FUNCTIONVERSIONS function versions
31     while (functionVersion < NUMBER_OF_FUNCTIONVERSIONS) {
32
33     //stores the number of found points per function version
34     unsigned long long totalPoints = 0;
35     //stores the number of found collisions which we haven't located yet
36     unsigned long num_collisions = 0;
37
38     //reset the array containing the found distinguished points to zero
39     cudaMemset(d_points, 0, THREADS*sizeof(DIST_POINT));
40
41     //invoke to setup the state for future randomizations
42     setupState<<<B,T>>>(d_states, rand(), functionVersion);
43     cudaDeviceSynchronize();
44
45     //for each function version we find 10*W distinguished points
46     do {
47
48     //stores the number of found points for this iteration
49     unsigned long pointsFound = 0;
50
51     //invoke kernel
52     findDistinguishedPoint<<<B,T>>>(d_states, d_points, functionVersion, gc);
53
54     //copy results from device memory to host memory
55     cudaMemcpy(points, d_points, THREADS*sizeof(DIST_POINT), cudaMemcpyDeviceToHost);
56
57     //write the distinguished points we found into the memory and store potential
        collisions in the collisions array
58     int_tuple res = writeBackPoints(memory, points, collisions, THREADS, pointsFound,
        num_collisions);
59     pointsFound = res.a;
60     num_collisions = res.b;
61     totalPoints += pointsFound;
62
63     //if we found enough collisions, or its the last iteration, run the collision-find
        kernel
64     if (num_collisions >= THREADS || totalPoints >= BETA*W) {
65         cudaMemcpy(d_collisions, collisions, THREADS*sizeof(DPCOLLISION),
66          cudaMemcpyHostToDevice);
67         locateCollision<<<B,T>>>(d_collisions, MIN(THREADS, num_collisions),
68             d_collision_results, functionVersion, gc);
69
70         //after copying to device memory, we put the excess collisions found to the
        start of the array
```

```
71          if (num_collisions > THREADS)
72              memcpy(collisions, (collisions + THREADS),
73                  (num_collisions - THREADS) * sizeof(DPCOLLISION));
74
75          cudaMemcpy(collision_results, d_collision_results,
76              THREADS* sizeof(VALUECOLLISION), cudaMemcpyDeviceToHost);
77
78          for (int j = 0; j < MIN(THREADS, num_collisions); j++) {
79
80              //check if its a robin hood
81              if (equal(collision_results[j].a, collision_results[j].b)) {
82                  continue;
83              }
84
85              if (isGoldenCollision(collision_results[j], gc)) {
86                  printf("SUCCEDED");
87                  goto end;
88              }
89          }
90
91          num_collisions = num_collisions % THREADS;
92      }
93
94
95      } while (totalPoints < BETA*W);
96
97      }
98
99      end:
100
101     //free all allocated memory.
102     free(memory);
103     free(points);
104     free(collision_results);
105     free(collisions);
106     cudaFree(d_points);
107     cudaFree(d_states);
108     cudaFree(d_collision_results);
109     cudaFree(d_collisions);
110
111     return 0;
112 }
```

Listing 3.1: Implementation of the main function

3.2.1 Datatypes

Before we explain the code example in Listing 3.1 any further, we introduce the utilized datatypes, mostly structs. Figure 3.3 shows the four structs used in the code examples. *BYTE* denotes an *unsigned char* while *SIZE* denotes the size of the values we work with in bytes. It holds that $SIZE = \lceil \frac{\log_2 n}{8} \rceil$. We use *DIST_POINT* to represent distinguished points. The struct is similar to the tuple-representation (x_0, x_d, d) used in Section 2.2, but additionally we also have the *valid* field. The purpose of the *valid* field will be explained in Section 3.5. We use the *GOLDENCOLLISION* struct to represent our hard-coded golden collision, as further explained in Section 3.3. The *VALUECOLLISION* and *DPCOLLISION* structs represent two colliding values or two potentially colliding *DIST_POINT*s, respectively.

Let us now look at Listing 3.1. In lines 3-28, we initialize our variables and allocate the memory we need. As a convention, all variables starting with

```
1 typedef struct {
2   BYTE start[SIZE];
3   BYTE end[SIZE];
4   BYTE valid;
5   unsigned int steps;
6 } DIST_POINT;
```

```
1 typedef struct {
2   BYTE value1[SIZE];
3   BYTE value2[SIZE];
4   BYTE result[SIZE];
5 } GOLDENCOLLISION;
```

```
1 typedef struct {
2     BYTE a[SIZE];
3     BYTE b[SIZE];
4 } VALUECOLLISION;
```

```
1 typedef struct {
2     DIST_POINT a;
3     DIST_POINT b;
4 } DPCOLLISION;
```

Figure 3.3: Datatypes

"$d_$" denote variables that refer to memory on the device. One iteration of the while loop in line 31 corresponds to executing the vOW algorithm for one function version. After initializing other variables, we invoke the *setupState* kernel, which essentially sets up the states *d_states*, which we require so that each thread can randomly sample starting points to find distinguished points. More details about the *setupState* kernel are given in Section 3.4. In line 46 we enter a do-while loop, which executes until we found $\beta \cdot w$ distinguished points, i.e., $totalPoints \geq \beta \cdot w$. In each loop iteration, we execute the *findDistinguishedPoint* kernel in line 53, where each thread attempts to find a distinguished point. These points are stored in the *d_points* array and copied back to host memory in line 56. In line 59-62, the CPU writes the found points into memory and checks for potential collisions, as described in Section 2.2. We store the number of potential collisions found in *num_collisions* and the number of valid distinguished points found in *totalPoints*. Note that during the execution of the *findDistinguishedPoint* kernel, not every thread necessarily succeeds in finding a distinguished point, as we will see in section Section 3.5. Until now, the code executes precisely as illustrated in Figure 3.2. Instead of invoking the *locateCollision* kernel each time after invoking *findDistinguishedPoint*, as shown in Figure 3.2, we add the conditional statement in line 65. We only invoke the *locateCollision* kernel if we have found enough potential collisions such that each thread has a collision to locate ($num_collisions \geq THREADS$). This change improves performance because if we invoke the *locateCollision* kernel each time, many threads would be idle since after calling the *findDistinguishedPoint* kernel, not all threads have found a distinguished point. Also, not every distinguished point leads to a potential collision. Hence $num_collisions \ll THREADS$ holds after one iteration (especially for small w). If the condition on line 65 holds, we invoke the *locateCollision* kernel and copy the array of colliding values back to host memory in lines 66-77. Lastly, in lines 79-90, we check if any collisions we found correspond

to our golden collision.

3.3 The Function f

Recall from Section 2.2 that in order to apply the vOW algorithm, we require a function f of the form $f : S \rightarrow S$ for some set S where $|S| = n$. Since we are merely interested in providing a generic algorithm implementation in this chapter, there are plenty of options for f. We use SHA256 as the underlying cryptographic hash function to define f since it appears to behave as a random function. Luckily there already exists an implementation of SHA256 for GPUs by Brad Conte [6], which we used for our implementation. Additionally, we define S as the set of all n-bit strings. We trim the output of the SHA256 algorithm to only the first n bits. For f to have a single golden collision, the call to SHA256 is wrapped with additional logic, which enforces a collision with only two input values.

```
1  __device__ void f(BYTE* indata, BYTE* outdata, unsigned int
       functionVersion, GOLDENCOLLISION gc) {
2    //hard-coded golden collision
3      if (equal(indata, gc.value1) || equal(indata, gc.value2))
       {
4      for (int i = 0; i < SIZE; i++)
5        outdata[i] = gc.result[i];
6
7      } else {
8          //SHA256 with the function version as a salt
9          CUDA_SHA256_CTX ctx;
10         cuda_sha256_init(&ctx);
11         cuda_sha256_update(&ctx, (BYTE *)&functionVersion, 4)
     ;
12         cuda_sha256_update(&ctx, indata, SIZE);
13         cuda_sha256_final(&ctx, outdata);
14
15         //apply a bit mask, in case n isn't divisible by 8
16     outdata[SIZE-1] = outdata[SIZE-1] & MASK;
17
18         //ensures that the golden collision is unique, i.e.,
     no other value maps to it.
19         if (equal(outdata, gc.result))
20       outdata[0] += 1;
21     }
22 }
```

Listing 3.2: Function f based on SHA256

Listing 3.2 shows the implementation of f. The function takes as input *indata* (which is the value we want to apply f to), *gc* (which is the golden col-

lision we want to hard-code into f) and $functionVersion$ (the current function version). f outputs $outdata$ which contains $f(indata)$. In lines 3-5, we hard-code the golden collision. In lines $9-13$, we apply SHA256 to $indata$ (Notice how in line 11 we use the function version as a salt, such that for each function version, f is different). Line 16 applies a bit-mask to the output in case $\log_2 n$ is not divisible by 8. Finally, in lines 19-20, we ensure that the golden collision is unique.

3.4 *setupState* Kernel

The $setupState$ kernel is responsible for creating a state of type $curandState$ for every thread. This state enables each thread to generate random starting points to find distinguished points. Listing 3.3 shows the implementations of the $setupState$ kernel. In line 3, we call the $curand_init$ function, which initializes a state, given a seed, sequence, and offset. The exact implementation details of the $curand$-functions are not important here but can be found in the CUDA toolkit documentation [23]. What is important for us is that we make the state of each thread dependent on the thread-id and the function version. Hence each thread generates different starting points compared to other threads or function versions. That is the case because the state fully determines the sequence of pseudorandom numbers that the function $curand_uniform$ returns. $curand_uniform$ returns a sequence of pseudorandom floats uniformly distributed between 0.0 and 1.0, and we will need this function to implement the $findDistinguishedPoint$ kernel, as we will see in Section 3.5.

```
1 __global__ void setupState(curandState* state, unsigned long
     seed, unsigned int functionVersion) {
2     int idx = blockIdx.x*blockDim.x + threadIdx.x;
3     curand_init(seed + functionVersion, idx, 0, &state[idx]);
4 }
```

Listing 3.3: *setupState* Kernel

3.5 *findDistinguishedPoint* Kernel

When executing the $findDistinguishedPoint$ kernel, each thread should be (attempting to) find a distinguished point and return it to the CPU as part of an array. In Section 2.2, we discussed how a processor would sample a starting point and then repeatedly apply f until it finds a distinguished point. We also discussed how after at most $\frac{20}{\theta}$ steps (where θ denotes the probability of a point being distinguished), the processor "gives up" and samples a new starting point in order to avoid being trapped inside a cycle. When implementing the vOW algorithm for CPUs, this approach is what is

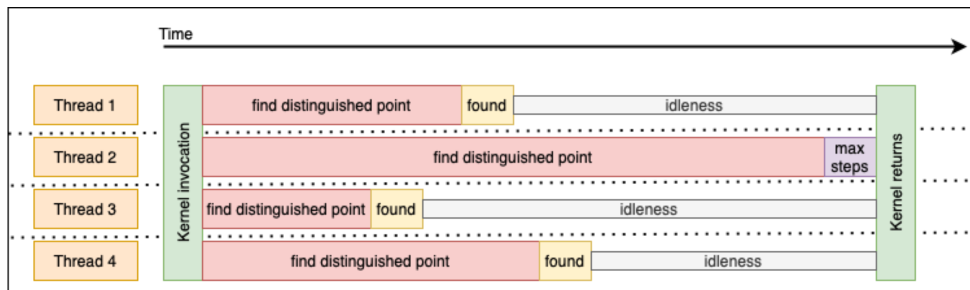

Figure 3.4: Illustration showing how threads that find a distinguished point quickly become idle

used. For GPUs, however, this approach is very inefficient. The reason is similar to before: Since threads are scheduled and executed in warps, they cannot simply execute the algorithm fully independently. Also, the kernel does not return until all threads are done executing. Hence, if we have many threads, chances are high that one thread will not find a distinguished point, i.e., it will apply f the maximum amount of times ($\frac{20}{\theta}$). Most other threads will have found a distinguished point and are idle, waiting for the slower threads to finish execution. Figure 3.4 illustrates this scenario: Thread 2 is unsuccessful in finding a distinguished point. Threads 1,3, and 4 are idle, waiting for thread 2 to finish.

The key idea to boost performance is to split the process into multiple iterations. Instead of allowing a thread to perform up to $\frac{20}{\theta}$ applications of f, we limit the amount to (for example) only $\frac{1}{\theta}$ steps. If a thread does not succeed in $\frac{1}{\theta}$ steps, it stores the current point it is at and continues the search from the same point in the next invocation of the kernel. If a thread has not found a distinguished point after 20 such iterations, in the next iteration, it does not resume from where it left off, but instead, it samples a new starting point. This way, the maximum amount of steps a thread performs per starting point is still $\frac{20}{\theta}$. Hence we still avoid cycles. This change boosts performance, as kernel execution is much shorter now, and thread-idleness is mitigated. Of course, some threads will still be faster than others, and some idleness is unavoidable. There is a tradeoff between making the kernel shorter (allowing fewer steps per iteration to reduce idleness) and the overhead of invoking a kernel and copying memory from the host to the device and vice versa.

In our implementation, we introduce the $GAMMA_FACTOR$ variable. A thread performs at most $\frac{1}{\theta}/GAMMA_FACTOR$ steps per iteration and at most $GAMMA \cdot GAMMA_FACTOR$ iterations per sampled starting point, where $GAMMA$ is defined to be 20 (as specified in the vOW algorithm). A higher $GAMMA_FACTOR$ value implies shorter kernels, less thread-idleness, and more kernel-overhead. A lower $GAMMA_FACTOR$ value implies longer kernels and more thread-idleness but, therefore, less kernel-

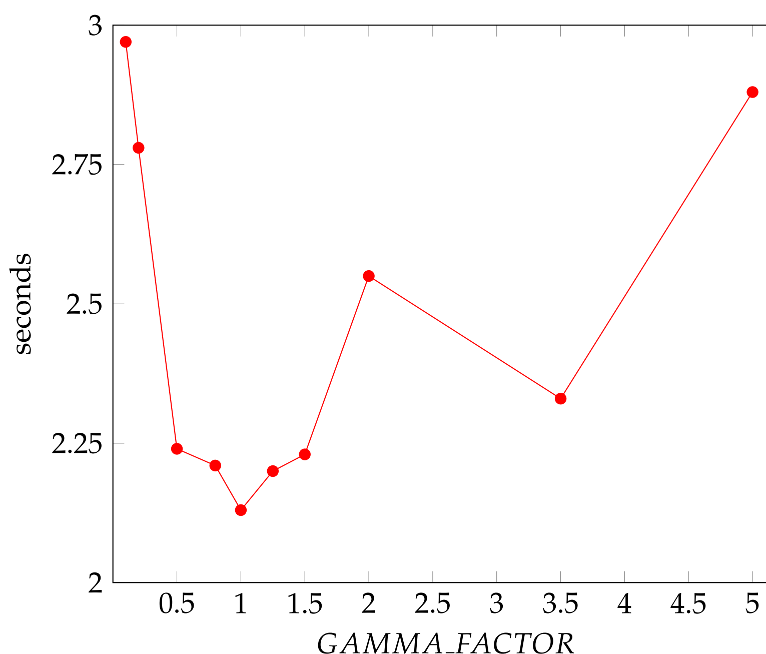

Figure 3.5: Graph showing the tradeoff between thread idleness and kernel invocation overhead for different values of *GAMMA_FACTOR*. The x-axis shows the value of *GAMMA_FACTOR* while the y-axis shows the average time for finding the golden collision in seconds (with $n = 2^{20}, w = 2^{10}$).

overhead. In order to find the optimal value for *GAMMA_FACTOR* we ran the full attack using the parameters $n = 2^{20}$ and $w = 2^{10}$ for different values of *GAMMA_FACTOR*. Figure 3.5 shows that $GAMMA_FACTOR = 1$ seems to be the optimal value. In our implementation, we define *GAMMA_FACTOR* to be equal to 1.

Let us now look at the concrete implementation details. Recall from Section 3.2.1 the definition of the *DIST_POINT* struct. We now make use of the *valid* field. The *valid* field is a single byte, where the MSB is 1 if the corresponding *DIST_POINT* represents a valid distinguished point and 0 otherwise. The other 7 bits are used as a counter, as depicted in Figure 3.6.

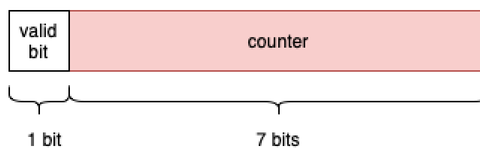

Figure 3.6: Illustration of the *valid* field in the *DIST_POINT* struct.

Let us go through an example: Let (x_0, x_d, d, c, v) denote an instance of

$DIST_POINT$ where x_0 is the starting point, x_d the current point, d the number of steps performed, c denotes the current counter value and v denotes the valid bit (the MSB of $valid$). Let m denote the maximum number of steps per iteration. A thread samples a starting point x'_0 and starts executing with $(x'_0, x'_0, 0, 1, 0)$. If after m steps, the thread couldn't find a distinguished point, it stores $(x'_0, x_m, m, 1, 0)$ in the array (where x_m is the result of applying f m times to x'_0), and in the next iteration it resumes execution while increasing the counter by 1, i.e., we have $(x'_0, x_m, m, 2, 0)$. However, If the thread is successful after $m' \leq m$ steps, it stores $(x'_0, x'_m, m', 0, 1)$ in the array. Notice how this tuple now represents a valid distinguished point. Lastly, assume the thread hasn't found anything in 20 iterations, i.e., after the 20th iteration, the thread stored $(x'_0, x_{20m}, 20m, 20, 0)$ in the array. In the next iteration, the thread notices that the counter value is 20 and samples a new starting point.

Listing 3.4 provides an implementation of the $findDistinguishedPoint$ kernel. In line 6, we set $maxSteps$ as the maximum number of steps per iteration. $DIST_BITS$ denotes the number of bits that determine whether a point is distinguished or not and is derived from θ. As mentioned above, it holds that $GAMMA_FACTOR = 1$ and therefore$maxSteps = \frac{1}{\theta}$ and we allow at most 20 iterations with the same starting point. We will discuss the distinguishedness property in Section 3.7. In lines 13-27, we sample a new starting point and initialize the fields. Note how we sample a new point if:

- We found a distinguished point in the last iteration, i.e., the MSB of $valid$ is 1 ($p.valid \geq 20$)
- The counter contained in the $valid$ field exceeds 20 ($p.valid \geq 20$)
- This is the first time we run the kernel for this specific function version ($p.valid = 0$)

The third reason is why we set the counter to 1 initially. It enables us to distinguish the case where we run the kernel for the first time and therefore trivially need to sample a starting point (when the counter is 0) from the case where the counter is smaller than 20, and we resume with the same starting point. We use the $curand_uniform$ function to sample pseudo-random floats between 0.0 and 1.0 as mentioned in Section 3.4. In lines 36-47, we repeatedly apply f until we find a distinguished point. The second condition in line 38 enforces that we do not consider an instance of $DIST_POINT$ to represent a valid distinguished point if the sampled starting point is a distinguished one. This is helpful since the tuple $(x_0, x_0, 0, 0, 1)$, where x_0 is distinguished, is useless for finding collisions. Also, consider the case where either preimage of the golden collision is a distinguished point (for a specific function version). It would be impossible to find the golden collision, as there would be no way to "pass through " the golden collision (we would always stop the search at one of the preimages). By adding the second con-

dition, we at least have a chance of finding the golden collision by directly sampling one of the preimages.

```
__global__ void findDistinguishedPoint(curandState* states, DIST_POINT* d_points,
    unsigned int functionVersion, GOLDENCOLLISION gc) {

    //number of steps in current iteration
    unsigned long long numSteps = 0;
    //maximum number of steps per iteration
    unsigned long long maxSteps = pow(2, DIST_BITS) / GAMMA_FACTOR;
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    curandState localState = states[idx];

    DIST_POINT p = d_points[idx];

    //if p is valid, expired, or new: sample
    if (p.valid >= GAMMA * GAMMA_FACTOR || p.valid == 0) {
        //sample starting point using curand_uniform
        for (int i = 0; i < SIZE; i++) {
            float myRandF = curand_uniform(&localState);
            myRandF *= 255.999999;
            p.start[i] = p.end[i] = (BYTE) truncf(myRandF);
        }
        //apply mask
        p.start[SIZE-1] = p.start[SIZE-1] & MASK;
        p.end[SIZE-1] = p.end[SIZE-1] & MASK;

        //when starting at new sampled point, reset steps and initialize valid to 1
        p.steps = 0;
        p.valid = 1;

    } else {
        //we resume from where we left off, increment counter
        p.valid += 1;
    }

    //write state back such that we only need to setup the state once
    states[idx] = localState;
    //"walk" along the random function until we find a distinguished point or reach
    maxSteps
    while (numSteps < maxSteps) {
        //if the point is not distinguished or we just seeded a new point, commence
        if (!distinguished(p.end, (BYTE *)&functionVersion) || (p.steps == 0 &&
    numSteps == 0)) {
            f(p.end, p.end, functionVersion, gc);
            numSteps++;
        } else {
            //if we find a distinguished point, set the valid-bit and the steps field
            p.steps += numSteps;
            p.valid = 0x80;
            break;
        }
    }

    //adjust the steps field if we didn't find a distinguished point yet.
    if (numSteps >= maxSteps) {
        p.steps += maxSteps;
    }

    //write back the "saved" points to global memory
    d_points[idx] = p;
}
```

Listing 3.4: Implementation of the *findDistinguishedPoint* kernel

3.6 *locateCollision* Kernel

Our implementation of the *locateCollision* kernel is straightforward. Given an array of *DPCOLLISION*s and its length, we locate the potential collisions by performing the exact process described in Section 2.2. Listing 3.5 shows an implementation of *locateCollision*.

```
1  __global__ void locateCollision(DPCOLLISION* d_collisions, int num_cols, VALUECOLLISION
       * results, unsigned long functionVersion, GOLDENCOLLISION gc) {
2      int idx = blockIdx.x * blockDim.x + threadIdx.x;
3  
4      //only the case if we found less collisions than we have threads
5      if (idx >= num_cols) {
6          return;
7      }
8  
9      DPCOLLISION col = d_collisions[idx];
10 
11     //these will store the two values making up the collision
12     BYTE colliding_value1[SIZE];
13     BYTE colliding_value2[SIZE];
14     //these will store the current values we are at while walking along the paths
15     BYTE curr_value1[SIZE];
16     BYTE curr_value2[SIZE];
17 
18     //copy the starting values into the variables
19     for (int cp = 0; cp < SIZE; cp++) {
20         curr_value1[cp] = col.a.start[cp];
21         curr_value2[cp] = col.b.start[cp];
22     }
23 
24     //walk along "longer" path until the pathlengths are equal
25     while (col.a.steps < col.b.steps) {
26         f(curr_value2, curr_value2, functionVersion, gc);
27         col.b.steps -= 1;
28     }
29 
30     //Robin hoods: if we have equally many steps left and the current points match, its
        not a real collision
31     if (equal(curr_value1, curr_value2)) {
32         for (int cp = 0; cp < SIZE; cp++) {
33             results[idx].a[cp] = 0;
34             results[idx].b[cp] = 0;
35         }
36         return;
37     }
38 
39     //resume by walking along both paths step by step until we find the collision
40     while (notEqual(curr_value1, curr_value2) && col.a.steps > 0) {
41         for (int cp = 0; cp < SIZE; cp++) {
42             colliding_value1[cp] = curr_value2[cp];
43             colliding_value2[cp] = curr_value1[cp];
44         }
45         f(curr_value2, curr_value2,functionVersion, gc);
46         f(curr_value1, curr_value1,functionVersion, gc);
47         col.a.steps -= 1;
48     }
49 
50     //no collision occured.
51     if (notEqual(curr_value1, curr_value2)) {
52         for (int cp = 0; cp < SIZE; cp++) {
53             results[idx].a[cp] = 0;
54             results[idx].b[cp] = 0;
55         }
56         return;
57     }
58 
59     //copy the found collision into global memory
60     for (int cp = 0; cp < SIZE; cp++) {
61         results[idx].a[cp] = colliding_value1[cp];
62         results[idx].b[cp] = colliding_value2[cp];
```

```
63      }
64  }
```

Listing 3.5: Implementation of the *locateCollision* kernel

In lines 5-7, we need to check if the index of a thread is out of bounds for the array. This can only occur during the last execution of the kernel for a given function version since, for all other iterations, we wait to invoke the *locateCollision* kernel until enough collisions are found. In lines 25-28, we ”walk“ along the longer path until both path lengths are equal. The way *writeBackPoints* in Listing 3.1 is implemented, the *a* field of a *DPCOLLISION* instance always contains the longer path, hence we have the loop condition $col.a.steps < col.b.steps$. In lines 31-37, we check whether a ”Robin Hood“occurs. In that case, we return two equal values (in our case 0) such that the CPU in Listing 3.1 in line 81 can detect that it is not a valid collision. In lines 39-64, we simultaneously ”walk“ along both paths until the collision is found, which we then write into the *results* array. Lines 51-57 are necessary in order to detect ”false collisions“ (see Section 3.7).

3.7 Distinguishedness Property

Any easily verifiable feature could be used as a distinguishedness property. For example, we could require a distinguished point to have a certain number of leading zeros (depending on θ). In our implementation, we use a similar property: A point is said to be distinguished if the first *DIST_BITS* bits are equal to *DIST_BITS* specifically chosen bits of the current function version. The implementation of the *distinguished* function shown in Listing 3.4 is given in Listing 3.6.

```
1  __device__ int distinguished(BYTE* point, BYTE*
       functionVersion) {
2      //check that the required number of bytes are equal
3      for (int i = 0; i < DIST_BITS / 8; i++) {
4          if (point[i] != functionVersion[i]) {
5              return 0;
6          }
7      }
8      //check that the remaining number of bits are equal
9      int j = DIST_BITS % 8;
10     BYTE p = point[DIST_BITS / 8] >> (8-j);
11     BYTE f = functionVersion[DIST_BITS / 8] >> (8-j);
12
13     return p == f;
14 }
```

Listing 3.6: Implementation of the *distinguished* function

In lines 3-7, we check whether the first $\lfloor DIST_BITS/8 \rfloor$ bytes are equal, and in lines 9-11 we check whether the higher $DIST_BITS\%8$ bits of the $\lfloor DIST_BITS/8 \rfloor$-th byte are equal. If both conditions hold, we return 1. We make the property dependent on the function version for two reasons. First, we avoid the "bad case" where one or both of the preimages of the golden collision are distinguished, as each function version has a different set of distinguished points. Second, we do not need to zero out our memory between function versions, as, with a very high probability, the points stored in memory from the previous function version will not be distinguished anymore. Hence, they will be replaced in time. Even if a stored point is distinguished for two consecutive function versions, in the worst-case scenario, we waste a bit of computation trying to locate a collision that is not there. For this reason, we require lines 51-57 in Listing 3.5.

Note that $functionVersion$ is only 32 bits long. For $DIST_BITS > 32$ this approach can still work by, for example, by defining $functionVersion64 = functionVersion||functionVersion$, where $||$ denotes concatenation, and then use $functionVersion64$ for comparison.

3.8 Core Synchronization

When implementing vOW for CPU, synchronization across cores, such that each core runs the algorithm using the same function version, is non-trivial. Costello et al. mention synchronization strategies in their paper [8, Appendix C]. On the scale of a single GPU, synchronization is trivial. Threads do not run the entire algorithm independently. Instead, the execution is split up into multiple kernel calls. Therefore, the threads are constantly "synchronized". When attempting to implement this algorithm in a distributed fashion, i.e., on a scale of multiple GPUS (in possibly multiple different geographical locations), the synchronization and memory access issues are identical for CPUs or GPUs. However, a distributed implementation of vOW is out of the scope of this thesis.

3.9 Evaluation

In this section, we will evaluate the performance of our implementation. The GPU used to run our algorithm is the GeForce GTX 1080 Ti by NVIDIA. It has $3072 = 3 \cdot 2^{10}$ CUDA cores, i.e., it supports a maximum of 3072 threads executing simultaneously. The GPU has a clock frequency of 1480 MHz. The CPU used to run our algorithm (and in Section 3.10 to compare the performance with a CPU implementation) is the Intel Core i7-4790K CPU. It has a clock frequency of 4 GHz.

n	w				
	2^8	2^{10}	2^{12}	2^{14}	2^{16}
2^{18}	2.41	2.46	3.32	2.96	–
2^{20}	2.43	2.15	2.43	3.19	3.12
2^{22}	2.52	2.63	2.55	2.47	3.31
2^{24}	2.71*	2.17	2.53	2.51	2.61
2^{26}	–	2.81*	2.18	2.49	2.6

Table 3.1: Run-time results of running the vOW algorithm

Recall from Section 2.2 that the expected run-time to find the golden collision using the vOW algorithm can be estimated by:

$$\frac{2.5t}{m}\sqrt{\frac{n^3}{w}} \tag{3.1}$$

where t denotes the time required for an iteration of f, and m denotes the number of cores used. van Oorschot and Wiener heuristically found the constant factor 2.5 by running the full attack for multiple values of n and w while counting the number of function iterations (calls to f) required to find the golden collision. They created a table where each entry is the coefficient of $\sqrt{n^3/w}$ in the number of function iterations required (see Table 1 in [30]).

We would like to replicate part of that table to verify that our implementation follows the same run-time formula. We ran the full attack for multiple values of n and w and counted the number of required iterations of f. For each pair of values n and w, we ran the attack 200 times to get an average. Table 3.1 shows the replicated table. Entries marked with an asterisk denote that the entry was averaged over 100 runs instead of 200 runs due to the amount of computation (and time) required. Pairs of values n and w which would lead to $\theta = 2.25 \cdot \sqrt{w/n} > 1$ have an empty entry, denoted by "$-$".

We can see that our implementation follows the same run-time formula (at least for the chosen values of n and w), as we can approximately replicate the values shown in Table 1 of [30]. van Oorschot and Wiener mention that there is minimal variance across different values of n for each particular value of w. For us, this is not the case. Entries, where w isn't much smaller than n (i.e. for $w = 2^{16}$ and $n = 2^{20}$), seem to be higher (≈ 3), although there doesn't seem to be an obvious explanation for this. Luckily in a realistic scenario, w is orders of magnitude smaller than n.

As mentioned in Section 2.2, the vOW algorithm parallelizes perfectly in a theoretical sense. We now analyze the parallelizability of our implementation. We run the vOW algorithm for one function version for certain values of n and w while varying the number of cores utilized. Figure 3.7 shows the results of running the attack using $w = 2^{10}$ and $n = 2^{20}$ (red line), $n = 2^{26}$ (blue line), or $n = 2^{30}$ (yellow line). Note that the plot uses logarithmic scales on both axes (also called a log-log plot). In a log-log plot, perfect parallelizability is represented by a straight line. When looking at Figure 3.7, we see that our implementation parallelizes well, as the lines are almost straight. The more cores we use, the more the line starts to "flatten out". This behavior is to be expected since when utilizing thousands of cores, things like memory access and copies become bottlenecks. Note that for larger n, the line appears to be "more straight", i.e., it parallelizes better (the yellow line is noticeably straighter than the red line). This is because for large n, the CPU overhead (all the work the CPU does) is significantly smaller compared to the work the GPU does. When n is small, the GPU does not do much work (it is easy to find distinguished points and locate collisions for small n). However, for small n, and when utilizing many cores, the CPU must perform many memory copies and writebacks. Therefore the CPU-overhead dominates, and we experience less performance gain when using more cores for small n. Luckily, in any practically relevant scenario, n is large.

3.10 Comparison with a CPU Implementation

The goal of this section is to compare the performance of our vOW implementation with the performance of the CPU implementation by Longa et al. [18] to see if our implementation is viable. We first run one function version of the vOW algorithm for multiple values of n and w using a single CPU core running the CPU implementation. We then do the same using a single CPU core and a GPU with 3072 CUDA cores. We use the same CPU and GPU as described in Section 3.9. Table 3.2 shows the results. Each entry represents how much faster the GPU implementation is compared to the CPU implementation. The cells are color-coded in order to make the patterns more visible.

We get the smallest speedup for n small and w large (upper right corner of Table 3.2). This holds for the same reason as explained in Section 3.9: When n is small, and w is relatively large, the CPU overhead dominates (copying memory, writing back points, setting up kernels). In a realistic scenario, n is much larger than w. Hence, the CPU overhead should be negligible for practical values of n and w. Additionally, we can see that the speedup is also low for very small w (in our case, $w = 2^8$). This is because of how we implemented vOW. As we are running the *findDistinguishedPoint* kernel

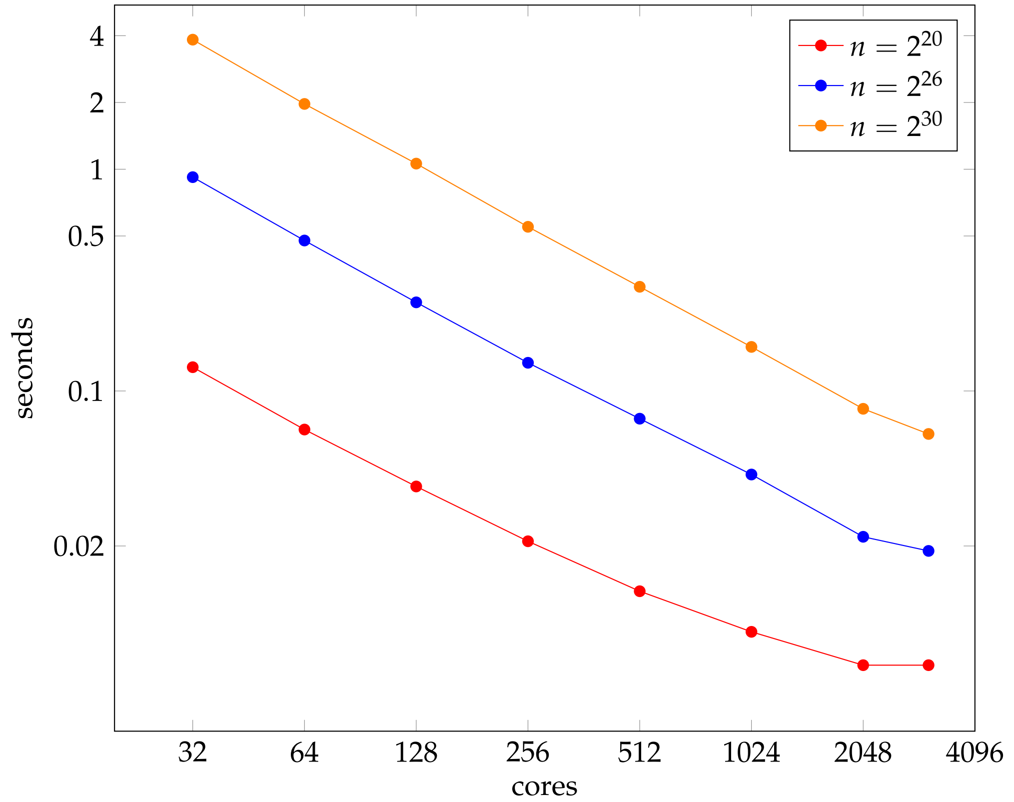

Figure 3.7: Plot showing the parallelizability of our implementation. The x-axis denotes the number of CUDA cores utilized. The y-axis shows the time required to complete one function version. (with $w = 2^{10}$).

n	w					
	2^8	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}
2^{18}	11.5	6.6	3.4	1.7	1.3	–
2^{22}	10.6	12.9	9.0	5.2	2.7	1.6
2^{26}	13.8	22.0	21.2	14.4	8.2	4.6
2^{30}	16.5	26.6	26.2	26.6	19.9	13.0
2^{34}	16.6	25.5	28.5	31.1	28.8	24.0
2^{38}	16.0	27.4	34.4	32.4	32.3	31.2
2^{42}	18.6	28.1	32.0	32.4	32.7	32.6

Table 3.2: Performance comparison of CPU and GPU implementation of the vOW algorithm

with 3072 threads, for each call to *findDistinguishedPoint* we locate approximately 1900 distinguished points[1]. For $w = 2^8$ we are only supposed to find $10w = 2560$ distinguished points. When using 3072 threads, however, we can only find a multiple of 1900 distinguished points per function version, i.e., we find approximately 3800 instead of 2560 distinguished points per function version. As this is way more than the optimal amount of $10w$ as discovered by van Oorschot and Wiener [30], the algorithm has a non-optimal run time, and therefore we experience a smaller speedup. This phenomenon only occurs for very small w (when w is about the same size or smaller than the number of threads per GPU). In practice, w will be much larger. We get the biggest speedup of all measured values for greater values of n and $n \gg w$ (lower right corner of Table 3.2). The maximal speedup we observe lies around 32, i.e., our GPU implementation executing on 3072 CUDA cores is 32 times faster than a CPU implementation running on a single core. The speedup might be even better for practical values of n and w, as the speedup seems to increase for greater values.

There are several reasons why we ″only“ experience a speedup of about 32 (not 3072). As we have seen in Figure 3.7, our implementation does not parallelize perfectly. Therefore a speedup of 3072 is unrealistic. We compared our implementation with the run-time of a single CPU core. Trivially, when running the CPU implementation on a single CPU core, we do not experience any synchronization or memory access issues that could diminish the performance. It is, therefore, not fair to say that 32 CPU cores would perform as well as 3072 GPU threads, as the CPU implementation might also not parallelize perfectly. Warp divergence within the GPU threads (due to the branch-rich vOW algorithm) also limits the speedup factor. Also, it would be unrealistic to assume that a single GPU thread running on a CUDA core performs similarly to a CPU thread running on a CPU core. GPU threads are lightweight and have fewer resources per core, as mentioned in Section 2.3. Finally, the vOW algorithm is not typical code that runs on GPUs. Some amount of divergence is unavoidable, and coalesced reads/writes are not always possible (due to branches and memory access patterns). Also, some GPU features are not used in our implementation. For example, we do not use shared memory at all, as GPU threads do not share any data during execution (whether they are in the same block or not).

Considering all the above reasons, it makes sense that we ″only“ get a speedup of 32 instead of 3072. Of course, our implementation is not perfect. Perhaps a much higher speedup can be achieved by using better-optimized code or maybe even a completely different approach to implementing vOW for GPUs.

[1] the value 1900 was taken from measurements and is independent from the value of n or w. It could be shown stochastically, which we will not do here.

To see if our implementation is financially viable, we compare the equipment cost for both the GPU and the CPU implementation. For the prior, we require a GPU, in our case the GeForce GTX 1080 Ti by NVIDIA, which has a MSRP (Manufacturer's suggested retail price) of $699 and a CPU (technically only a single core), in our case the Intel Core i7-4790K, which has an MSRP of $350. Hence, we get a total cost of $1049. For the latter, we require 32 CPU cores (assuming perfect parallelizability). As each Intel Core i7-4790K has 4 cores, we require 8 of them, reaching a cost of $2800. In this specific scenario, using the GPU implementation would make more sense financially. The CPU and GPU used in this cost analysis were released in early 2014 and early 2017, respectively. As the CPU is 3 years older, one might think that this is not a fair comparison. In recent years, CPUs did not get much faster. A benchmark comparison [29] of our CPU with the Intel Core i7-7700K CPU, released in early 2017, shows that there was only an increase in speed of about 7%, i.e., our cost analysis is still somewhat accurate, even with the difference of the release dates. A caveat of the cost analysis is that MSRPs might not accurately represent the actual retail price.

3.11 Possible Improvements

As mentioned in the previous section, our implementation is imperfect. There are plenty of other design choices that one could have come up with and multiple approaches to further optimize our implementation. Therefore, we would like to mention some potential improvements in this section.

3.11.1 Improve *locateCollision* Kernel

The *locateCollision* kernel suffers from the same issue the *findDistinguishedPoint* kernel did before we split it up into multiple iterations: Some collisions are found faster than others, as the number of steps to perform in order to locate the collision differs. This leads to the same problem of threads being idle after finding "their" collision. We could split up the process into multiple iterations using a similar approach as with the *findDistingiushedPoint* kernel. In each iteration, we only perform a specific number of steps. If we have not located the collision by then, we store the current points we are at and resume in the next iteration. Implementing this change would be a bit harder for the *locateCollision* kernel, as we would not only have to store "not yet found" collisions but also somehow identify which threads found a collision in the last iteration and which threads did not. We then only assign to the successful threads a new collision to locate. There are also other details one would need to consider, which we will not discuss here.

3.11.2 Tailor Implementation to Specific Parameters

In a practical scenario, where we would want to apply our implementation of the vOW algorithm to a problem at hand, the parameters n and w (and therefore θ) are usually fixed. The function f of which we want to find the golden collision has a fixed domain and co-domain size, and our available amount of memory w is also known and fixed. In this case, we could optimize the data types and structures we use in our implementation to use minimal space. To give a simple example: If $n = 2^{41}$, we could use the last 7 bits of a stored value (which uses 6 bytes, as $\lceil 41/8 \rceil = 6$) for the counter and the *valid* bit. Additionally, we could experiment with different values for α, β, γ, or θ, as for fixed n and w (and when running the attack on GPU instead of CPU), the values specified by van Oorschot and Wiener [30] might be non-optimal.

3.11.3 Approach Using Streams and Events

In the final subsection of this chapter, we briefly discuss an alternative way of implementing the vOW algorithm for GPUs. The key difference in this approach is that we dedicate a certain number of CUDA cores to finding distinguished points and a certain number of CUDA cores to locating collisions. This creates a pipeline as illustrated in Figure 3.8. Note that only part of the CUDA cores execute stage 2 of the pipeline, and the other part executes stage 4. Also, note that this implementation requires at least 3 CPU threads rather than just one.

CUDA streams and CUDA events, as discussed in Section 2.3, can be used to implement such a pipeline. Both kernels would need to be executed on separate non-default streams, and events would be used to communicate when kernels have finished executing. Careful evaluation is needed to determine how to split the available resources among the second and fourth pipeline stage such that they have approximately equal length and we can avoid pipeline stalling. It is uncertain how such an implementation would perform compared to ours.

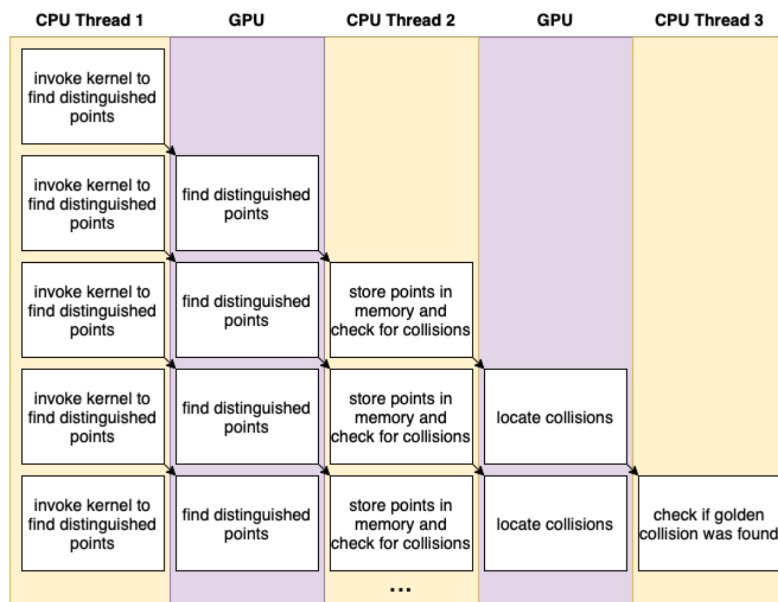

Figure 3.8: Implementing the vOW algorithm using a pipeline of depth 5.

Chapter 4

SIKE vOW Implementation

For this chapter, our goal was to use our implementation of the vOW algorithm to run an attack on SIKE using GPUs. This would require us to implement most of the functionalities used in the SIDH protocol. This includes functions performing basic operations in finite extension fields, such as addition or multiplication. It includes functions that calculate 2-isogenies, 4-isogenies, or the j-invariant of a given elliptic curve. Also, multiple data types and structures would be required to represent and store elliptic curves or j-invariants efficiently. Implementing all this from scratch would be a tremendous effort beyond this thesis time frame. Therefore, our goal was to find an already existing implementation of SIDH, which we could profit from to implement the attack. There already exists an implementation of SIKE on GPUs [26]. However, no source code was made available. The Microsoft Research repository for optimized implementations of SIDH [17] contains architecture-specific implementations of SIDH for the AMD64 and ARM64 platforms. However, it also includes a portable version written in pure C. The code for the portable implementation of SIDH is written in a modular fashion. At the "bottom" are functions that perform basic arithmetics in $\mathbb{F}_p$. These are used to build functions that perform arithmetics in $\mathbb{F}_{p^2}$. Lastly, we have the top-layer functions that perform isogeny computations, convert between representations, or calculate j-invariants.

The plan was to "convert" the existing portable code into code that runs on GPUs such that we get a working attack on SIKE using our implementation of the vOW algorithm. In a second step, we would optimize the bottom-layer functions performing the modular field arithmetics for GPUs (for example, by using inline PTX assembly).

It turns out that translating "CPU code" into code that executes well on GPUs is not as easy as one might think. The converted but non-optimized code of the portable SIDH version ran slowly and improperly. Although the exact reasons are unknown, as we have not analyzed the issue in-depth, the

most probable reason for the bad performance is the immense computation done in the SIKE protocol. As mentioned in Section 2.1, SIKE puts a relatively high computational burden on its users, which also reflects on the attack. When attacking SIKE using vOW on GPUs, each thread must perform some of these computations, requiring many resources. GPU cores do not have the same quantity of designated resources as a CPU core. Therefore, executing the same code will likely lead to excessive register spilling and other performance-draining issues.

These issues do not imply that attacking SIKE using vOW on GPUs is impossible. They imply that one would have to implement such an attack with code written to be run on GPUs explicitly.

Rewriting a big part of the SIDH functionality could have fixed these issues, but due to limited time, we were unable to do so. We still decided to leave this chapter in the thesis to warn that, generally, one can not expect cryptographic code designed to run on consumer CPUs to run well on GPUs.

Chapter 5

Conclusion

We introduced the reader to SIKE, which is a KEM submitted to the post-quantum cryptography standardization process initiated by NIST and discussed the mathematical details underlying the SIDH protocol. We gave an overview of the GPU architecture and provided an introduction to CUDA, an API by NVIDIA, which allows for general-purpose programming on NVIDIAs GPUs. An in-depth explanation of the vOW algorithm was given, including a description of an attack on SIKE using vOW. The central part of the thesis was an implementation of the vOW algorithm for GPUs using CUDA and the programming language C. We discussed several implementation details and challenges. By measuring the performance of our implementation on a single GPU with 3072 cores, we discovered that our implementation is approximately 32 times faster than a CPU implementation of vOW running on a single CPU core. Although the speedup is limited, our cost analysis suggests it could be worth using GPUs to run the vOW algorithm. Possible improvements and problems with our code were discussed, and alternative design choices were presented. Finally, we attempted to attack SIKE using our vOW implementation for GPUs but failed to implement it in our first attempts and ran out of time to complete the implementation. In conclusion, using GPUs as a resource to solve cryptographic problems can be beneficial but requires building extensive knowledge of the underlying architecture and careful programming. Not all code that executes fine on CPUs will perform equally well on GPUs.

Glossary

AES Advanced Encryption Standard.
API Application Programming Interface.

CPU Central Processing Unit.
CSSI Computational SuperSingular Isogeny.
CUDA Compute Unified Device Architecture.

GHz Gigahertz.
GPU Graphics Processing Unit.
GUI Graphical User Interface.

IND-CCA Indistinguishability under adaptive Chosen-Ciphertext Attack.
IND-CPA Indistinguishability under Chosen-Plaintext Attack.

KEM Key Encapsulation Mechanism.

MHz Megahertz.
MITM Meet-In-The-Middle.
MSB Most Significant Bit.
MSRP Manufacturer's suggested retail price.

NIST National Institute of Standards and Techonology.

PKE Public Key Encryption.
PQC Post-Quantum Cryptography.
PTX Parallel Thread Execution.

SIDH Supersingular Isogeny Diffie-Hellman.
SIKE Supersingular Isogeny Key Encapsulation.
SM Streaming Multiprocessor.

vOW van Oorschot and Wiener.

XOF eXtendable-Output Function.

Bibliography

[1] Gora Adj, Daniel Cervantes-Vázquez, Jesús-Javier Chi-Domínguez, Alfred Menezes, and Francisco Rodríguez-Henríquez. On the cost of computing isogenies between supersingular elliptic curves. In Carlos Cid and Michael J. Jacobson Jr., editors, *Selected Areas in Cryptography – SAC 2018*, pages 322–343, Cham, 2019. Springer International Publishing.

[2] Gorjan Alagic, David A. Cooper, Quynh Dang, Thinh Dang, John M. Kelsey, Jacob Lichtinger, Yi-Kai Liu, Carl A. Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, Daniel Smith-Tone, and Daniel Apon. Status report on the third round of the nist post-quantum cryptography standardization process, 2022-07-05 04:07:00 2022.

[3] Jega Anish Dev. Bitcoin mining acceleration and performance quantification. In *2014 IEEE 27th Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 1–6, 2014.

[4] Wouter Castryck and Thomas Decru. An efficient key recovery attack on SIDH (preliminary version). Cryptology ePrint Archive, Paper 2022/975, 2022. https://eprint.iacr.org/2022/975.

[5] Andrew Childs, David Jao, and Vladimir Soukharev. Constructing elliptic curve isogenies in quantum subexponential time. *Journal of Mathematical Cryptology*, 8(1):1–29, 2014.

[6] Brad Conte. Basic implementations of standard cryptography algorithms. https://github.com/B-Con/crypto-algorithms, 2012.

[7] Craig Costello. Supersingular isogeny key exchange for beginners. Cryptology ePrint Archive, Paper 2019/1321, 2019. https://eprint.iacr.org/2019/1321.

[8] Craig Costello, Patrick Longa, Michael Naehrig, Joost Renes, and Fernando Virdia. Improved classical cryptanalysis of sike in practice. Cryptology ePrint Archive, Paper 2019/298, 2019. https://eprint.iacr.org/2019/298.

[9] Oliver Dudler. Generic implementation of vOW for GPUs. https://gitlab.com/o935/bachelorthesis/-/tree/main/OW2, 2022.

[10] T. Elgamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[11] Luca De Feo, David Jao, and Jérôme Plût. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. *Journal of Mathematical Cryptology*, 8(3):209–247, 2014.

[12] Elichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. *Journal of Cryptology*, 26(1):80–101, 2013.

[13] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. *Combinatorics, Probability, and Information Theory*, pages 207–298. Springer New York, New York, NY, 2014.

[14] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the fujisaki-okamoto transformation. Cryptology ePrint Archive, Paper 2017/604, 2017. https://eprint.iacr.org/2017/604.

[15] Samuel Jaques and John Schanck. *Quantum Cryptanalysis in the RAM Model: Claw-Finding Attacks on SIKE*, pages 32–61. 08 2019.

[16] Péter Kutas and Christophe Petit. Torsion point attacks on "sidh-like" cryptosystems. Cryptology ePrint Archive, Paper 2022/654, 2022. https://eprint.iacr.org/2022/654.

[17] Patrick Longa, Basil Hess, Geovandro Pereira, and Joost Renes. SIDH v3.5.1 (C Edition). https://github.com/microsoft/PQCrypto-SIDH, 2022.

[18] Patrick Longa, Michael Naehrig, Fernando Virdia, and Joost Renes. vOW4SIKE - Parallel Collision Search for the Cryptanalysis of SIKE. https://github.com/microsoft/vOW4SIKE, 2020.

[19] David Lubicz and Damien Robert. Computing isogenies between abelian varieties. *Compositio Mathematica*, 148(5):1483–1515, 2012.

[20] Svetlin A. Manavski. Cuda compatible gpu as an efficient hardware accelerator for aes cryptography. In *2007 IEEE International Conference on Signal Processing and Communications*, pages 65–68, 2007.

[21] Samuel Neves and Filipe Araujo. On the performance of gpu public-key cryptography. In *ASAP 2011 - 22nd IEEE International Conference on Application-specific Systems, Architectures and Processors*, pages 133–140, 2011.

[22] NVIDIA. Visual Profiler website. `https://developer.nvidia.com/nvidia-visual-profiler`. Accessed: 2022-07-17.

[23] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. CUDA, release: 10.2.89, 2020.

[24] Wuqiong Pan, Fangyu Zheng, Yuan Zhao, Wen-Tao Zhu, and Jiwu Jing. An efficient elliptic curve cryptography signature server with gpu acceleration. *IEEE Transactions on Information Forensics and Security*, 12(1):111–122, 2017.

[25] Microsoft Research. NIST submitted SIKE reference implementation. https://github.com/microsoft/PQCrypto-SIKE. Accessed: 2022-07-21.

[26] Seog Chung Seo. Sike on gpu: Accelerating supersingular isogeny-based key encapsulation mechanism on graphic processing units. *IEEE Access*, 9:116731–116744, 2021.

[27] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, oct 1997.

[28] Andrew V. Sutherland. Identifying supersingular elliptic curves. *LMS Journal of Computation and Mathematics*, 15:317–325, 2012.

[29] UserBenchmark. UserBenchmark. `https://cpu.userbenchmark.com/Compare/Intel-Core-i7-4790K-vs-Intel-Core-i7-7700K/2384vs3647`. Accessed: 2022-08-31.

[30] Paul C. van Oorschot and Michael J. Wiener. Parallel collision search with cryptanalytic applications. *J. Cryptol.*, 12(1):1–28, 1999.

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):

Authored by (in block letters):
For papers written by groups the names of all authors are required.

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

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)**

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