GPU Implementation Techniques for Differential Collision Search Attacks

A case study of SHA-1

Jürgen Windhaber

GPU Implementation Techniques for Differential Collision Search Attacks

A case study of SHA-1

Master's Thesis

at

Graz University of Technology

submitted by

Jürgen Windhaber

juergen.windhaber@gmail.com

Institute for Applied Information Processing and Communications (IAIK),
Graz University of Technology
Inffeldgasse 16a
A-8010 Graz, Austria

16th September 2011

Supervisor: Univ.-Prof. Dr. Vincent Rijmen
Advisor: Dr. Christian Rechberger

GPU Implemenationstechniken für differenzielle Kollisionssuchen

Eine Fallstudie an SHA-1

Masterarbeit

an der

Technischen Universität Graz

vorgelegt von

Jürgen Windhaber

Institut für Angewandte Informationsverarbeitung und Kommunikationstechnologie (IAIK),
Technische Universität Graz
Inffeldgasse 16a
A-8010 Graz

16. September 2011

Diese Arbeit ist in englischer Sprache verfasst.

Begutachter:	Univ.-Prof. Dr. Vincent Rijmen
Betreuender Assistent:	Dr. Christian Rechberger

Abstract

Shortcut collision attacks are an important field of operation in IT security. These attacks are that important because they not only show weaknesses in algorithms but in a further consequence they also give insight into how to make them more secure. One characteristic of shortcut collision attacks is that they often need a long time to compute due to their high computational complexity. New generation GPUs are offering an attractive way to reduce the computational time significantly.

This thesis evaluates how suitable GPUs are in general to solve that sort of problems. It also offers a set of techniques that show how to implement these kinds of algorithms efficiently on GPUs. As an illustration of these techniques computational intensive parts of a shortcut collision attack on SHA-1 are implemented as a case study. It shows that although GPUs cannot unfold their full potential on that sort of problems a significant speedup is achieved in comparison to conventional CPU implementations.

Kurzfassung

Kollisionsattacken stellen ein wichtiges Einsatzgebiet in der IT Sicherheit dar. Diese Attacken sind bedeutend weil sie einerseits Schwachstellen in Algorithmen aufzeigen und andererseits auch wichtige Erkenntnisse liefern um diese Algorithmen sicherer zu machen. Ein Merkmal von Kollisionsattacken ist, dass sie oftmals aufgrund ihrer hohen Komplexität eine lange Zeit zur Berechnung brauchen. Neue Generationen von GPUs stellen eine sehr attraktive Alternative dar, um die Berechnungsdauer dieser Probleme signifikant zu reduzieren.

Diese Arbeit versucht zu bewerten wie geeignet GPUs im generellen sind, um Probleme dieser Art zu lösen. Es werden verschiedene Techniken vorgestellt, die Lösungswege aufzeigen, um diese Art von Algorithmen effektiv auf GPUs zu berechnen. Um diese Techniken zu illustrieren wurden rechnerisch intensive Teile einer Kollisionsattacke auf SHA-1 als Fallstudie implementiert. Obwohl GPUs bei dieser Art von Problemen nicht ihr volles Potenzial ausspielen können, zeigt die Fallstudie doch deutlich, dass GPU Implementationen gegenüber herkömmlichen CPU Implementationen große Vorteile in Sachen Berechnungsdauer aufweisen.

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Place	Date	Signature

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Ort	Datum	Unterschrift

Contents

List of Figures

List of Tables

Acknowledgements

Writing this thesis was an endeavor with some stumbling blocks along the way. First, I'd like to thank my parents Erna and Karl for their support throughout my study. I also owe thanks to my girlfriend Jasmina for her support and motivation.

Also a big thank you goes to my advisor Christian Rechberger for his help and patience. I like to thank Vincent Rijmen for agreeing to be my supervisor and giving me the opportunity to write this thesis.

Last but not least I would like to thank all my friends and colleagues for all the interesting discussions about my thesis but also about all the other interesting things we debated.

Jürgen Windhaber
September, March 2011

Chapter 1

Introduction

Most people in the industrialized parts of the world are living in ”information-driven” societies. Information in general influences and affects them in their every-day life, and therefore it stands to reason that this information needs to be protected. Responsible for this protection in general is cryptography, which was already used in ancient times. Although in contrast to today almost only to hide information.

Today cryptography is used for a lot of different applications. One set of tools is called hash algorithms and is responsible for two important applications called message authentication and modification detection. It goes without saying that these mechanisms have to be ”secure”, and for hash algorithms an important security criterion is collision resistance. [MvOV96] A collision attack on these hash algorithms should be computational infeasible, but recently many of the most used Hash Algorithms such as SHA-1 [oST02] or MD5 [Riv92b] have been broken by a series of attacks. Although in the case of SHA-1 the complexity to find an actual collision is too high, existing collision attacks may lay the foundation for more powerful attacks.

New techniques in collision search attacks are pushing the complexities of these attacks into a direction where a computation of a collision seems possible. For example, the currently best known collision attack on SHA-1 has a complexity somewhere around 2^{60} which is too high to solve on a single standard computer. However it puts it in the range of different other approaches, such as distributed computing[1] or general-purpose computing on graphics processing units or short GPGPU. Collision attacks on cryptographic hash functions are very often expansive in a computational point of view. State of the art CPUs offer a lot of computational power, but the time required to find collisions on modern day hash functions is high anyhow.

An alternative to conventional CPUs for complex and time intensive computations is on the rise for a while now. This alternative is commonly known as GPGPU and uses a GPU or a combination of CPU and GPU to solve problems of general purpose. These GPUs are generally equipped with a high number of processing kernels and are capable to run thousands of threads in parallel. Not long ago the usage of these GPUs was somewhat limited because of the absence of an API which provides developers with tools to conveniently access the GPU. The only way to communicate with GPUs was trough graphical APIs, such as OpenGL [SGI92] or DirectX [Cor09].

In 2006 ATI presented a low level interface called CTM [AMD06] (Close to metal) to access the GPU directly. Competitor NVIDIA was not far behind and introduced CUDA [NVI09a] (Compute Unified Device Architecture) in February 2007, which provides developers with a C like API and is therefore

[1] A distributed computing system consists of a certain number of computers which are typically connected through a network.

arguable easier to use than CTM. For the first time these tools made it possible to use the computational power of modern day graphic cards directly to solve problems in a fraction of time compared to state of the art CPUs. Because of their often high complexity, collision attacks are suited very good to implement on GPGPUs.

One big problem though is the non-deterministic program flow in combination with unpredictable memory access patterns which is common in most collision attacks. Up till now and very probably also in the near future all the GPUs have a SIMT (Single Instruction Multiple Thread) architecture. This architecture is not suited to deal with non-deterministic program flows. There are different limitations which more or less all can be traced back to the SIMT architecture of the GPUs. For this purpose certain design strategies were developed to counter these limitations, or at least try to minimize the limiting factors.

A variation of the currently best attack of SHA-1 in view of complexity is used as a case study to show different approaches and to conquer the above mentioned problems. Also, the difference between an ”ideal” implementation without any limitations at all and the developed approaches was measured to see what would be possible if GPUs would have a more suitable architecture like MIMD (Multiple Instruction Multiple Data).

1.1 Terms and Acronyms

In this thesis the following terms and acronyms are used:

MDC	Message detection code
MAC	Message authentication code
NIST	National institute of standards and technology
NSA	National security agency
GPGPU	General-purpose computing on graphics processing units
SIMD	Single instruction multiple data
MIMD	Multiple instruction multiple data
SIMT	Single instruction multiple thread
Warp	A group of a number of threads
SM	Streaming multiprocessor
TPC	Thread processing cluster

Table 1.1: Terms

1.1.1 Symbols

In this thesis the following symbols are used:

$\oplus$	Bitwise XOR operation
$\wedge$	Bitwise AND operation
$\vee$	Bitwise OR operation
$\neg$	Bitwise complement operation
$<<$ or ROL_n	Bitwise rotation to the left in a n-bit variable

Table 1.2: symbols

Chapter 2

Hash Functions

Cryptographic hash functions provide an effective way to protect large quantities of information with the help of a short hash value generated by a hash algorithm. A hash function takes an input value of arbitrary length and computes an output value of fixed length, called hash value. [Pre94]

The applications for hash algorithms can be categorized into two main fields of application:

- Message authentication.
- Modification detection.

Message authentication codes (MACs) take as input a message and a secret key. The first application area of a message authentication code is data integrity. Data integrity assures that the data has not been modified by an unauthorized person or process. The second application area is data origin authentication. Data origin authentication is used to authenticate messages. [MvOV96, p. 323]

The second application is called modification detection code (MDC). Modification detection codes are used in many areas. Typical examples would be the creation of digital signatures, or checksums over downloaded files. [MvOV96, p. 323]

2.1 Requirements of Hash Functions

Cryptographic hash functions have to fulfill certain properties. It is important to mention that there is a distinction between security requirements and constructive requirements [MvOV96, pp. 323 - 324].

2.1.1 Constructive Requirements

- **easy to compute**
  The hash value y of a given input value x is easy to compute.
- **compression**
  The hash function computes a hash value y from a given input value x with an arbitrary length to a fixed length.

2.1.2 Security Requirements

- **preimage resistance**
  It must be computationally infeasible to find an input value x for a given hash value y, $h(x)=y$. In other words the hash function must be a one way function.
- **2nd preimage resistance**
  It must be computationally infeasible to find a second input value z for a given input value x that both compute the same hash value, $h(z)=h(x)$.
- **collision resistance**
  It must be computationally infeasible to find two arbitrary input values x and z which compute the same hash value, $h(z)=h(x)$.
- **near collision resistance**
  It should be hard to find any two input values x and z such that $h(x)$ and $h(z)$ only differ in a small number of bits.

There are also additional requirements in certain cases. For example in the case of the SHA-3 candidates which are discussed in detail in Section 2.2.4. Also requirements like memory consumption, code size and the simplicity of the algorithms structure are taken into account.

2.2 Iterated Hash Functions

Most cryptographic hash functions are iterated hash functions with a so called Merkle-Damgård [Mer79, pp. 13–15] [Mer90, Dam90] construction which is shown in Figure 2.1. The message is split into several blocks with fixed length. These blocks are computed separately. Each block serves as input to an internal function E.

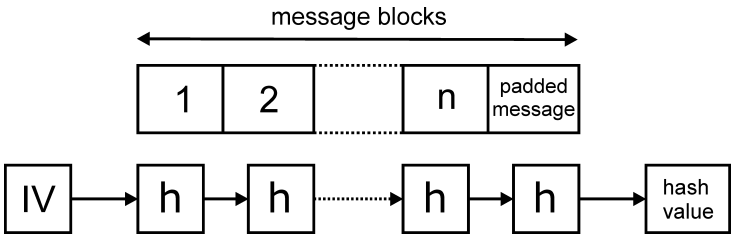

Figure 2.1: General structure of an iterated hash function as Merkle-Damgård construction

There are different methods to compute the compression function itself. The most prominent methods are:

Davies-Meyer mode is shown in Figure 2.2 and can be defined as $H_i = E_{m_i}(H_{i-1}) \oplus H_{i-1}$. [MvOV96, pp. 340 - 341]

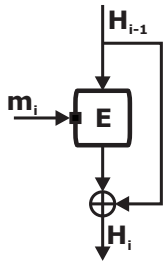

Figure 2.2: Davies-Meyer mode

Miyaguchi-Preneel mode is shown in Figure 2.3 and can be defined as $H_i = E_{g(H_{i-1})}(m_i) \oplus H_{i-1} \oplus m_i$. [MvOV96, pp. 340 - 341]

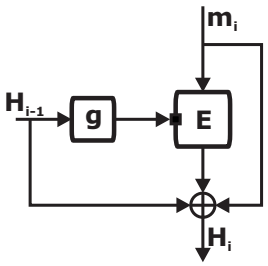

Figure 2.3: Miyaguchi-Preneel mode

Matyas-Meyer-Oseas mode is shown in Figure 2.4 and can be defined as $H_i = E_{g(H_{i-1})}(m_i) \oplus m_i$. [MvOV96, pp. 340 - 341]

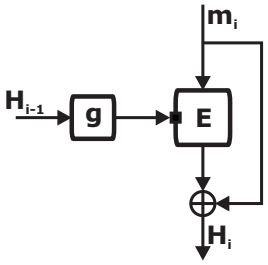

Figure 2.4: Matyas-Meyer-Oseas mode

On the last block a so called message padding is performed. For example the message padding of SHA-0 and SHA-1 is processed in the following way. The input values have a fixed length of 512-bits. First they are split into 512-bit blocks. At the end of the last message block a ”1” followed by ”0”s followed by a 64-bit integer is padded. The 64-bit integer value describes the full length of the message. Therefore the length of the input message is limited by 2^{64}. [oST02]

2.2.1 The MD4 Family

Members of the so called MD4 family are by far the most commonly used hash functions today. The MD4 [Riv92a] algorithm was published in the year 1990 by Ronald L. Rivest. MD4 had also a main influence on the development of MD5 [Riv92b], HAVAL [ZPS93] , RIPEMD [BDP97] and the SHA family which is described in Sections 2.2.2 and 2.2.3. Figure 2.5 shows the most prominent members of the MD4 Family.

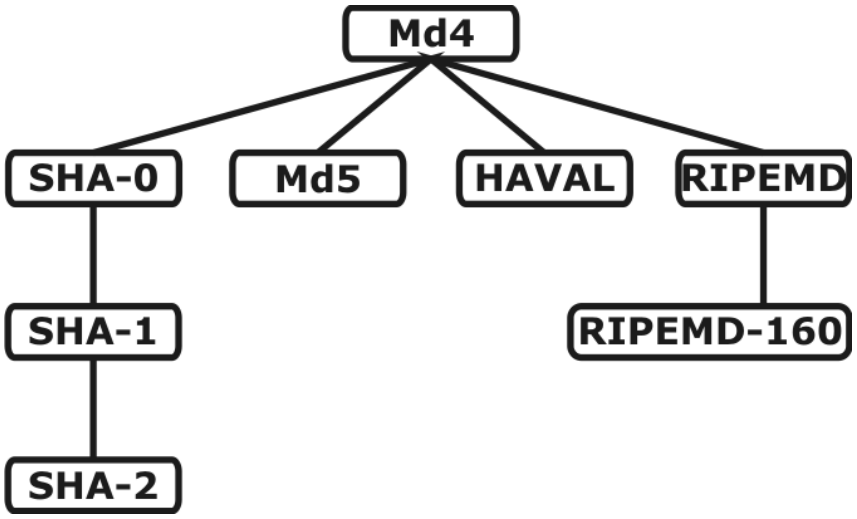

Figure 2.5: Members of the MD4 family

2.2.2 SHA-0 and SHA-1

The SHA family is a group of hash algorithms developed by the National Institute of Standards and Technology (NIST) and the National Security Agency (NSA). SHA-0 [Nat93] was published in the year 1993. Two years later the algorithm was replaced by SHA-1 [oST02]. SHA-0 and SHA-1 are iterated hash algorithms which process blocks of 512-bits and produce a hash value of 160 bit.

The core function of the hash algorithm itself consists of two parts:

- Message expansion
- State update

Message Expansion

In the message expansion the 512-bit message block is split into 16 32-bit segments ($M_0, M_1, M_2, ...M_{15}$) called message words. The function expands these 16 message words M_i to 80 message words W_i. The difference between SHA-0 and SHA-1 consists only of the left rotation in the message expansion as shown in Equation 2.2. [oST02]

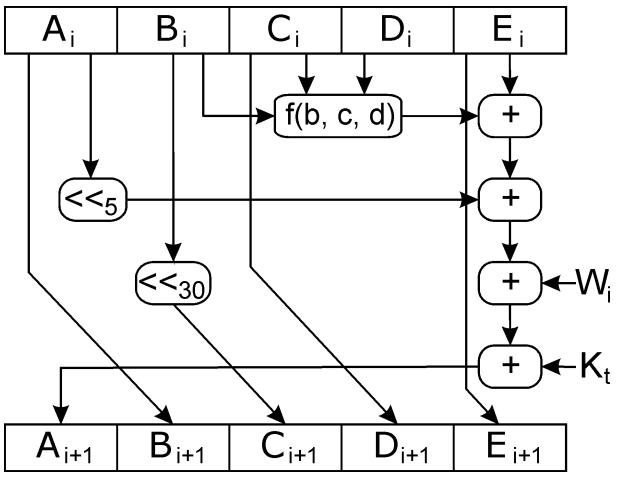

Figure 2.6: State update function of SHA-0 and SHA-1

For Sha-0:

$$\begin{aligned} &W_i = M_i,\ \forall i,\ 0 \leq i < 16 \\ &W_i = W_{i-3} \oplus W_{i-8} \oplus W_{i-14} \oplus W_{i-16},\ \forall i,\ 16 \leq i < 80 \end{aligned} \tag{2.1}$$

For Sha-1:

$$\begin{aligned} &W_i = M_i,\ \forall i,\ 0 \leq i < 16 \\ &W_i = ROL_1(W_{i-3} \oplus W_{i-8} \oplus W_{i-14} \oplus W_{i-16}),\ \forall i,\ 16 \leq i < 80 \end{aligned} \tag{2.2}$$

State Update Function

After the message expansion the expanded message W is used to update the so called chaining variables $(A_i, B_i, C_i, D_i, E_i)$ in 80 steps $(i = 0, 1, ..., 79)$.

$$\begin{aligned} A^{(i+1)} &= ADD(W^{(i)}, ROL_5(A^{(i)}, f^{(t)}(B^{(i)}, C^{(i)}, D^{(i)}), E^{(i)}, K^{(t)}) \\ B^{(i+1)} &= A^{(i)} \\ C^{(i+1)} &= ROL_{30}(B^{(i)}) \\ D^{(i+1)} &= C^{(i)} \\ E^{(i+1)} &= D^{(i)} \end{aligned} \tag{2.3}$$

The state update function has 80 steps and four different rounds which are shown in Table 2.1. Figure 2.6 shows how the chaining variables $(A_i, B_i, C_i, D_i, E_i)$ are updated at each step. The constant variables $K^{(t)}$ are added in every step and they are different for each round. The initial value IV $(A_i, B_i, C_i, D_i, E_i)$ is also a set of constants. After the four rounds chaining variables are added to the initialized value feed-forward. This process reiterates as long as all 512-bit blocks are computed. The final value is than called hash value or message digest. [oST02]

Table 2.1: The different rounds of the state update function

STEPS	ROUND	BOOLEAN	$f(B, C, D)$
1-20	1	IF	$(B \wedge C) \vee (\neg B \wedge D)$
21-40	2	XOR	$B \oplus C \oplus D$
41-60	3	MAJ	$(B \wedge C) \vee (B \wedge D) \vee (C \wedge D)$
61-80	4	XOR	$B \oplus C \oplus D$

2.2.3 SHA-2

SHA-2 [oST02] was introduced in the year 2002 and consists of SHA-256, SHA-384, SHA-512. These three algorithms have longer hash values as their predecessors. Also some implementation details are different to the SHA-0 and SHA-1 algorithms but the underlying design principles are similar. In Table 2.2, the maximum message size, the block size, the word size and the message digest size for all members of the SHA family are shown.

Table 2.2: Secure hash algorithm properties [oST02]

ALGORITHM	MESSAGE SIZE	BLOCK SIZE	WORD SIZE	MESSAGE DIGEST SIZE
SHA-0	$< 2^{64}$	512	32	160
SHA-1	$< 2^{64}$	512	32	160
SHA-256	$< 2^{64}$	512	32	256
SHA-384	$< 2^{128}$	1024	64	384
SHA-512	$< 2^{128}$	1024	64	512

2.2.4 SHA-3

In 2007 NIST issued a international competition, similar to the AES competition [FIP01, NBB+] in 1998, to develop a new secure hash standard until 2012 called SHA-3. The need for a new secure hash standard resulted from the fact that recently big advances in cryptanalysis of hash algorithms were made [Nat09b]. Chapter 3 will show different attacks on SHA-1 which created the need for a new secure hash standard. Sixty-four different algorithms were submitted to NIST in October 2008. 51 algorithms out of the 64 met the minimum acceptance criteria to be accepted as first round candidates. In comparison to the AES competition with 21 submissions [RD99] this is a huge advancement, at least in view of quantity.

For round two, NIST announced 14 candidates:

- BLAKE [AHMP08]
- Blue Midnight Wish [GKK+08]
- CubeHash [Ber08]
- ECHO [BBG+08]
- Fugue [HHJ08]
- Grøstl [GKM+08]

- Hamsi [zK08]
- JH [Wu08]
- Keccak [BDPA08]
- Luffa [CSW08]
- Shabal [BCCM+08]
- SHAvite-3 [BD08]
- SIMD [LBF08]
- Skein [FLS+08].

NIST announced 5 candidates for round three:

- BLAKE [AHMP08]
- Grøstl [GKM+08]
- JH [Wu08]
- Keccak [BDPA08]
- Skein [FLS+08].

The email announcement states that none of the second round candidates were clearly broken. One criterion is also the performance of the algorithms. Also an important criterion is that all the finalists display a clear round structure. [oST10]

Table 2.3 shows important milestones for the hash algorithm competition:

Table 2.3: Milestones of the preliminary timeline for the SHA-3 hash algorithm competition [oST02]

YEAR, QUARTER	DESCRIPTION
2008 4Q	Submission deadline for new hash functions
2009 2Q	Announcing first round candidates
2010 3Q	Announcing the finalists
2012 2Q	Address public comments, and select the winner

A detailed classification of all candidates can be found in [FFG09].

Chapter 3

Collision Attacks

3.1 Generic Attacks

Generic attacks are not dependent on design principles of the attacked algorithms. These attacks work at the basis of mathematical rules and can be applied to all algorithms and functions [MvOV96, p. 369]. In the following two sections two generic attacks are described in detail.

3.1.1 Brute Force

Brute Force is a way to break an algorithm by initializing an exhaustive search. For a successful collision attack on SHA-0 the complexity is 2^{160} because only 2^{160} different hash values exist. Because of the requirements of hash functions described in Section 2.1 it must be computationally infeasible to accomplish such an exhaustive search.

3.1.2 Birthday Attack

The birthday paradox has its name from the observation that if there are 23 people in a room then there is a probability of more than 50% that at least two of them have the same birthday. The mathematical principal behind the birthday paradox can be used to create a collision attack called birthday attack. For the birthday attack only the output length of the hash value is of importance, so it can be applied to every hash function in the same manner regardless of the hash functions internal structure.

One of the first applications of the birthday attack, and probably the most well-known, was published by Yuval[Yuv79]. The main observation is that if one is randomly drawing members from a set with n elements the probability is very high that the same element will be picked twice after $O(\sqrt{n})$ attempts.

This means, that for collision attacks on hash algorithms with a message digest size of n a collision can be found with a complexity of $2^{n/2}$. For example in the case of SHA-0 and SHA-1 this results in a complexity of 2^{80}. [MvOV96, p. 369]

Algorithm for the birthday attack:

- Pick a message x (preferable a meaningful message)
- compute $y = h(x)$ and safe y in an array
- alter x to x' and compute $y' = h(x')$
- compare y' with all elements in the array
- if no match is found put y' into the array and compute new y'

3.2 Attacks on the MD4 Family

Another view on the MD4 family tree shown in Figure 3.1 depicts that most members are already broken.

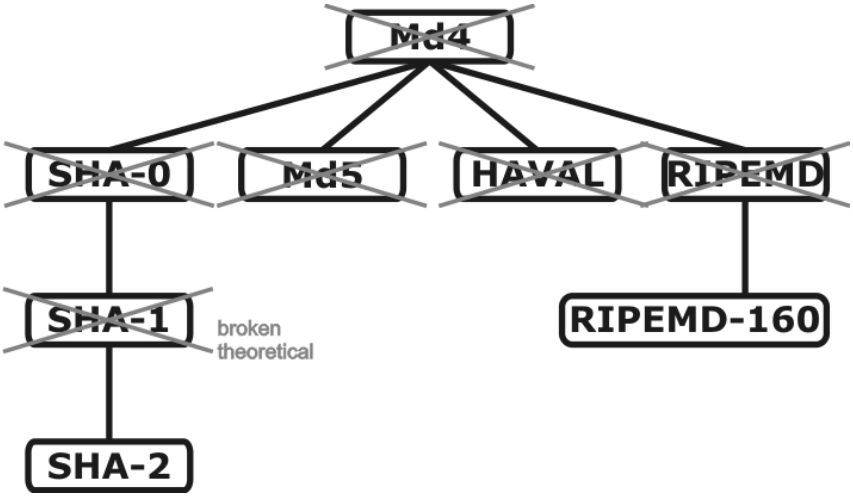

Figure 3.1: Members of the MD4 family which are broken are marked with an X

- MD4 broken in 1996 [Dob96]
- MD5 broken in 2004 [WFLY04]
- HAVAL-128 broken in 2004 [WFLY04]
- RIPEMD broken in 2004 [WFLY04]
- SHA-0 broken in 2004 [BCJ+05] (Paper published 2005)
- SHA-1 theoretically broken in 2005 [WYY05b]

3.3 Cryptanalysis of SHA-0 and SHA-1

Collision attacks on the SHA family have a long history. SHA-0 and SHA-1 are probably the most attacked and assessed hash algorithms at all. Due to the fact that SHA-0 has a simpler structure in the message expansion compared to SHA-1, SHA-0 was the first choice of cryptographers to attack. The first theoretically successful attack on SHA-0 was published 1998, five years after its introduction, and is described in Section 3.3.1. In 2004 the first near collisions in SHA-0 were found which are described

in Section 3.3.2. After extending previous attacks described in Section 3.3.3, the big breakthrough was in 2005 when the first collision on SHA-0 was found and also SHA-1 was theoretically broken. Section 3.3.4 describes the attack on SHA-0, and Section 3.3.5 describes the attack on SHA-1. The last and most promising attack in view of finding a real collision in SHA-1 is described in Section 3.3.7 and is also the subject of the case study discussed in Section 6.

3.3.1 Differential Collisions in SHA-0 (Chabaud Joux)

Chabaud and Joux presented the first collision attack on SHA-0 [CJ98]. Three algorithms were established called SHI1, SHI2 and SHI3. Each one of these three algorithms has marked similarities to SHA-0 but each of these three algorithms simplifies several implementation details. From that it follows that each one of these three algorithms has another run-time leading to a collision as shown in Table 3.1.

Table 3.1: Algorithms and their complexity

NAME	COLLISION COMPLEXITY	DESCRIPTION
SHI1	128	ADD function is replaced by XOR, $f_{(if)}$ and $f_{(maj)}$ are replaced by $f_{(xor)}$
SHI2	2^{20}	ADD function is replaced by XOR
SHI3	2^{44}	$f_{(if)}$ and $f_{(maj)}$ are replaced by $f_{(xor)}$
SHA-0	2^{61}	

To fully understand the attack of Chabaud and Joux some terms have to be defined first:

- **local collisions**
  A local collision is a collision consisting of six steps. It is a fact that SHA-0 has local collisions that can be started at any step. These local collisions are responsible for creating full collisions of the algorithm.
- **differential path**
  A differential path describes the differences between two variables. The differential path is used in the attack to describe local collisions. For a full 80-step collision the differential path describes several local collisions that lead to a full collision. It is possible that these paths which describe the local collisions have overlaps in some cases. Table 3.2 shows a differential path for 6 steps that leads to a local collision.
- **disturbance vector**
  A disturbance vector is a vector where the bits are set if a local collision is starting at that point. For the case of SHA-0 it is sufficient to use an 80 bit vector to mark the starting points of the local collisions. Another observation is that the first 16 variables determine the remaining 64. From that it follows that there are only 16 bit to choose freely which leads to 2^{16} different disturbance vectors in total. The complexity of the collision search is dependent on the Hamming weight[1] of the disturbance vector.

[1] The Hamming weight describes the number of bits which are set.

Table 3.2: A 6-step local collision in SHA-0 / SHA-1 [CJ98]

	ΔM	ΔA	ΔB	ΔC	ΔD	ΔE
i	2^j	2^j	0	0	0	0
i+1	2^{j+5}	0	2^j	0	0	0
i+2	2^j	0	0	2^{j+30}	0	0
i+3	2^{j+30}	0	0	0	2^{j+30}	0
i+4	2^{j+30}	0	0	0	0	2^{j+30}
i+5	2^{j+30}	0	0	0	0	0

Basic Attack Strategy

Chabaud and Joux defined three algorithms. Each of them similar to SHA-0, but several nonlinear parts were replaced with linear parts. Therefore it is easier to produce collisions in these algorithms than in the real SHA-0. In the final collision attack Chabaud and Joux attempt that the defined algorithms "play together" which is shown in Section 3.3.1.

SHI1

Here Chabaud and Joux took the compression function from SHA-0 with two changes. First the ADD function is replaced by XOR. Second the $f_{(if)}$ and the $f_{(maj)}$ functions are replaced by the $f_{(xor)}$ function.

$$A^{(i+1)} = XOR(W^{(i)}, ROL_5(A^{(i)}, f^{(t)}(B^{(i)}, C^{(i)}, D^{(i)}), E^{(i)}, K^{(i)}))$$

In the compression function only the ADD function, the $f_{(if)}$ and the $f_{(maj)}$ functions are not linear. With these changes the whole compression function is a linear function. At this time they lose sight of the fact that the vector W is computed by the message expansion. Only with this premises it is possible to assume that it is allowed to change every bit on W. The fact that everything in this model is linear makes it possible to get a differential path which leads very easy to a full collision. Now a disturbance vector of 80 entries is build. Every "1" in the vector signifies that on this position $W_j^{(i)}$ is negated. This works only for the first 75 steps because it is not possible to correct disturbances after step 80. In the case that the five zeros at the end on the error vector must hold, there are only 128 possible inputs left.

SHI2

Compared to SHI1, $f_{(if)}$ and $f_{(maj)}$ remain unaffected. Only the ADD function is changed like in SHI1. It can be shown that in some cases $f_{(if)}$ and $f_{(maj)}$ compute the same outputs as the $f_{(xor)}$ function which is used in SHI1. The goal is to change the inputs so that $f_{(if)}$ and $f_{(maj)}$ act as the $f_{(xor)}$ from SHI1, in other words the functions $f_{(if)}$ and $f_{(maj)}$ behave like $f_{(xor)}$. These changes raise the complexity because $f_{(if)}$ and $f_{(maj)}$ behave like a $f_{(xor)}$ with a certain probability. With all these new conditions it is possible to find a full collision on SHI2 with a complexity of about 2^{20}.

SHI3

Once again the compression function from SHA-0 is taken. $f_{(if)}$ and $f_{(maj)}$ are replaced by $f_{(xor)}$. The ADD function stays unchanged. The problem here is that a carry effect can appear in the disturbances.

The goal is to prohibit this carry effect. A helpful observation is that no carry can appear at bit position 31. Another observation is that if a bit flips from 0 to 1 it must be corrected with a flip from 1 to 0 (a bit-flip from 1 to 0 acts equal). With these changes it is possible to find a full collision on SHI3 with a complexity of 2^{44}.

SHA-0

Here the procedure to prevent a carry in the SHI3 function and the procedure to simulate the ADD function in SHI2 must play together. With these changes it is possible to find a full collision on SHA-0 with a complexity of 2^{61}. The achieved complexity of 2^{61} is better than the results achieved by a birthday attack, which is described in Section 3.1.2.

3.3.2 The First Near Collisions of SHA-0

Eli Biham and Rafi Chen published a collision attack [BC04] on SHA-0 which is based on the fact that many bits of a given message are so called neutral bits. The attack is based on the attack of Chabaud and Joux [CJ98] which is described in Section 3.3.1. Biham and Chen present a 65-round collision with a complexity of 2^{29} and a near collision of the full 80-round function where only 18 bits differ with a complexity of 2^{40}. Furthermore, they present an attack on an 82-round SHA-0 which has a complexity of 2^{43} and a full 80-round attack with a complexity of 2^{56}.

Definition of Neutral Bits

To explain the work of Biham and Chen the following definitions are needed:

- δ_r
  δ describes the expected differences in the chaining variable A in each round based on the difference of M and M' which is called Δ. A pair of messages confirms to δ_r if $A_i \oplus A'_i = \delta_i$ for every $i \in \{1, ..., r\}$.

- **neutral bit**
  The bit on position $i \in \{0, ...511\}$ is a neutral bit in relation to M and M' if it conforms to δ_r before and after flipping the bit in M and M' on position i.

- **pair of neutral bits**
  The pair of bits i and j are a neutral pair of bits in relation to M and M' if they conform to δ_r before and after flipping any subset of the bits in M and M'.

- **set of neutral bits**
  The set of bits $S \subseteq \{0, ..., 511\}$ is a neutral set of bits in relation to M and M' if all pairs of messages conform to δ_r before and after flipping any subset of the bits in M and M'.

- **set of 2-neutral bits**
  The set of bits $S \subseteq \{0, ..., 511\}$ is a 2-neutral set of bits in relation to M and M' if every bit in S is neutral and every pair of bits in S is neutral.

Basic Attack Strategy

Biham and Chen used the fact that many bits in the message are neutral bits. Some techniques were developed to find a big set of neutral bits to find a collision.

Finding 2-neutral Sets of Bits of M and M'

First the neutral bits in the message pair M and M' must be found. For this purpose on every bit position $i \in \{0, ...511\}$ the condition of the neutral bit must be tested. The 2-neutral set of bits is then the maximal subset of bits that the following condition fulfills. The subset is built up at all neutral bits which in combination with all other neutral bits conforms to δ_r. Biham and Chen observed that about $1/8$ of the found pairs confirm to δ_r. In the concrete case a near collision of a full SHA-0 needs 2^{43} pairs ($2^{43} * 2^{-3} = 2^{40}$) because the expected complexity of a near collision is about 2^{40}.

Finding better 2-neutral Sets of Bits of M and M'

In order to find a message pair with a larger 2-neutral set of bits which conforms to δ_r the given message pair is modified. The given message pair is modified in a way that the probability that the new message pair still fulfills the condition of δ_r is maximized. If a new message is found that confirms to δ_r and has a larger set of 2-neutral bits as the previous message the previous message is replaced by the generated message.

Collision Search

The first work that must be done is to increase the rounds that confirm to δ_r. To achieve that, the given pair must be modified with the methods in the previous section. When a big enough 2-neutral bit set is found it is possible to find a near collision of the full SHA-0 with a complexity of 2^{40} where only 18 bits differ. Furthermore a full collision on an 82 step SHA-0 can be found with a complexity of 2^{43}.

3.3.3 Update on SHA-1 (Rijmen Oswald)

This paper published by Rijmen and Oswald [RO05] extends the approach from Chabaud and Joux [CJ98] which is presented in Section 3.3.1. Chabaud and Joux exploit the weaknesses of the message expansion function in SHA-0. In SHA-1 this weakness no longer exists. For this reason it was necessary to find a better searching algorithm. Rijmen and Oswald use algorithms and methods of the coding theory to improve their attack. Furthermore they also analyzed other linear approximations for $f_{(if)}$ and $f_{(maj)}$.

Basic Attack Strategy

The problem is to find input values where the linear model and the original hash algorithm have an equal behavior. In other words the differences in the linear model correspond to the differences in the real SHA-1 model. The Hamming weight should be as small as possible because there is a relation between the complexity and the Hamming weight. In other words as lower the Hamming weight is as lower is the complexity to find a collision. From this it follows that a codeword with a low Hamming weight must be found.

Finding a Better Linear Approximation

The replacement of $f_{(if)}$ and $f_{(maj)}$ by $f_{(xor)}$ is not optimal. On this account a better linear approximation must be found. The main advantage of the replacement by $f_{(xor)}$ is that all 80 steps are equal. The disadvantages are that in certain cases the probability that an output bit flips, in the approximation and in the real case, especially in $f_{(if)}$ is opposite and unlike other linear functions. $f_{(xor)}$ for example has a height diffusion rate. Other linear approximations of $f_{(if)}$ and $f_{(maj)}$ that deliver better probabilities are shown in Table 3.3.

Table 3.3: The probability that the output bit changes its value when the input bits are changed according to the input difference, for MAJ and IF and for all linear approximations [RO05]

δ	OUTPUT FLIP PROBABILITY								
	$f_{(if)}$	$f_{(maj)}$	linear functions						
	$xy \oplus \bar{x}z$	$xy \oplus xz \oplus yz$	x	y	z	$x \oplus y$	$x \oplus z$	$y \oplus z$	$x \oplus y \oplus z$
000	0	0	0	0	0	0	0	0	0
001	1/2	1/2	0	0	1	0	1	1	1
010	1/2	1/2	0	1	1	1	0	1	1
011	1	1/2	0	1	1	1	1	0	0
100	1/2	1/2	1	0	0	1	1	0	1
101	1/2	1/2	1	0	1	1	0	1	1
110	1/2	1/2	1	1	0	0	1	1	0
111	1/2	1	1	1	1	0	0	0	1

Finding Codewords

For SHA-0 it is easier to find codewords with low Hamming weight as for SHA-1. For SHA-0 the search space can be limited so that it is possible to find the best codeword with a complexity of 2^{16}. In the case of SHA-1 the complexity to find the best codeword is too high for computation. On this account Rijmen and Oswald use a 53-step version of SHA-1 that delivers better results than the Birthday Attack.

3.3.4 Efficient Collision Search Attack on SHA-0 (Wang *et al.*)

In this paper Wang *et al.* describe a new technique to find full collisions on SHA-0 [WYY05c] with a complexity of 2^{39}. Equal techniques can be used to find near collisions on SHA-0 with a complexity of 2^{33}. Also for the collision attack of SHA-1 [WYY05b] with a complexity of 2^{69} these techniques can be used.

Basic Attack Strategy

Wang *et al.* introduce a nonlinear model for the first 20 steps. To achieve that, they use message modification techniques that are described later. The other 60 steps have a linear characteristic with a high probability shown in Figure 3.2. For the fact that the first 20 steps are not linear Wang *et al.* are able to disregard several conditions from the disturbance vector which is shown in Table 3.4.

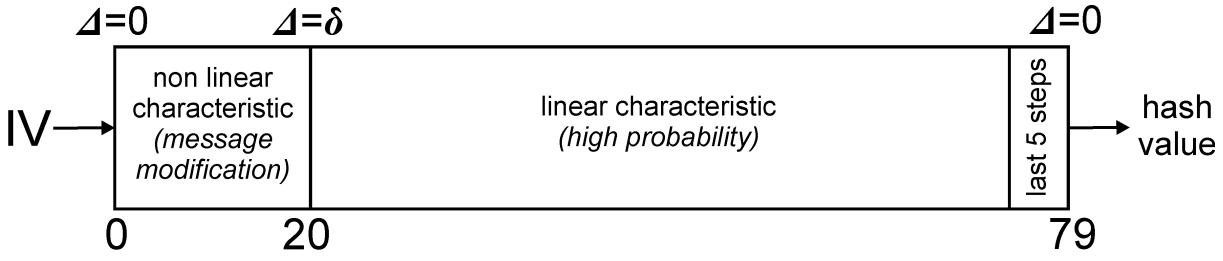

Figure 3.2: Shows the nonlinear- and the linear part of the model

Table 3.4: The three conditions on the disturbance vector [Wan97]

	CONDITION	PURPOSE
1	bit in disturbance vector = 0 in position: 75,76,77,78,79	to produce the full collision in the last step
2	bit in disturbance vector = 0 in position: -5,-4,-3,-2,-1	to avoid truncated local collisions in first few steps
3	no consecutive "1"s in the first 17 positions	to avoid an impossible collision path due to a property of IF

Disturbance Vector

In the attack that Chabaud and Joux [CJ98] published in the year 1998, three conditions on the disturbance vector were given (see Table 3.4). The new attack only requires that condition 1 must hold (This condition must hold because it is not possible to correct differences after step 80). In that fact Wang *et al.* were able to find disturbance vectors with lower Hamming weight. This has a positive effect on the search complexity. There are two additional conditions on M depending on the disturbance vector:

$$M_{i+1,7} = \neg M_{i,2}$$
$$M_{i+2,2} = \neg M_{i,2} \text{ (only for round 3)}$$

All messages with an index higher than 15 are generated from the lower ones. Therefore the conditions can be recalculated to the first 16 message words. Hence, these conditions can be easily fulfilled.

Message Modifaction Techinques

Message modification techniques are used to correct the message words to that effect that the conditions on the message words are fulfilled. First it is necessary to find a differential path that leads to a collision. Second it is necessary to find conditions that this differential path is valid. Third the message modification itself, described below, to fulfill all conditions in the first 20 steps.

- **Basic message modification techniques**
  Basic message modification techniques can be deployed for the first 16-steps because it is possible to change the first 16 message words directly by changing A on the current position i. There are several ways to satisfy a condition on the message words. First if it is allowed, the bit can be flipped directly. The second alternative is to flip the bit that stands on the position $i - 1$ and hope that a carry appears. The third option is to flip the bit on position $i - 5$ on the previous A.

- **Advanced message modification techniques**
  For the steps higher than 16 it is not possible to flip the bit directly because the message words are computed from the first 16 message words $(M_0, M_1, ..., M_{15})$. Wang *et al.* correct the conditions by correcting A_{16}. For example the condition on $A_{18,32}$ is not fulfilled, they satisfy the condition by applying basic message modification techniques on $A_{16,22}$.

Complexity Analysis

Because the advanced message modification works up to step 20 and disturbance vectors were found with a lower Hamming weight the complexity to find a full collision on SHA-0 can be reduced to 2^{39}. The lowest search complexity for a full collision on SHA-0 at this time was 2^{56} which was found from

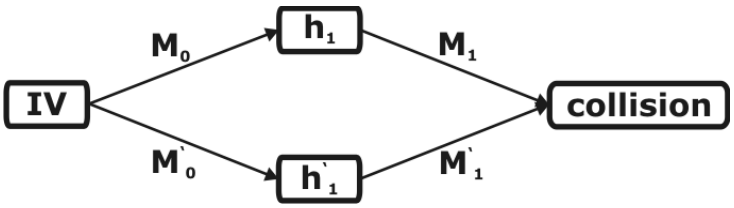

Figure 3.3: a two-block collision

Biham *et al.* [BCJ+05]. The fact that the complexity to find a full collision on SHA-0 is 2^{39} makes it possible that a collision can be found on a standard computer in a few days.

3.3.5 Finding Collisions in the Full SHA-1

The basic strategy behind the attack is pretty similar to the attack on SHA-0 described in Section 3.3.4. Because of that fact only the differences are covered in this section.

Disturbance Vector

As mentioned before in the case of SHA-0 an 80-bit vector is used as disturbance vector. For SHA-1 this 80-bit vector is not sufficient anymore. Because of the rotation in the message expansion 16 32-bit variables are needed instead of 16 1-bit variables to describe the whole search space. This results in a search space as large as 2^{512}. To search the entire space would not be an easy task. Therefore, Wang*et al.* use heuristics to reduce the search space and search only in areas that likely contain vectors with low hamming weight. The remaining space is at about 2^{38}, which is a huge improvement compared to 2^{512}. In order to create an attack which is more efficient than the birthday attack, the hamming weight of the disturbance vector has to be less than 27. This is only possible if the conditions shown in Table 3.4 are removed.

Collision Search

The fact that the conditions in Table 3.4 are not valid anymore for the given disturbance vectors, complicates the construction of a valid differential path, so two near collisions are used to form a collision. These two-block collisions are constructed in the sense, that the differences erase each other. Figure 3.3 illustrates the assembly of such a multi-block or in this case a two-block collision. M_0 and $M_0^{'}$ produce h_1 and $\delta h_1 = h_1^{'} - h_1$

Message Modifaction Techinques

The same message modification techniques are used as described in Section 3.3.4 with the exception that advanced message modification can be applied up to step 22. Later Wang *et al.* claimed [WYY05a] (results unpublished) to apply message modification up to step 25 using another differential path and additional message modification techniques.

Complexity Analysis

These techniques led to the first attack which has a lower complexity than the birthday attack. With the help of message modification techniques and early stopping it is possible to find near collisions with a complexity of 2^{68}. The near collision on the second message block can be derived with the same complexity as the first near collision. Because of that fact the complexity of the full collision is just

increasing by a factor of two. From that it follows that the overall complexity to produce a full collision is 2^{69}. According to unpublished results [WYY05a] Wang *et al.* were able to decrease the complexity to about 2^{63}.

3.3.6 The Amplified Boomerang Attack

The presented attack [JP07] is an adapted version of the amplified boomerang attack on block ciphers [KKS00], which is again based on the boomerang attack proposed by Wagner [Wag99].

The Boomerang Attack

To completely understand the amplified version of the boomerang attack the original boomerang attack [Wag99] itself has to be explained first. Basically the boomerang attack is a differential attack which uses a structure of four plaintexts and certain differentials that are applied to the plaintexts and the internal states of the decryption/encryption phase of these given plain-texts. Figure 3.4 shows the structure of the attack which is explained as follows:

1. Choose an input pair X_i, X_j where $X_j = X_i \oplus \Delta_0$
2. Encrypt the pair and half way through using E_0 and get Y_i and Y_j with the relation $Y_j = Y_i \oplus \Delta_1$
3. Encrypt Y_i and Y_j using E_1 which results in Z_i and Z_j
4. Generate Z_k and Z_l through $Z_k = Z_i \oplus \Delta_1$ and $Z_l = Z_j \oplus \Delta_1$
5. Decrypt Z_k and Z_l using E_1 which results in Y_k and Y_l which results into the relations $Y_k = Y_i \oplus \Delta_0$ and $Y_l = Y_j \oplus \Delta_0$
6. Compute relation between Y_k and Y_l:
   $Y_j = Y_i \oplus \Delta_1$, $Y_k = Y_i \oplus \Delta_0$, $Y_l = Y_j \oplus \Delta_0$
   $Y_k = Y_i \oplus \Delta_0$
   $Y_k = Y_j \oplus \Delta_1 \oplus \Delta_0$
   $Y_k = Y_l \oplus \Delta_0 \oplus \Delta_1 \oplus \Delta_0$
   $Y_k = Y_l \oplus \Delta_1$

$Y_k = Y_l \oplus \Delta_1$ determines the relation $X_k = X_l \oplus \Delta_0$. Because of this relations it is possible to distinguish E from random permutations.

Adapting the Boomerang Attack to Attack Hash Functions

Because of the fact that there is no decryption in hash functions the boomerang attack can't be applied directly. The so called amplified boomerang attack [KKS00] however can be applied to attack hash functions with certain adaptations. The adaptations and the subsequent attack are described in [JP07]. The basic idea is to apply so called auxiliary differential paths, which work good for a certain number of steps, to a main differential path to reduce the overall work factor. The principals of the boomerang attack are now used to attach the auxiliary differential paths to the main differential path. Figure 3.5 shows the structure of the adapted version of the attack.

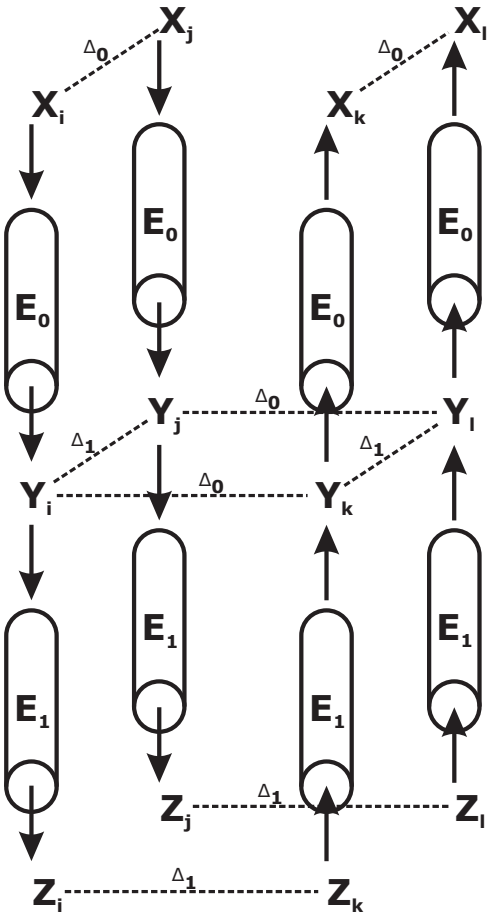

Figure 3.4: Structure of the boomerang attack

Like in the original attack a plaintext pair is chosen with a differential Δ which is the main differential path. In addition to the main differential path an auxiliary path is applied to the message pair. As mentioned before this path is only ”good” for a limited number of steps, and therefore covers only the early and the middle steps of the compression function. This attack only succeeds if the main differential path contains all the conditions needed for each auxiliary path that is applied to the main differential path. In order to build such a differential path Joux and Peyrin use a path generator proposed in [CR06].

Complexity Analysis in View of SHA-1

According to unpublished results [WYY05a] the best known attack Wang *et al.* were able to apply, was message modification up to step 25. With the new techniques presented above it should be possible that messages conform to the differential path up to step 28 which means that 5 additional constrains on the message are fulfilled. Hence the overall complexity decreases by a factor of 32 if various other conditions could be met.

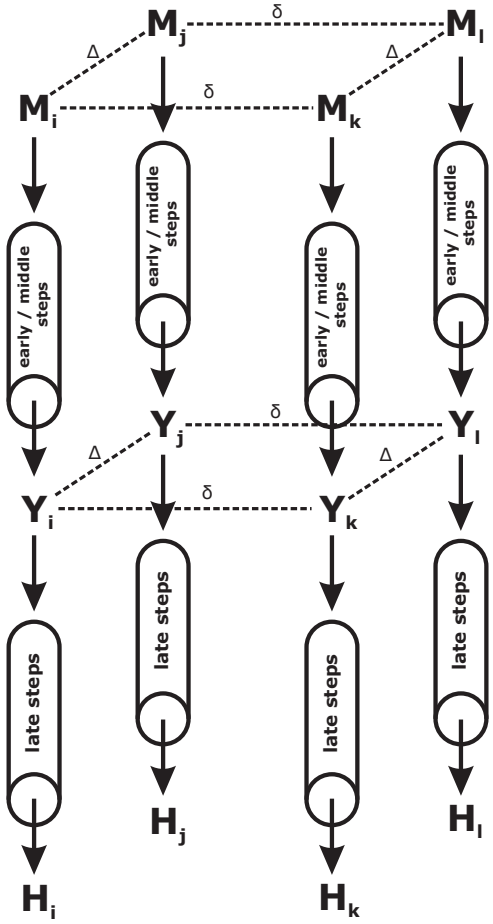

Figure 3.5: Structure of the amplified boomerang attack adapted to attack hash functions

3.3.7 Clustered Truncated Differentials and Fast Collision Search

The method presented in [Rec09] and [MR] is an advancement of message modification and is at the time of writing this thesis the best known attack on SHA-0 and SHA-1. To find a collision on SHA-1 using this methods, there was also a distributed computer effort [MRR09] launched which is suspended at the time. A variant of this attack is implemented in the case study which is described in Chapter 6

Equations

To find a pair of colliding messages, which is conforming to a certain characteristic, equations have to be solved which are bound to this characteristic. Because of the fact that in every step the state update function is computed, which adds diffusion to the input message, equations in later steps are in general harder to solve than equations in earlier steps. One way to solve equations is through message modification which is described in Section 3.3.4.

Message Modification Patterns

In the attack two different message modification patterns are used. The differentiation is between short patterns and long patterns which are explained in the paragraphs 3.3.7 and 3.3.7. In general long patterns are able to solve equations of higher steps than short patterns. This advancement comes with a price in view of degrees of freedom. There are also certain limitations connected with the usage of these patterns.

Short Patterns: Short patterns are affecting equations between the steps 17 and 22. There are no short patterns known for equations in steps later than 22. In general a small weight difference is injected into the message pair. This difference propagates through the state update function and flips a relevant equation bit with a high probability. In addition the probability that an already solved equation changes its state to not valid, is very low. Figure 3.6 shows the structure of short patterns. The difference introduced at a certain step, requires the re-computation of all the steps up to the step of the targeted equation. Short patterns have a big advantage over long patterns in view of degrees of freedom. Compared to long patterns short patterns require much less degrees of freedom.

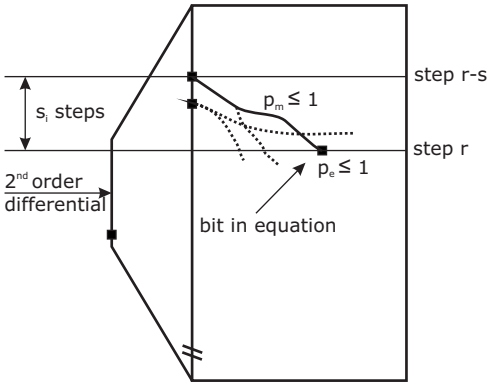

Figure 3.6: Short pattern

Long Patterns: Long patterns are based on the principles of short patterns. As mentioned in Section 3.3.7 more degrees of freedom are used compared to short patterns to force a local collision. These patterns can be used to fulfill equations between the steps 23 and 30. The differences which are introduced in the earlier steps reappear because of the message expansion in later steps. They solve the targeted equation with a high probability. Equally to short patterns the equations which are already solved remain untouched with a high probability. Figure 3.7 shows the structure of long patterns.

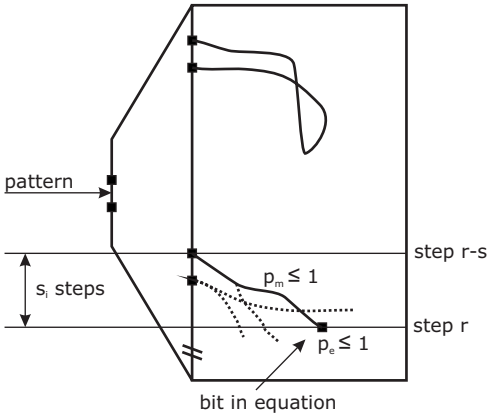

Figure 3.7: Long pattern

Finding Characteristics

As mentioned before solving equations in higher steps with the help of long patterns requires a lot degrees of freedom. In consequence to this the remaining search space is not big enough to perform a full search. Therefore different characteristics are used for a full search. In the attack described in Section 3.3.5 Wang *et al.* constructed the used characteristic by hand. To run a full search it is therefore necessary to create a tool which is able to automatically generate different characteristics. This problem is targeted in the paper [CR06] by Cannière and Rechberger.

In general they construct characteristics by adding conditions to a characteristic as long as the workfactor[2] of the characteristic is improving. This process consists of two important components:

- The conditions have to be consistent and have to propagate.
- There has to be a determination which conditions to add.

Finding Collisions

Applying the tools discussed in the sections above the estimated complexity should be in the range of about 2^{60}. This is the best estimated complexity for collision attacks on SHA-1 at the time. Also a complexity of 2^{60} is getting in range of a computation.

[2]The workfactor of a characteristic is the required effort that two messages follow the characteristic

Chapter 4

Stream Processors

In general stream processors try to fill the gap between special-purpose processors and general-purpose processors. [KDR+03] Particularly for real-time media applications special-purpose processors are better suited than general-purpose processors because they are able to fulfill the in general high performance demands of media applications better. However the big disadvantage is that special-purpose processors are not flexible in their programming or at least limited in their flexibility. [KRD+03]

General-purpose processors on the other hand are highly flexible, but their architectures are in general not very well suited for the special demands of typical media applications. [KRD+03] They are in general optimized to reduce data latency and reuse data, which results in much more complex data and instruction delivery systems on the chip. In contrast to stream processors general-purpose processors devote only a fraction of the space available on the chip to the actual computation. [Pet09] For example on the Itanium 2 processor only 6.5% of the die is dedicated to ALUs and their registers. [NCBF+02]

An application which is suited for a stream processor often fulfills the following three characteristics: [KDK+01]

- There is always a huge amount of data parallelism involved because the data elements are highly independent.
- For each data element a high number of operations is typically performed resulting in a high latency tolerance.
- There is almost no reuse of global data.

Not long ago stream processors were almost only used in media applications. This changed drastically in the last years because on one hand GPUs are now commonly used in many PCs and on the other hand manufacturers now provide APIs to program these GPUs to compute problems of general purpose. Two prominent applications using GPUs for example would be folding@home [Fol] and seti@home [ACK+02].

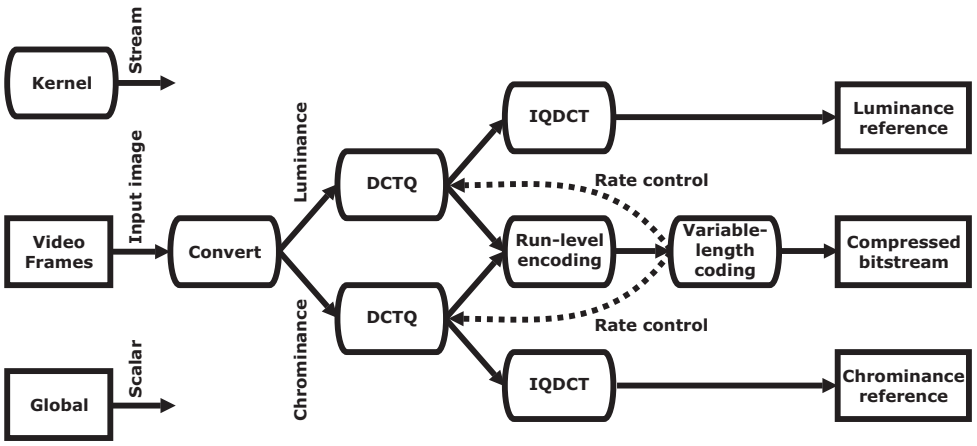

Figure 4.1: The stream programming model explained on the example of an MPEG-2 I-Frame encoder

4.1 Stream Programming Model

To fully understand the mechanisms and the structure of stream programs, first a clear terminology has to be defined. For that purpose the so called stream programming model is the tool of choice. The basic concept of the stream programming model is to arrange an application into kernels and streams to expose concurrency and locality. It is easy to see that this structure is well suited for stream architectures with a high number of ALUs. [Rix02]

A Kernel is basically a small program which is applied to every element in the input stream. In view of the kernel each input element is independent and all operations are performed locally. Therefore the computation of these elements can be done in parallel. Each Kernel has an input-stream and an output-stream. Normally data which is generated by the kernel locally, by computing the elements in a stream, doesn't need to be referenced by other stream elements or kernels. [Rix02] However in some cases the programmer has some kind of shared memory to his disposal which makes it possible to share local data of different stream elements within a kernel. With a few exceptions kernels are only allowed to access data from input streams or output streams. [KRD+03]

Streams are records or fields of data which are serving as input and output to a kernel. A stream element can be a primitive data type, such as an integer or a float, but can also be a whole data structure. [KRD+03]

Figure 4.1 shows the principles of the stream programming model as an exemplary MPEG-2 I-frame encoder. "Video frames", "Luminance reference", "Chrominance reference" and "Compressed bitstream" are container for global data. "Input Image" is the first stream which is fed into the kernel "Convert". The "Convert" kernel itself produces data streams which are again fed into other kernels. At the end the different kernels are streaming their produced values back into the global data containers. [KRD+03]

4.2 Additional Terminology

4.2.1 Scatter and Gather

Although communication between the individual streams should be minimized, in some cases it can be advantageous. Sometimes for performance sake, it is helpful if a kernel has access to other cells than the one currently being processed. Especially in the case of data communication on GPUs gather and scatter are two types of communication worth mentioning. If a kernel, which currently processes a stream element, needs information from other stream elements it gathers information from other parts of memory. Scatter occurs if a kernel processing a stream element distributes information to other parts of the memory. [PF05]

4.2.2 SIMD

SIMD stands for single instruction multiple data, which means that these architectures issue the same instruction for different data elements. Often stream architectures are structured in a way that a certain number of ALUs are building a SIMD cluster. Within the cluster all ALUs compute the same instruction. SIMD cluster are normally independent, which means that different SIMD cluster are often able to compute different instructions at the same time. [NVI09c]

4.3 The Imagine Stream Processor

The Imagine stream processor was developed at Stanford and at MIT from 1996 to 2001 and is basically a load and store architecture for streams. 48 arithmetic units organized in 8 arithmetic clusters act as workhorses for the processor and are able to obtain a computational performance of 20 gflops. [Rix02] Imagine is a programmable stream processor which is controlled by a host processor and is able to execute applications directly with the help of streams and kernels. Figure 4.2 pictures the Imagine stream architecture.

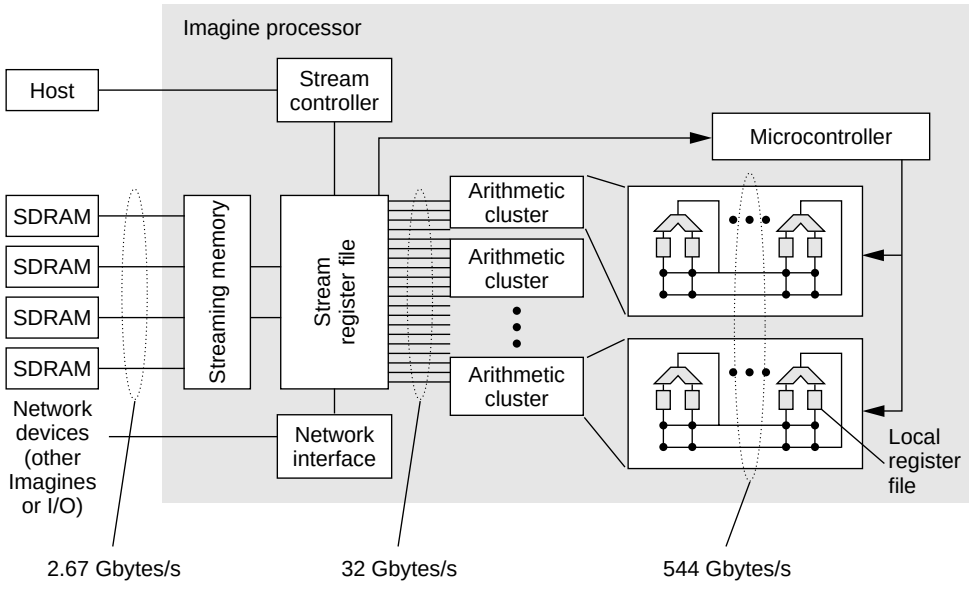

Figure 4.2: The imagine architecture [Rix02]

The main stream instructions are:

- **Load** transfers streams from the DRAM to the SRF
- **Store** transfers streams from the SRF to the DRAM
- **Send** transfers streams from the network to the SRF
- **Cluster op** computes the kernels in the given cluster and writes the output streams back into the SRF

The so called SRF (stream register file) is responsible for storing and loading streams which are generated by the kernels. The host processor issues stream instructions to the stream controller which when ready issues them to the 8 arithmetical clusters. The arithmetical clusters compute the given instructions in SIMD fashion. [KDK+01]

4.4 The Cell Multiprocessor

The cell processor was introduced in the year 2005 and is a collaboration of IBM, Sony and Toshiba. The motivation to develop this multiprocessor originated from the fact that traditional architectures would not be able to deliver the computational power necessary for their needs in the near future. One of the goals was to create a multiprocessor which was able to achieve about 100 times the processing power of the PlayStation2. [KDH+05] Outstanding performance especially on game and multimedia platforms, real-time response to user and network, applicability to a wide range of platforms and support for introduction in 2005 were also main objectives for the cell processor. [CRDI07]

The Cell architecture shows strong similarities to a CPU/GPU setup in an architectural view. In general the Cell processor consists of a Power Processing Element (PPE) and eight synergistic processor elements (SPEs) connected through a coherent on-chip element interconnect bus (EIB) shown in Figure 4.3. The PPE basically takes over the role of the CPU and the SPEs can be compared to the thread processors on the GPU. One big advantage of the cell architecture is that the EIB is on chip in comparison to the pci-express bus. [KDH+05]

In view of computational power one SPE has a peak performance of 25.6 Gflops. Therefore the whole theoretical computational power of the SPEs is 204.8 Gflops. Compared to GPUs which were released approximately at the same time, the Cell processor has a very similar computational capability than the top of the line GPUs in 2005. Figure 4.6 shows benchmarks of GPUs and the cell multiprocessor.

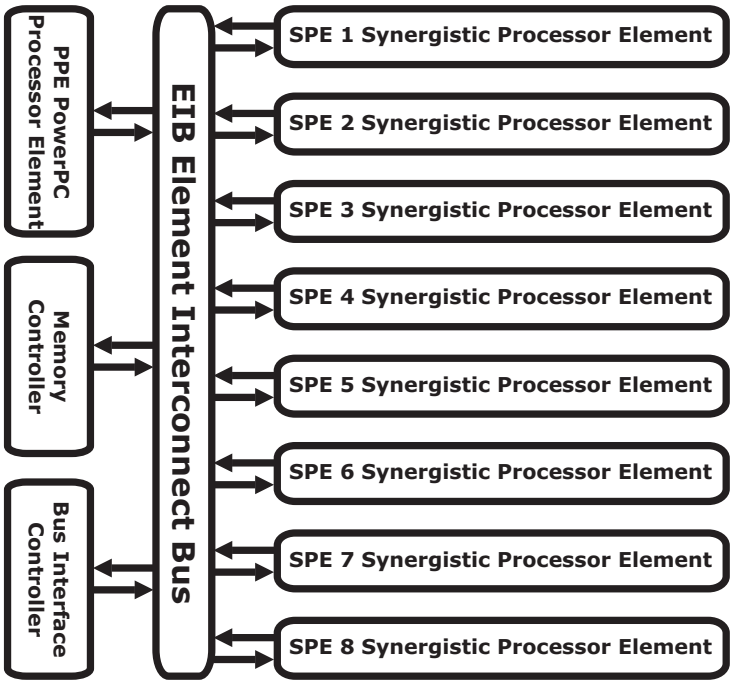

Figure 4.3: Architectural overview of the cell multiprocessor

4.5 GPGPU

For many years now programmers use GPUs to compute problems of general purpose. In general GPUs offer high performance not only in view of computational power but also in view of memory throughput. For a long time these capabilities were hard to access due the fact that GPUs were formally designed to process graphical data and not data of general purpose. As computer graphics applications became more and more demanding, for example to compute 3D Data, the structure of GPUs grew also more flexible.

4.5.1 Historical vs. Modern

The following sections show an overview of the technical developments of GPUs over time.

Traditional Graphics Pipeline

Figure 4.4 shows a traditional graphics pipeline which existed in the depicted structure in principle for the last 20 years. Every function had its own processor for the computation. [NVI06]

Structure of the traditional graphics pipeline: [NVI06]

- **Vertex:** In the first step the GPU receives vertex data from the CPU. The vertex shader, who evolved from so called ”Transform and Lighting”[1], transforms the 3D position of the vertex into the 2D screen position. They are also responsible for coloration and position properties.
- **Triangle:** The next step is the is the ”triangle stage”, where primitives such as triangles, lines or points are generated from the vertices.

[1]Transform converts spatial coordinates into a 2D view. Lighting is responsible for the coloration of objects due to the influence of lighting objects.

- **Pixel:** The primitives are then converted into pixel fragments by the ”pixel” unit. After further processing the pixel fragments are written into the frame buffer.
- **ROP:** The ROP (Raster Operation) stage is responsible for checking visibility, transparency and anti-aliasing.
- **Memory:** The final processed pixel is sent to the frame buffer memory for scan-out and display to the monitor.

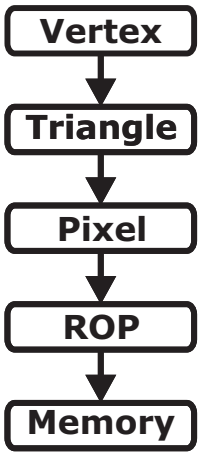

Figure 4.4: Classic graphics pipeline

Modern Graphic Pipeline

With the introduction of NVIDIAs 8000 series and AMDs HD 2000 series the setup of the traditional graphics pipeline changed drastically. The graphics pipeline model is replaced with a so called ”Unified Pipeline and Shader Design” which is shown in Figure 4.5. [YG07]

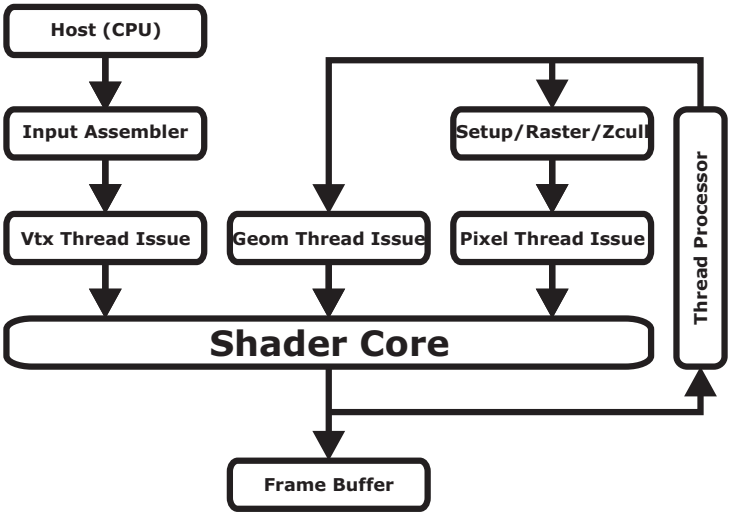

Figure 4.5: Unified pipeline and shader design

This approach changes the standard sequential flow into a loop oriented flow. Data is fed into the shader core, which processes it and writes the outcome into the registers, where it can be again fed into

the shader core and so on. [NVI06] A comparison between this architecture and the classical graphics pipeline design shows that the unified shader core is much better suited for general purpose computing because all ALUs are programmable and therefore are able to compute different types of programs. [YG07]

4.5.2 Comparison of GPUs and CPUs

It is interesting to see how GPUs and CPUs evolved over time in view of computational capabilities and bandwidth capabilities. Figure 4.6 shows clearly that in view of raw computational power stream processor architectures are superior to general-purpose processors. It's a fact that the computational power of the imagine stream processor and the cell multiprocessor aligns with the computational power of state of the art GPUs, at the time of their launch. Therefore, it stands to reason that the superior computational capabilities of stream processors compared to general purpose processors is due to the architectural differences.

Table 4.1 clearly shows that the gap between GPUs and CPUs widens over time in view of computational capabilities shown in Figure 4.6. This is probably because of the fact that stream processors compared to traditional multi-purpose processors are a relatively young field of application. Due to this fact there is probably more room for improvement in the field of stream processors which explains the increasing ratio shown in Table 4.1

An interesting observation is that although given similar premises between bandwidth capabilities shown in Figure 4.7 and computational capabilities, the ratio between GPUs and CPUs in view of bandwidth capabilities is fairly constant as Table 4.1 shows.

Table 4.1: Ratio GPUs/CPUs in view of bandwidth and computational power

Ratio GPUs/CPUs		
Year	Gflops	Bandwidth
2004	14	11
2006	20	8
2007	13	12
2008	26	11
2009	39	6

4.5.3 Theoretical Performance Capabilities

It is possible to theoretically compute the performance of GPUs in view of Gflops by knowing the ALU frequency, the number of ALUs and how much operations the architecture itself can compute in one clock cycle. Equation 4.1 shows how to compute the theoretical performance of a GPU.

$$Gflops = \#ALUs * ALU clockrate * possible\ instructions\ per\ clock\ cycle \tag{4.1}$$

For example the NVIDIA GTX 260:
ALU frequency: 1242 MHz
Number of ALUs: 192
Instructions per clock cycle: 3 (MADD (2 flops) and MUL (1 flop))
Gflops = 1.242 * 192 * 3 = 715.392 Gflops

Another example would be the ATI HD 4870:
ALU frequency: 750 MHz
Number of ALUs: 800
Instructions per clock cycle: 2 (MADD (2 flops))
Gflops = 0.75 * 800 * 2 = 1200 Gflops

It is very important to state that these numbers are only theoretical and cannot be directly translated into real performance. In the case of NVIDIA for example the extra MUL operation is not always available, as Section 6.3.4 shows.

4.5.4 Gflops and Bandwidth Comparison

Figure 4.6 shows that the performance measurements of ATI and NVIDIA in view of Gflops follow a very similar path. This indicates that both manufacturers are at a close distance to each other in view of their technical progress. It can also be seen that the trendlines of ATI and NVIDIA shown in Figure 4.6 follow a similar path which falls into line of the previous statement.

Very interesting to see is also that the two stream processors, Imagine described in Section 4.3 and Cell described in Section 4.4, exhibit similar benchmark results than the GPUs which were current at the time.

The exact numbers for Figure 4.6 are given in Appendix A

Figure 4.7 shows that the bandwidth measurements of ATI and NVIDIA in view of Gflops follow a very similar path. It can also be seen that the trend lines of ATI and NVIDIA shown in Figure 4.7 follow a similar path. This correlates with the trend of the performance capabilities of ATI and NVIDIA. The exact numbers for Figure 4.7 are given in Appendix A

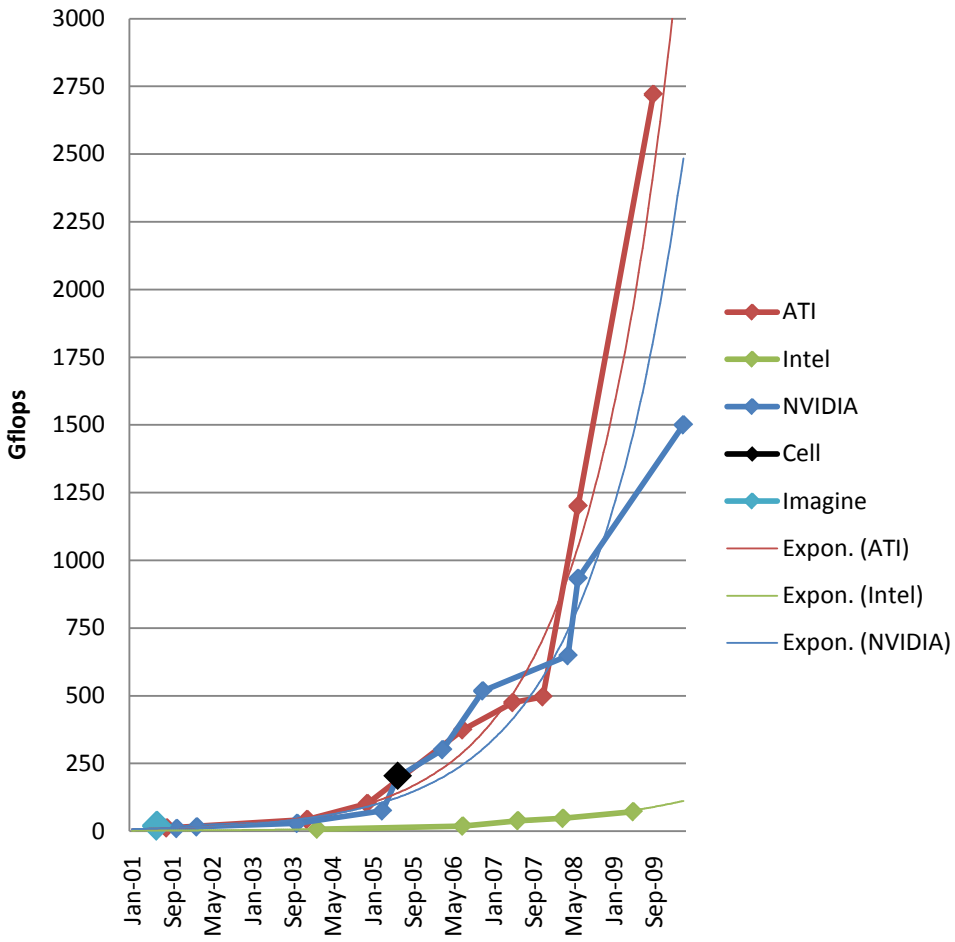

Figure 4.6: Comparison Gflops

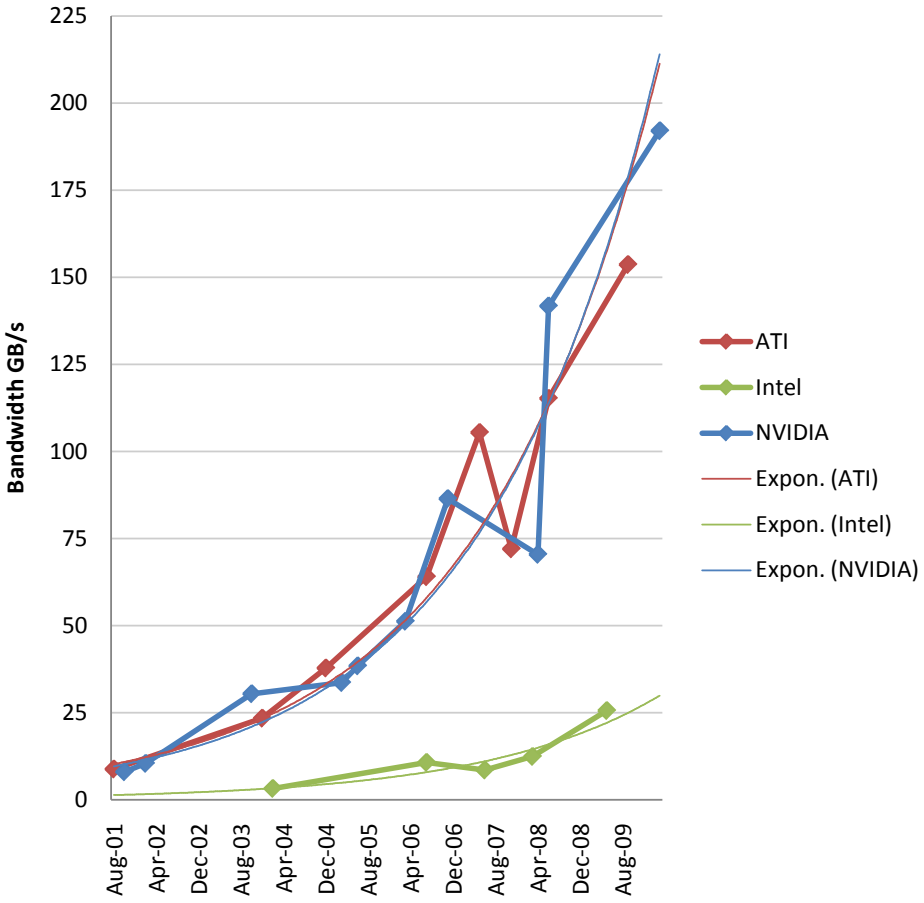

Figure 4.7: Comparison bandwidth

Chapter 5

Cuda

CUDA stands for Compute Unified Device Architecture and is basically a general purpose parallel computing architecture. It uses CUDA capable GPUs to solve computationally complex problems. [NVI09c] Many design decisions regarding the case study in Section 6 only make sense in conjunction of hardware capabilities and hardware limitations of the GPU and of CUDA itself. Therefore a basic introduction of the concept and a detailed description of the key components in view of the case study are given in this section. Regarding terminology from now on the GPU is also referred to as ”device” and the CPU is also referred to as ”host”. The whole system is built-up on a layered structure shown in Figure 5.1. On top are the application layers which are based on the different driver APIs. [NVI09a]

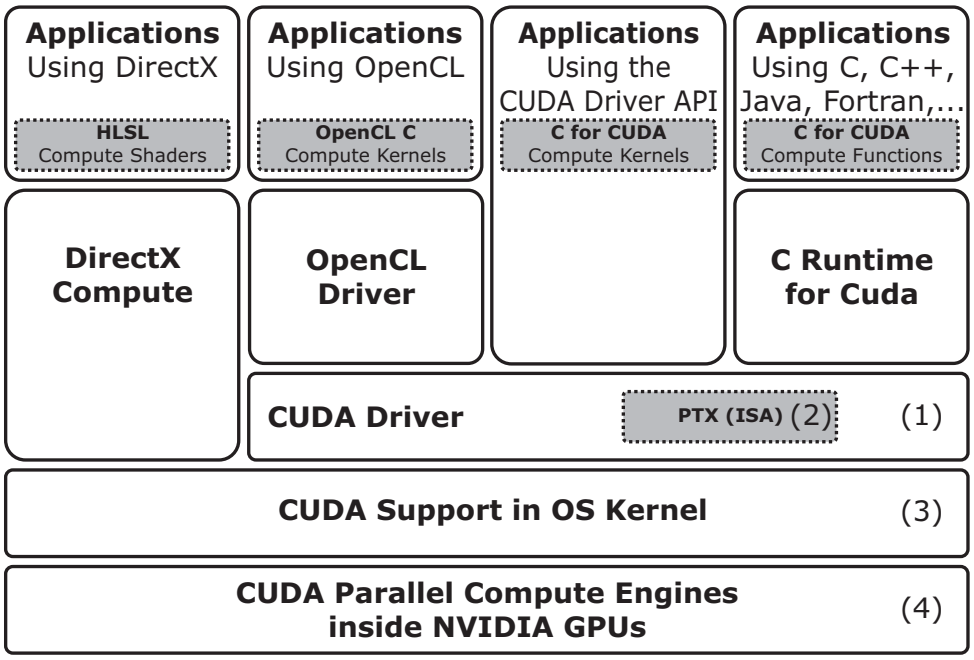

Figure 5.1: The CUDA architecture

The CUDA architecture itself consists of the following four key components [NVI09a]:

- User-mode driver, which provides a device-level API for developers. (1)
- PTX instruction set architecture (ISA) for parallel computing kernels and functions. (2)

- OS kernel-level support for hardware initialization, configuration, etc. (3)
- Parallel compute engines inside NVIDIA GPUs. (4)

5.1 Architectural Differences Between GPU and CPU(Data Caching vs. Data Processing)

Due to different fields of applications, GPUs and CPUs have fundamentally different hardware structures. A typical CPU devotes a large amount of space to data caching and flow control as can be seen in Figure 5.2. For a typical GPU however, data caching and flow control are two tasks which are relatively easy to perform. The fact that on a GPU data elements are computed in parallel by the same program or more precisely by the same instructions, (see Section 5.3.3) is responsible that flow control is not that sophisticated for GPUs than for CPUs. Also typical applications for GPUs have high arithmetic intensities, which in assistance to high parallelism leads to the effect, that memory access latency can easily be hidden due to calculations instead of data caching. [NVI09a]

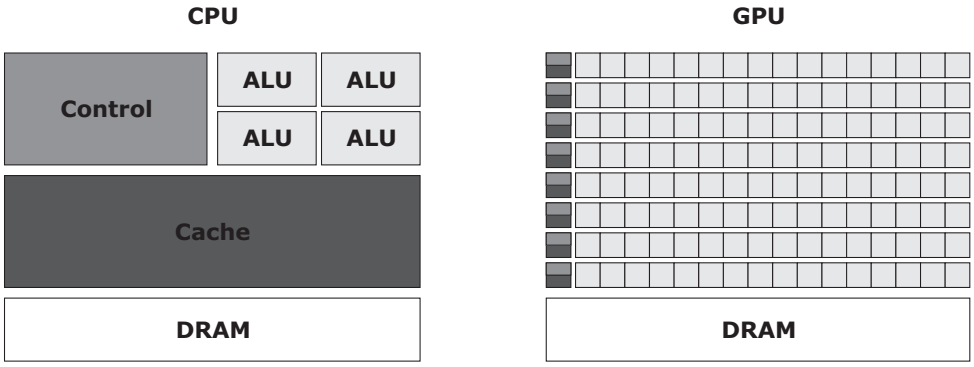

Figure 5.2: Sample GPU space and CPU space on the die

5.2 Device and Host Communication

Host and device are communicating through the PCI express interface, which in a nutshell is a general purpose I/O interconnect, for a wide variety of computing platforms. PCI express works as a point-to-point connection and provides a very high-speed interconnect. Each PCI express link contains a certain number of lanes and is therefore variable in view of bandwidth capabilities[Int02]. The GeForce 200 series from NVIDIA for example uses PCIe 2.0 x16. PCIe 2.0 is designed to transfer 500 MB/s per lane, which multiplies up to 8 GB/s for the x16 (16 lanes) configuration [PCI07, Int04]. Figure 5.3 shows the basic architecture of a general purpose desktop with a PCI express interconnect.

5.3 Architectural Overview of CUDA Capable GPUs

As of now the G80, G90, GT200 and Fermi are the only CUDA capable GPUs. In principal all different GPUs have a similar architectural structure. They are all based on the uniform shader architecture, which is described in Section 4.5.1.

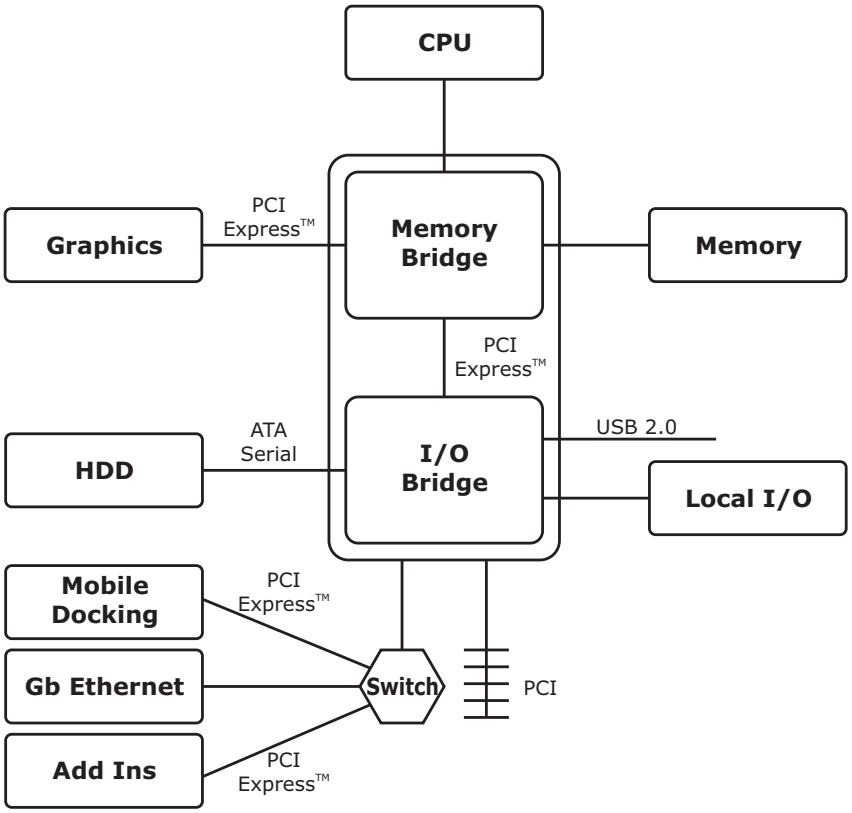

Figure 5.3: PCI express architecture

5.3.1 Streaming Multiprocessor

On the GPU all the units are divided into so called streaming multiprocessors. On the G80, G90 and GT200 architecture a streaming multiprocessor consists always of 8 thread processors, one warp scheduler, shared memory and constant memory. Figure 5.4 shows the architecture of the streaming multiprocessor. In the case of G80 and G90 two streaming multiprocessors form a so called thread processing cluster (TPC). The GT200 architecture is very similar in that way, with the exception, that 3 streaming multiprocessors form one TPC [NVI09c]. The Fermi architecture is different in regard that each streaming multiprocessor consists of 32 thread processors and two warp schedulers. Also the streaming multiprocessors are no longer bundled into TPCs. [NVI09b] The structure of the streaming multiprocessors may differ between Fermi and the other CUDA capable GPUs, but in general they are all built-up upon the same principal.

5.3.2 Memory Model

Threads have access to different types of memory. Each type of memory has different characteristics and different advantages. These memory spaces include local memory, global memory, constant memory, texture memory, shared memory and registers. A basic distinction between all types is in ”off-chip memory” and ”on-chip memory” as shown in Figure 5.5. In general, ”off-chip memory” has a much slower access rate than ”on-chip memory”.

Global memory and texture memory are the biggest memory spaces, and have also with local memory the greatest access latency. Followed by constant memory, shared memory and registers. [NVI10b, pp. 19-20]

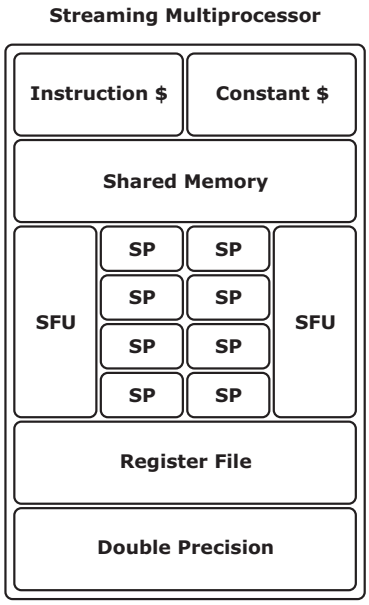

Figure 5.4: Streaming multiprocessor G80, G90, GT200

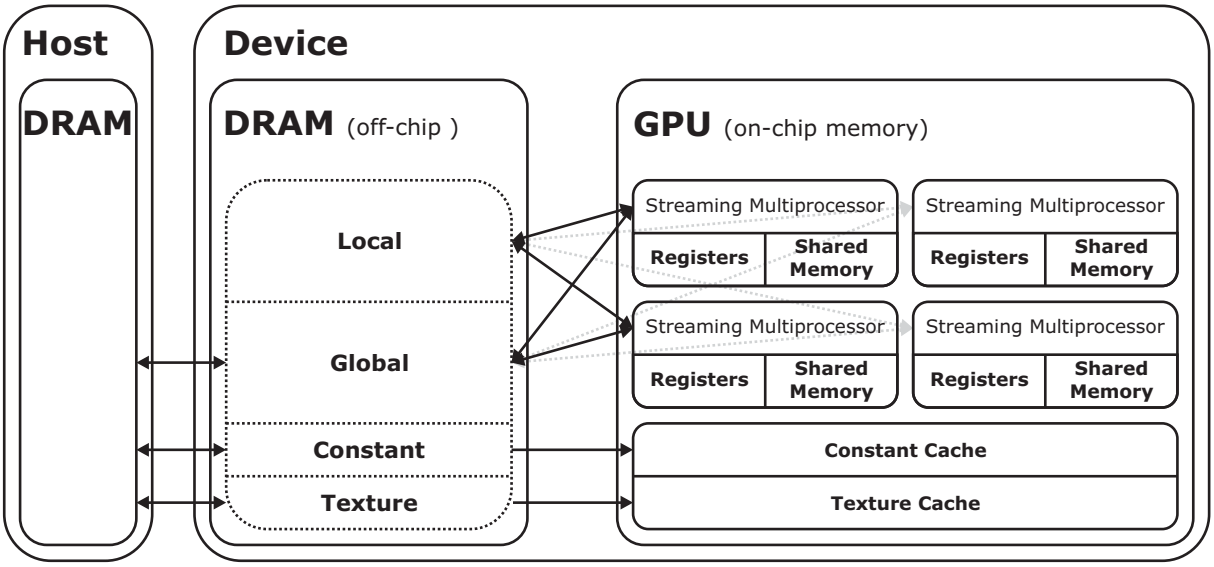

Figure 5.5: The various memory spaces on a CUDA device

Table 5.1 shows the various attributes of the different memory types which are described in detail in the following.

Global Memory

Global memory resides in the device memory and is accessed via 32-, 64-, or 128 byte transactions. A warp accesses the global memory in such a manner that it coalesces the memory access of the threads into a number of memory transactions. The number of transactions is dependent on the size and the distribution of the requested memory addresses. Dependent on these two criteria, the memory access is more or less efficient. Only if the requested data is naturally aligned the operation is executed in one

Table 5.1: Attributes of the different memory spaces

MEMORY	LOCATION ON/OFF CHIP	CACHED	ACCESS	SCOPE	LIFETIME
Register	On	n/a	R/W	1 thread	Thread
Local	Off	only com. cap. 2.0	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	only com. cap. 2.0	R/W	All threads and host	Host allocation
Constant	Off	Yes	R	All threads and host	Host allocation
Texture	Off	Yes	R	All threads and host	Host allocation

single global memory instruction. [NVI10c, p. 86]A detailed description of coalesced memory accesses is given in Section 5.5.7

Local Memory

Like global memory, local memory resides also in device memory. The main difference is that the compiler decides when variables are put into local memory space. [NVI10c, p. 87]

There are three main reasons for this to happen:

- Arrays are indexed at runtime and not at compile time.
- Large structures or arrays which would occupy to much register space.
- A kernel uses more registers than available.

Shared Memory

Shared memory resides on the chip. Therefore its access is much faster than the access of local- and global memory. It is expected to be a low-latency memory. Shared memory is partitioned into 16 equally sized memory banks which can be accessed at the same time. If two or more addresses of a memory request fall into the same memory bank, the memory access has to be serialized to avoid any bank conflicts. The bank with the highest count of accessed addresses determines the number of necessary memory request and subsequently the decrease in bandwidth. [NVI10c, p. 10, p. 86]

Constant Memory

Constant memory is located in the device memory but its access is speeded up by caching it into the constant cache. The constant cash is residing on-chip and is therefore a low-latency memory space. [NVI10c, p. 86, p. 143]

Registers

Each thread has its own set of registers which cannot be shared with other threads. The number of available registers for each thread is determined by the number of threads running on the same streaming multiprocessor and vice versa. [NVI10c, p. 86, p. 143]

5.3.3 Single Instruction Multiple Thread (SIMT)

Each streaming processor is designed to execute a large number of threads in parallel. To compute and schedule these threads, the streaming multiprocessor implements a new concept called single instruction multiple thread or short SIMT. The SIMT concept is similar to the single instruction multiple data (SIMD) concept, because both architectures execute a single instruction on a certain number of processing elements. SIMT differs from SIMD in a way that it is possible for the programmer to write thread-level parallel code as well as data-level parallel code. Also SIMT instructions specify the branching and execution behavior on thread level in contrast to SIMD, where the software has to handle the SIMD width. [NVI10c]

5.3.4 Warps and Thread Blocks

On a streaming multiprocessor, threads are always grouped in so called warps. For devices with compute capability 1.x each warp consists of 32 threads which are executed in parallel. All threads in a warp have the same start program address. Each thread has its own instruction address counter and register state. Therefore, a thread is able to branch and compute instructions independently within a warp. When a streaming multiprocessor is executing a block, it partitions the block into individual warps. On the streaming multiprocessor the block size is always a multiple of 32. In special cases, such as coalesced memory accesses, the term half-warp is also important to know. A half-warp consists always of 16 threads. The first 16 threads of a warp are building the first half-warp. The last 16 threads are building the second half-warp. [NVI10c]

5.3.5 Multithreading

Each streaming multiprocessor contains the execution context of each thread and subsequently each warp. Therefore switching the execution context of the warps has no cost for the multiprocessor. At every instruction issue time, the warp scheduler selects a warp with active threads and issues the next instruction for those threads. From that it follows that the number of warps, and in further consequence the number of blocks a streaming multiprocessor is able to hold, is limited by the hardware itself. [NVI10c]

5.4 CUDA API

5.4.1 Versioning and Compute Capability

The compute capability of a CUDA capable GPU is defined by a minor- and a major revision number. GPUs with the same core architecture have the same major revision number. The minor revision number defines improvements within the same core architecture. [NVI10c, p. 14]

5.4.2 Kernel

At the beginning the program is always executed on the host. At a certain point in the program, the host administrates a so called kernel invocation. The kernel invocation is the entry point to the device and has always the following structure:

```
kernel<<<dimGrid, dimBlock>>>(...)
```

While dimGrid describes the size, or more precisely the number of blocks in the grid, dimBlock describes the number of threads within a block. [NVI10c, p. 7]

5.4.3 Thread Structure

In order to fully utilize the GPU, it is necessary to run a really high number of threads at the same time. To work with that many threads efficiently the CUDA API provides a grouping mechanism. Figure 5.6 shows how the threads are arranged. Always a certain number of threads are building one block. Again, a certain number of blocks are building one grid. Furthermore, each thread has a unique and consecutive ID within a block called thread ID. The threadIdx Vector, which contains the thread ID, is composed of three components. Therefore, threads can be identified as one-dimensional, two-dimensional or three-dimensional vectors. Figure 5.6 for example depicts a two-dimensional thread block of size (D_x, D_y). Each thread within a block is running on the same streaming multiprocessor. This is an important fact because it is the only way to efficiently and safely use the on-chip shared memory. The limitation to the same streaming multiprocessor also restricts the number of threads in a block. GPUs with different compute capabilities have also a different number of threads one block is able to contain. The exact numbers are given in Section 5.5.2. [NVI10c, pp. 8-10]

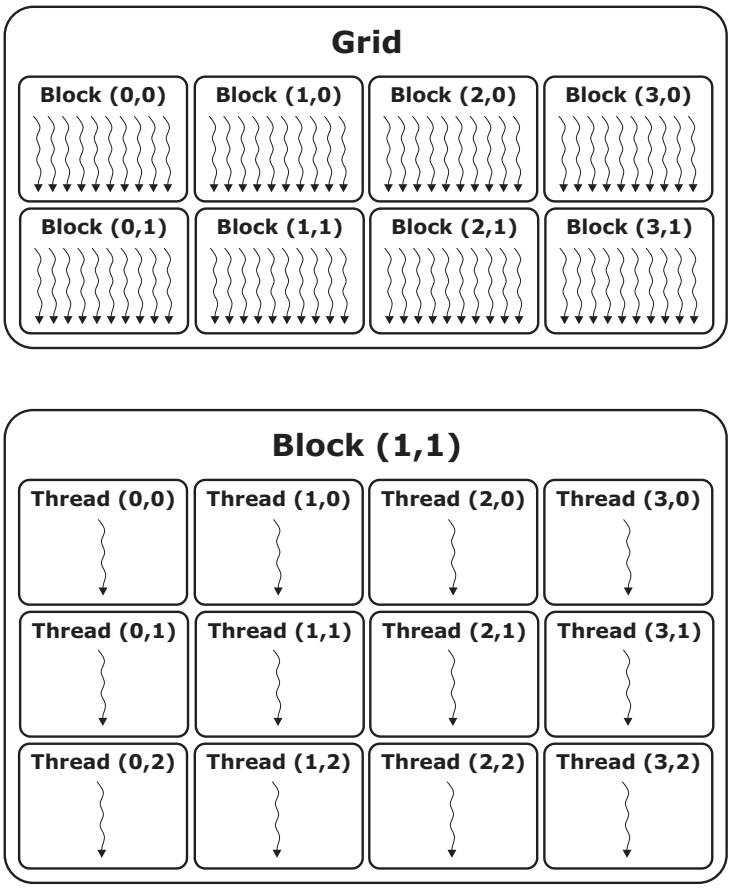

Figure 5.6: Structure of the grid, the blocks and the threads

5.4.4 Compiler

Normally, the source code which is compiled with nvcc is a combination of host and device code. The basic workflow of nvcc is to separate the host code from the device code. The host code can be compiled

with any complier because it is entirely executed on the host. Device code is either written in PTX or in CUDA C and is further compiled into binary code. [NVI10c, pp. 15-16]

5.4.5 Heterogeneous Programming

As mentioned in Section 5.4.2, the kernel invocation is the entry point to the device. The fact that host and device are two physically separated entities makes it possible that after the kernel invocation the host part of the program executes in parallel on the CPU. To facilitate this concurrent execution on host and device, the following functions have to be asynchronous:

- Kernel launches.
- Functions which perform memory transactions and are suffixed with the keyword *Async*.
- Functions which perform device to device memory transactions.
- Functions which set memory.

It has also to be mentioned that if a synchronous function is called, control is not returned to the host thread until the device finishes its execution. Host and device maintain their own memory space in DRAM, addressed to as host memory and device memory. [NVI10c, pp. 11-12, p.34]

5.5 Hardware Limitations

To comprehend all the design decisions made in the case study, it is paramount to understand the hardware limitations concerning the used device. In this section an overview of the most crucial limitations in view of computation efficiency is given.

5.5.1 Memory Latency

Memory latency describes the number of clock cycles needed for a warp to be ready to execute its next instruction. To cover up all latencies, and achieve full utilization of the streaming multiprocessor, it is necessary that the warp scheduler has always an instruction ready to issue for another warp and for any clock cycle during the latency period. In other words full utilization is achieved if the latency of each warp is hidden by other warps. The number of instructions required to fully hide latency depends on the instruction throughput. The reason for a warp not being ready to execute its next instruction is almost always that the instruction's input operand or input operands are not available at this point in time.

It is also possible that latency occurs although all operands reside in registers and are therefore by nature low-latency because of their on-chip nature. This type of latency is always caused by register dependency. The latency equals the execution time of the previous instruction, which is dependent on the instruction itself.

If at least one input operand resides in off-chip memory, the latency is between 400 and 800 clock cycles. The number of warps required to achieve full utilization in that case, is dependent on the kernel code. The ratio between instructions with on-chip memory operands and instructions with off-chip memory operands determines the number of warps required for full utilization. This ratio is commonly called the arithmetic intensity of the program.

The following formula calculates the number of warps necessary to fully hide latency:

$$W = \frac{l}{ai * ii} \tag{5.1}$$

- W is the number of active warps necessary on the streaming multiprocessor
- l is the latency of input operand residing on off-chip memory
- ai is the arithmetic intensity of the program
- ii equals the required clock cycles to issue one arithmetic instruction. For devices with compute capability 1.x ii is 4. For devices with compute capability 2.0 ii is 2.

Other common reasons a warp is not ready to execute its next instruction, are memory fences and synchronization points. A synchronization point can cause a streaming multiprocessor to idle, if only one active block is executed on the streaming multiprocessor. It helps to have two or more active blocks on the streaming multiprocessor, because synchronization points affect only warps within the block. [NVI10c, pp. 82 - 83]

5.5.2 Constraints in View of Available Registers and Shared Memory

The number of warps and blocks a streaming multiprocessor is able to host for a given kernel are dependent on the following criteria:

Execution Configuration of a Function Call

The execution configuration specifies the size and the dimensions, of the grid and the blocks, which will be used to compute the function on the device. [NVI09c, p. 111]

Memory Recourses of the Device and Subsequently of the Streaming Multiprocessor

The general specifications of devices and their streaming multiprocessors are dependent on the compute capability of the device itself. For example the maximum x-dimension or y-dimension of a grid is 65535 for all so far existing devices. On the other hand the maximum threads a streaming multiprocessor is able to host is 768 for compute capability 1.0 and 1.1, 1024 for compute capability 1.2 and 1.3 and 1536 for compute capability 2.0. [NVI10c, p. 83, p. 140]

Resource Requirements of a Kernel

The number of warps and blocks a streaming processor is able to host and process for a given kernel is dependent on the recourses (registers and shared memory) of the streaming multiprocessor itself and the recourses used by the kernel. If to execute one block, a kernel uses more registers or more shared memory than available, the kernel will fail to launch. [NVI10c, p. 78-83]

The total number of warps W_{block} in one block is as follows:

$$W_{block} = ceil(\frac{T}{W_{size}}, 1) \tag{5.2}$$

- T is the number of threads per block.
- W_{size} is the warp size.
- $ceil(x, y)$ equals x rounded up to the nearest multiple of y.

The total number of registers R_{block} allocated for one block is as follows:

For devices of compute capability 1.x:

$$R_{block} = ceil(ceil(W_{block}, G_W) * W_{size} * R_k, G_T) \tag{5.3}$$

For devices of compute capability 2.0:

$$R_{block} = ceil(R_k * W_{size}, G_T) * W_{block} \tag{5.4}$$

- G_W is the warp allocation granularity, equal to two for compute capability 1.x only.
- R_k is the number of registers used by the kernel.
- G_T is the thread allocation granularity, equal to 256 for devices of compute capability 1.0 and 1.1, 512 for devices of compute capability 1.2 and 1.3 and 64 for devices of compute capability 2.0.

The total amount of shared memory S_{block} in bytes allocated for a block is as follows:

$$S_{block} = ceil(S_k, G_S) \tag{5.5}$$

- S_k is the amount of shared memory used by the kernel in bytes.
- G_S is the shared memory allocation granularity, which is equal to 512 for devices with compute capability 1.x and 128 for devices with compute capability 2.0.

5.5.3 Constraints in View of Shared Memory Usage

Shared memory is a special type of on-chip memory and can only be used in computational mode and not in graphics mode.

Its advantages compared to other types of memory are:

- Fast on-chip memory with low latency.
- Can be accessed in form of an array in contrast to registers.
- Its values can be shared between different threads within a block.

To use all these advantages and achieve high bandwidth certain access criterions must be adhered. Shared memory is segmented into n equally sized memory banks, which can be accessed in parallel. For devices with compute capability 2.0 n is 32, which represents the number of threads in a warp. For devices with compute capability 1.x n is 16, which represents the number of threads in a half-warp. The n memory requests are only executed simultaneously if each thread in a warp or half-warp accesses elements in the n different banks, or all threads access the same element.

If x addresses of a memory request fall into the same bank the requests for this bank have to be serialized into x separate memory transactions. The bank with the highest x determines the bandwidth of the whole transaction and causes a so called x-way bank conflict.

To avoid or at least minimize bank conflicts it is important to know how memory addresses map to memory banks and how to schedule these memory requests optimally. Shared memory banks are organized in such a manner that consecutive 32-bit words are assigned to consecutive memory banks. Each memory bank has a bandwidth of 32-bits per clock cycle. To avoid bank conflicts it is often advantages to align and pad the memory. [NVI10c]

5.5.4 Register Spillage

Normally auto-variables reside in register space. In the following cases the compiler likely places these variables into local memory:

- Arrays for which the compiler cannot determine that they are indexed with constant quantities at compile time.
- Large structures or arrays which would consume too much register space
- Any variable if the kernel uses more registers than available

Local memory is an off-chip memory and therefore has the same high latency and low bandwidth as global memory accesses. It has the same requirements as global memory in view of coalesced memory access. One difference between local memory and global memory is however that local memory is organized in such a manner that consecutive 32-bit words are accessed by consecutive thread IDs. These memory accesses are therefore fully coalesced as long as all threads n a warp access the same relative address. For example the same index in an array or the same member in a structure.

One way to find out if auto-variables or a memory structure is placed into local memory is to inspect the PTX code. Even if the inspection of the PTX code reveals that no placement into local memory has

taken place it is not a guarantee that in subsequent compilation phases the compiler might still do that. To get information of the local memory usage of the cubin object, it is possible to instruct the compiler to report the total local memory usage after the compilation.

Another crucial point is that the arithmetic intensity of the program changes with the register spillage and therefore full utilization of the device is not guaranteed anymore. [NVI10c, p. 87-88]

5.5.5 Loop Unrolling

By default the compiler unrolls small loops with a known number of iterations. To control loop unrolling of any given loop the **#pragma unroll** directive is used. For small loops which are called often and/or run through often, loop unrolling can result in reducing execution time. [NVI10c, p. 131]

5.5.6 Thread Divergence

All threads within a warp have to execute the same instruction in order to compute all instructions in parallel. If within a warp different instructions have to be computed at the same time, the computation happens sequentially. This is called thread divergence because the execution of different instructions at the same time always requires branching in the code. These divergence problems have a significant impact on the computational time. [FSYA07]

Control flow instructions like **if**, **switch**, **do**, **for** and **while** are able to significantly alter the instruction throughput and in further consequence the execution time by causing threads of the same warp to follow different execution paths. This is called thread divergence and has the effect that all different execution paths have to be serialized. After all different execution paths complete their tasks the threads converge back to the same execution path. The worst case scenario would be if all threads within a warp diverge and their paths had to be serialized. Such a scenario would result in a significant increase of execution time.

One possibility to avoid divergence for **if** and **switch** statements is branch predication. By using branch predication all instructions are evaluated and therefore there is no divergence for these statements. All instructions are provided with a per-thread predicate which is set true or false based on the controlling condition. All instructions are getting issued for execution but only those who are marked with a true predicate are getting evaluated. The compiler replaces a branch instruction with a predicated instruction only if the number of instructions controlled by the branch condition is less or equal to 7 for warps that are likely to diverge and 4 otherwise. [NVI10c, pp. 92 - 93]

5.5.7 Coalesced Memory Access

Global memory and local memory are the two types of memory which profit from coalesced memory access. If certain access requirements are fulfilled the device is able to coalesce loads and stores for threads of a half warp for devices with compute capability 1.x and threads of a warp for devices with compute capability 2.0 to one memory transaction. Figure 5.7 depicts the coalescing of a half warp, which consist of 16 threads, with 4-byte words. In this case the global memory is aligned in 64-byte rows. [NVI10b, pp. 19 - 26]

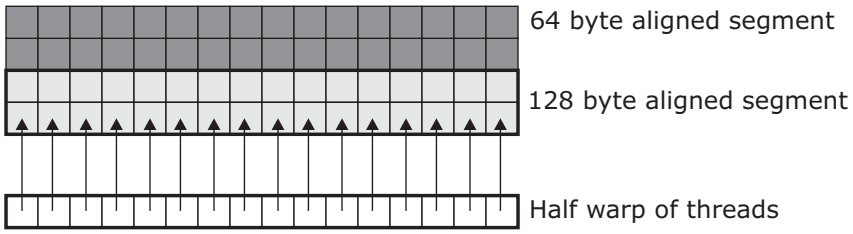

Figure 5.7: Coalesced sequential address that fall within two 128-byte segments

The requirements for coalesced memory access are dependent on the compute capability of the device:

- For devices with compute capability 1.0 and 1.1 coalesced memory access is possible if the k-th thread in a half warp accesses the k-th element in a segment which is aligned to 16 times the size of the elements accessed.
- For devices with compute capability 1.2 and 1.3 coalesced memory access is possible for any access pattern which fits into a segment size of 32 bytes for 8-bit words, 64 bytes for 16-bit words, or 128 bytes for 32- and 64-bit words
- For devices with compute capability 2.0 memory access is coalesced into the minimum number of L1-cache-line-sized aligned transactions necessary to satisfy all threads.

In view of the case study in Chapter 6 especially access criterions of devices with compute capability 1.3 are of prime importance. The size of the aligned memory segment is always 32 bytes for 8-bit data, 64 bytes for 16-bit data and 128 bytes for 32-, 64- and 128-bit data.

The exact procedure for memory transactions of devices with compute capability 1.3 is as follows:

- Find the thread with the lowest thread ID which is active in the half warp.
- Find memory segment by inspecting the requested address of the given thread
- Mark all active threads requesting memory within the same segment
- Execute the transaction and mark all serviced threads as inactive.
- Repeat until all threads in the half warp are serviced.

Effects of Misaligned Accesses

Misaligned accesses have the consequence that requested data resides in two segments as shown in Figure 5.8.

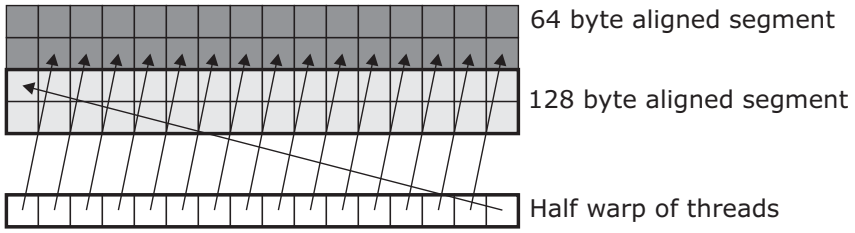

Figure 5.8: Misaligned sequential address that fall within two 128-byte segments

To get all the requested data, two transactions are necessary for devices with compute capability 1.2 and 1.3 and 16 transactions are necessary for devices with compute capability 1.0 and 1.0. Figure 5.8 shows the bandwidth development of two devices with different offsets.

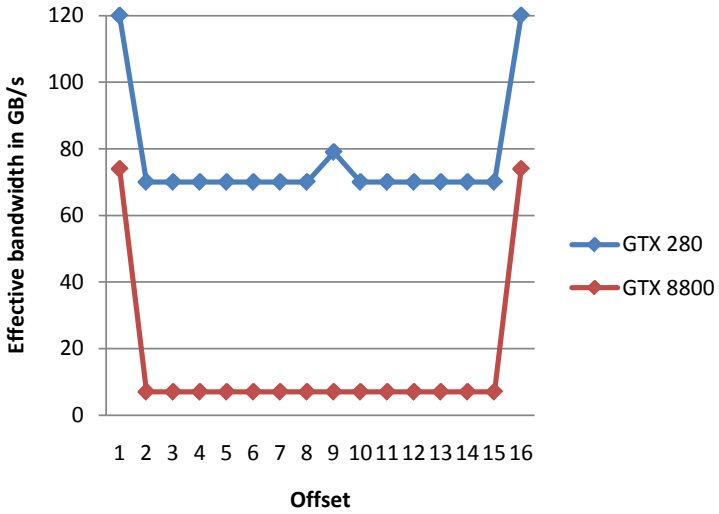

Figure 5.9: Bandwidth of memory access with increasing offset

Effects of Strode Accesses

If the access pattern displays a stride access performance can degrade as higher the stride is. This kind of access pattern occurs often in the case of multidimensional data or multidimensional matrices. Figure 5.10 depicts an access pattern with a stride of two.

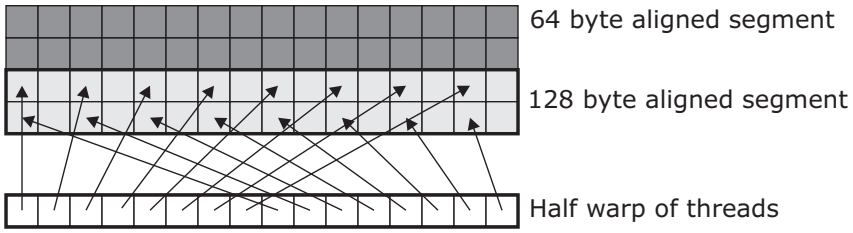

Figure 5.10: A half warp accessing memory with a stride of 2

As shown in Figure 5.10 a stride of 2 results in a single transaction, but half of the elements are not used which also cuts the bandwidth in half. The behavior of the bandwidth with increasing strides can be seen in Figure 5.11.

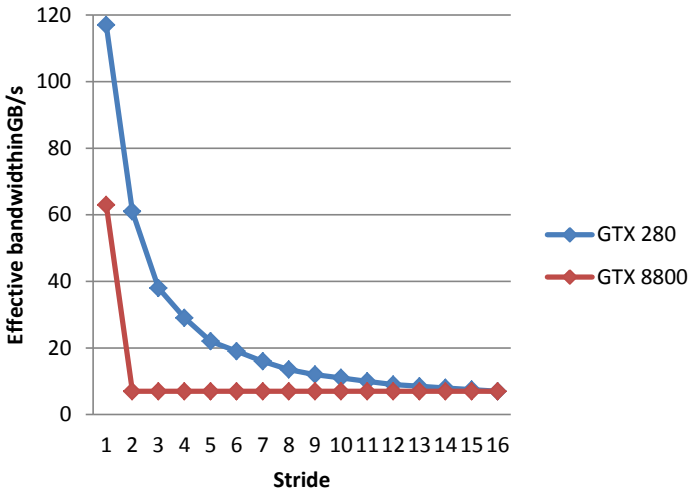

Figure 5.11: Bandwidth of memory access with an increasing stride

Chapter 6

Case Study

In principal the reason for the case study is to determine if GPUs are suitable to efficiently solve problems displaying structures typical to shortcut collision attacks. It is of importance to define efficiently in this context. On one hand efficient could mean that an attack is computed faster on a GPU as on a CPU. On the other hand efficient could mean that the implementation of a typical shortcut attack is able to fully or at least almost fully utilize the GPU. Both points of view are eligible and will be addressed in the following sections.

As case study serves a variant of the at the time most efficient short cut collision attack on SHA-1 regarding complexity, which is described in Section 3.3.7. This attack features typical structures present in most collision attacks and is therefore also a suitable representative for other attacks.

6.1 General Considerations

The first step is to decide which type of platform is a good choice for the case study. In principal ATI, NVIDIA and the Cell Multiprocessor are the three platforms which are widespread enough to be considered as candidates. All three are based on the stream programming model and are therefore in principal suitable. As Figure 4.6 in Section 4.5.4 depicts the Cell Multiprocessor was introduced in the year 2005 and is therefore at least in view of computational power not competitive against state of the art GPUs from ATI and NVIDIA. In contrast to ATIs Stream, NVIDIAs CUDA is a very widespread and commonly used tool. Therefore CUDA is the platform of choice for this case study.

A typical collision attack works in a manner that it computes one or more steps at a time. After that it checks conditions and reacts accordingly to their evaluation. After the evaluation further steps are computed or the algorithm finishes. This results in a non-deterministic behavior in view of program flow. On stream processor architectures this results in thread divergence as described in Section 5.5.6.

Another problem is that due to the non-deterministic behavior also memory read and memory write accesses occur randomly. Random memory accesses in CUDA especially in global memory and local memory are a problem in view of memory throughput as shown in Section 5.5.7.

There are different strategies to address these problems. The following sections show three different approaches. Each approach has its own structure and design principle. Therefore the benefits and tradeoffs are different also. Important is to find out the ”efficiency” of each particular approach. Also the limitations and their impact on the efficiency are very important to know. One important goal of this

case study is to provide guidelines. These guidelines can provide assistance to which strategy is suitable for which sort of problem.

6.2 Technical Specifications and Benchmarks

On the host side the used CPU is based on Intel's Nehalem architecture which was introduced in November 2008 [Int10a]. The exact model is "Core i7-920" which has 4 cores and a clock speed of 2.66 GHz. The device is a NVIDIA GeForce GTX 295 which is a dual-GPU graphics card and contains two independent PCBs or printed circuit boards. Each board contains a GT200b GPU and 896 MB GDDR3 memory. Table 6.1 shows the technical specifications of the NVIDIA GeForce GTX 295.

Table 6.1: Technical specifications of the NVIDIA GeForce GTX 295 graphics card

	GeForce GTX 295
Fabrication Process	55 nm
CUDA Cores	480 (240 per GPU)
Number of multiprocessors	30
Graphics Clock	576 MHz
Processor Clock	1242 MHz
Memory Clock	999 MHz
Standard Memory Config	1792 MB GDDR3 (896MB per GPU)
CUDA Capability	1.3
Constant memory	64 KB
Shared memory per multiprocessor	16 KB
Registers available per multiprocessor	16K 32-bit registers (64 KB total)
Warp size	32
Maximum number of threads per block	512
Maximum number of resident blocks per multiprocessor	8
Maximum number of resident warps per multiprocessor	32
Maximum number of resident threads per multiprocessor	1024

The specifications of the GPU have a high impact on the implementation itself. One important key figure is the occupancy rate which describes the ratio between the number of warps residing on a streaming multiprocessor and the maximum number of warps residing on a streaming multiprocessor. The number of resident warps is dependent on the register usage, the shared memory usage of a single thread and the number of threads per thread-block. Devices with compute capability 1.3 such as the GeForce GTX 295 are able to host up to 1024 threads per streaming multiprocessor. To achieve full occupancy, one thread is allowed to use only 16 32-bit registers and 32 byte of shared memory. A "low" occupancy rate is not necessary equal to a "low" performance. The necessary occupancy rate to achieve full performance is dependent on the arithmetic intensity of the program which is described in Section 5.1.

6.3 Cost-Performance Ratio

It is clear that different setups of GPUs and GPUs produce different results. For example a setup of a high-end CPU with a low-end GPU would produce a much lower performance ratio in contrast to a setup of a low-end CPU with a high-end GPU. From that it follows that the conducted benchmarks have also to be viewed with this cost factor in mind.

There are many different approaches to calculate the costs of GPUs and CPUs. One approach would be to compare the power consumption of these devices. Another would be to compare acquisition costs. For the purpose of the case study the acquisition costs were chosen. This decision was made mainly because of the fact that these costs are accessible before the point of acquisition whereas the costs for power consumption are only accessible at execution time and only with additional equipment.

Another problem is which components to take into account. The motherboard can be seen as a basis for the CPU and the GPU. Both CPU and GPU make use of fast outside memory which in case of the GPU is placed on the graphics card. In case of the CPU this outside memory is the main memory. So the components chosen to calculate the cost ratio are the graphics card on the side of the GPU and the CPU in combination with the main memory on the side of the CPU. Also the necessary periphery like the casing or the hard-disk has to be taken into account. The costs of the parts, which can not be assigned to the CPU or the GPU, are split in half and added to each side.

At the time of purchase the acquisition costs of the components used in the case study were as follows:

- Graphics card: 440€
- CPU: 275€
- Main memory: 120€
- Motherboard: 242€
- Periphery: 230€

On the side of the CPU this totals in 631€. On the side of the GPU this totals in 676€. This results in a performance-cost ratio of approximately 0.93. So for the best performance ratio of 37 calculated in Section 6.9.2, the cost-performance ratio would be 34.5. One important fact is that motherboards are able to support several graphics cards which benefits the side of the GPU.

6.3.1 Benchmarking Methods

Benchmarks are conducted in two different ways. One way is to conduct benchmarks with the so called ”NVIDIA Compute Visual Profiler”. The other way is to use the CUDA built-in timer functions. Listing 6.1 shows the necessary functions to measure the execution time of a kernel. The timer functions are self-explanatory. Important is also to call the **cudaThreadSynchronize()** routine after the kernel call. Because of the non-blocking nature of the CUDA kernels statements after the kernel invocation can be computed before the kernel finishes its computation.

```
1  unsigned int timer;
2  cutCreateTimer(&timer);
3  cutStartTimer(timer);
4
5  kernel<<<dimGrid, dimBlock>>>(...);
6
7  cudaThreadSynchronize();
8  cutStopTimer(timer);
9  printf("Execution time: %f ms\n", cutGetTimerValue(timer));
10 cutDeleteTimer(timer);
```

Listing 6.1: CUDA timer

6.3.2 Dual Issue

To fully utilize the capabilities of the GPU an understanding of how instructions are issued and computed is paramount. Each streaming multiprocessor has the help of so called special function units or short SFUs. The special functions units are among other things able to perform so called MUL[1] instructions. For that reason the processing cores of the GeForce GTX 200 GPUs are able to perform dual-issue of multiply-add operations or short MAD[2] and MUL operations by using the streaming processors MAD unit to perform a MAD operation and using the SFU to perform a MUL operation in one clock cycle. Benchmark tests conducted in Section 6.3.4 however show that the extra MUL operation is not always available [NVI08].

6.3.3 Theoretical Benchmarks

The first approach to compare the performance ability of GPUs and CPUs are theoretical benchmarks. With regard to theoretical benchmarks of GPUs and CPUs in general, Section 4.5.4 gives a good view over a certain timespan. For the actual used device the theoretical benchmarks are as follows:

- NVIDIA GeForce GTX 295: 1.242 * 3 * 240 = 894 Gflops per GPU (MUL and MAD)
- NVIDIA GeForce GTX 295: 1.242 * 2 * 240 = 596 Gflops per GPU (MUL or MAD)
- NVIDIA GeForce GTX 295: 1.242 * 1 * 240 = 299 Gflops per GPU (integer operations or only floating-point operations which are not MUL or MAD)
- Intel Core i7-920: 42.56 Gflops [Int10b]

6.3.4 Synthetic Benchmarks

An alternative option is, to conduct so called synthetic benchmarks, to compare the performance ability of GPUs and CPUs. In the case of CPUs synthetic benchmarks are a very well explored field. Prominent members of synthetic benchmarks for CPUs are Dhrystone [Wei84] and Whetstone [CW76]. Neither Dhrystone nor Whetstone are able to benchmark GPUs. The available tools which benchmark GPUs and CPUs in a comparable manner are immature in a sense that they produce results which are not very plausible given the results in Section 6.3.3 and 6.3.5

[1] MUL: c = a * b
[2] MAD: c = a * b + a

The fact that synthetic benchmarks are important in view of the case study gave the reason to self-conduct benchmarks. As mentioned in Section 6.3.3 the NVIDIA GeForce GTX 295 has different benchmarks dependent on the instructions it has to compute. The first basic classification is to differentiate between integer computation and floating-point computation. A further classification is to distinguish between multiplications, additions, bitwise operations and the special cases MUL and MAD. Listing 6.2 shows the complete function for the floating-point-addition benchmark.

In line one there is an allocation of shared memory. This allocation in combination with the instructions in line 6 and 15 is just to ensure that the compiler is not optimizing the other instructions out of the function. The instruction **#pragma unroll** in line 9 is responsible to unroll the loop to ensure that there is no instruction overhead due to loop iterations. The sole purpose of the loop in line 10 is to focus the main computational load on the benchmark instructions.

The benchmark instruction itself is shown in line 12. It consists of two ”simple” floating-point operations which according to Table 6.2 requires two instructions to compute. Table 6.2 also shows that these kind of instructions reach about 280 Gflops which is close to the theoretical benchmark of 299 Gflops shown in Section 6.3.3.

Listing 6.3 shows the operations to benchmark the MAD capabilities of the device. Results in Table 6.2 show that for the multiplication and the addition only one instruction is issued. Data in the table also shows that there is an increase in speed of ”only” 72% where there should be an increase of 100%. Because of the fact that NVIDIA itself doesn't disclose all necessary data it is not possible to evaluate the exact reason for this discrepancy.

As mentioned in Section 6.3.2 GeForce GTX 200 GPUs are able to compute an extra MUL operation with the help of the SFUs. Table 6.2 shows that in about 51% of all cases the extra MUL instruction can be performed by the SFU which results in a ”peak performance” of 606 Gflops.

With regards to the case study additions and bitwise integer operations are of most importance. According to Table 6.2 the integer benchmark function displays similar behavior than the ”simple” floating-point benchmark function. This is can be explained because the SFU is only available for MUL operations and the streaming processors MAD units can't be applied either.

Table 6.2: Synthetic benchmarks for one GPU of the NVIDIA GeForce GTX 295 graphics card

method	gputime	instr	ops / lcycle	instr / lcycle	instr / op	Gips	Gflops
kflopMAD()	16993.8	4500712	4	2.01	0.50	264.84	482.06
kflopMULMAD()	10209.2	4466397	3	2.00	0.67	437.49	601.81
kflopMULMADMAD()	13517.9	6694860	4	2.99	0.75	495.26	606.01
kflop()	14609.5	4499789	2	2.01	1.01	308.00	280.37
kiop()	14759.8	4470882	2	2.00	1.00	302.91	277.51
kiopMULMAD()	77355.9	22494684	4	10.06	2.51	290.79	105.90

```
1  __shared__ float result[BLOCK_SIZE];
2
3  __global__ void kflop()
4  {
5
6    float a = result[threadIdx.x];
7    float b = 1.01f;
8
9    #pragma unroll
10   for(int i = 0; i < 1024; i++)
11   {
12     a = b + a + b;
13   }
14
15   result[threadIdx.x] = a+b;
16 }
```

Listing 6.2: kflop()

```
1      a = b * a + b;
```

Listing 6.3: kflopMAD()

```
1      a = b * a + b;
2      a = a * b;
```

Listing 6.4: kflopMULMAD()

```
1      a = b * a + b;
2      a = a * b;
3      a = a * b;
```

Listing 6.5: kflopMULMADMAD()

```
1      a =  b & a + b;
```

Listing 6.6: kiop()

```
1      a = b * a + b;
2      a = a * b;
3      a = a * b;
```

Listing 6.7: kiopMULMAD()

A complete listing of all benchmark functions is shown in Appendix A.1

6.3.5 Application Based Benchmarks

In the implementation of the attack the state update is the most computational intensive part. The state update function has a very simple structure and therefore can be easily optimized for GPU usage. Measuring unit is the computation of one message block.

The benchmark ratio between CPU and GPU presents a rough indication in a sense that it can be seen as an upper bound for the three different approaches of the case study. It is also a strong indicator how well suited the GPU architecture is in general to solve problems which present that kind of structure.

The first consideration is the memory consumption of the implementation:

- W: 16 32-bit variables
- Chaining variables: 5 + 1 32-bit variables

The memory requirements for the algorithm itself are 22 32-bit variables. The compilation flag –*ptxas-options*=*-v* shows the memory consumption per thread. For an occupancy rate of 1, which means that 32 warps are active, the maximum number of registers a thread is able to use is 16. A lower occupancy rate does not necessarily result in lower performance. The occupancy rate however is one of the indicators how effectively memory latencies can be hidden. Further details are given in Section 5.5.1.

SHA-1 GPU Implementation With Registers

A native approach is to put all variables into the registers. Table 6.3 shows that one thread needs 24 32-bit registers. At a block size of 512 the occupancy rate is 0.5 which means that 16 warps are active on a streaming multiprocessor. One key figure is the ratio between the GPU implementation and the CPU implementation which is at about 60. Also important are the 277 Giops which are equal to the synthetic benchmark results in Section 6.3.4. This, and the fact that Gops and Giops are very similar, indicates that the implementation uses its hardware capabilities nearly optimal.

Table 6.3: Key data about a SHA-1 GPU implementation running on the NVIDIA GeForce GTX 295 with the help of registers

regs / thread	time gpu micro sec	instructions	active warps	occupancy	ratio	Gops	Giops
24	13016	3947598	16	0.5	60	303.30	277.36

SHA-1 GPU Implementation With Shared Memory

To decrease the register usage per thread it is possible to transfer some variables into the shared memory. The first approach is to put the chaining variables into the shared memory. As Table 6.4 shows 3 registers can be saved by using the shared memory. The 289,84 Gops indicate that there is no idle time for the streaming processors itself. A Giops count of 160,02 on the other hand shows that the implementation is not ideal in sense of runtime. The data in Table 6.4 also shows that this implementation needs approximately 66% more instructions than the implementation in Section 6.3.5 uses. This is probably be due to shared memory load and store operations. It is also the reason why the Gops count is nearly optimal while the Giops count drops significantly.

The second approach is to put W into shared memory. Table 6.4 shows that 11 registers can be saved due to putting W into the shared memory. The usage of ”only” 13 registers per thread would imply a full occupancy rate only taking the register usage into account. On the other hand the amount of memory each thread uses in the shared memory space is 64 bytes. With 16KB in total available in the shared memory this results in 256 active threads per streaming multiprocessor. 256 active threads are the equivalent of 8 warps and an occupancy rate of 25%. Normally it would be a problem to hide memory latency with an occupancy rate of only 25%. In this case there is no usage of global memory and therefore no latency to hide. Table 6.4 depicts also that the Giops in both cases are similar. The slight discrepancy is probably due to the instruction overhead of using more shared memory. Two results are bracketed. This is because in that case the visual profiler produced obviously wrong results. The reasons for the wrong results could not be determined.

Table 6.4: Key data about different SHA-1 GPU implementations running on the NVIDIA GeForce GTX 295 with the help of shared memory

function	regs / thread	time gpu micro sec	instructions	shared mem / block	active warps	ratio	Gops	Giops
shared_cv	22	22559.5	6538576	10264	16	34.57	289.84	160.02
shared_W	13	11752.2	(540790)	16344	8	32.87	(46.02)	152.99

SHA-1 GPU Implementation With Local Memory

Another way to reduce the register usage per thread is the usage of local memory. Local memory is part of the global memory space. In order to sustain the maximum instruction throughput the arithmetic intensity has to be high enough to mask memory latencies. Table 6.5 shows the key data of the same SHA-1 implementation as described in Section 6.3.5 but each time with a different number of registers used. Figure 6.1 shows a steady increase in Giops conformal to the increase of register usage. The same Figure also shows that at a usage of 21 registers the implementation reaches the maximum instruction throughput. This happens because of the fact that at the usage of 21 registers and three local memory variables the arithmetic intensity in combination with the occupancy rate is high enough to mask latencies in local memory accesses.

Table 6.5 also shows that the number of instructions increases steady with additional usage of local memory. This explains also the gap between Gops and Giops. Because of the fact that the case study has the same structure with regards to the most computational intensive parts, it stands to reason that its runtime behavior follows the same path as pictured in Table 6.5 and Figure 6.1.

Table 6.5: Key data about different SHA-1 GPU implementations running on the NVIDIA GeForce GTX 295 with the help of local memory

regs / thread	time gpu micro sec	local mem	lmem access	instructions	ai	necessary warps	ratio	Gops	Giops
4	194514	4564	24512352	6189953	0.76	132.00	4	31.82	18.56
5	185109	4220	23035744	6090700	0.79	126.07	4	32.90	19.50
6	162988	4036	21091104	5838013	0.83	120.42	5	35.82	22.15
7	115373	2012	13272992	5360213	1.21	82.54	7	46.46	31.29
8	99402	1904	11707392	5165749	1.32	75.54	8	51.97	36.32
9	83533	1664	9973696	4958770	1.49	67.04	9	59.36	43.22
10	61952	1336	7729120	4700359	1.82	54.81	13	75.87	58.27
11	54852	1336	7231424	4615204	1.91	52.23	14	84.14	65.81
12	47329	1320	6608480	4517139	2.05	48.77	16	95.44	76.27
13	44892	1316	6404128	4481474	2.10	47.63	17	99.83	80.42
14	42151	1304	6147040	4442793	2.17	46.12	18	105.40	85.64
15	38016	1288	5853696	4398958	2.25	44.36	20	115.71	94.96
16	35420	1264	5550464	4357054	2.35	42.46	22	123.01	101.92
17	29505	1228	5174720	4315736	2.50	39.97	26	146.27	122.35
18	25611	1088	4627584	4277083	2.77	36.06	30	167.00	140.96
19	21205	912	3829952	4216835	3.30	30.28	37	198.86	170.24
20	16176	700	2900480	4143420	4.29	23.33	48	256.15	223.17
21	13121	472	1971008	4080908	6.21	16.10	59	311.02	275.13
22	12976	256	1061312	4013249	11.34	8.82	60	309.28	278.21
23	12886	4	16480	3940656	717.35	0.14	60	305.82	280.16

Interpretation of Results

In general it is best to only use registers if possible. High register usage often results into low occupancy rates. This is negligible if there is no latency to hide. The usage of shared memory is also low latency but results in additional instructions. These additional instructions are probably the result of additional load and store instructions. The usage of local memory or global memory is only advisable if the arithmetic intensity is high enough to hide the occurring latency.

6.4 General Implementation Considerations

There is an existing implementation of the attack which runs very efficient on CPUs. The program runs as a single thread. The first thing to do is to convert the sequential program flow into a parallel program flow. There are many different ways to do this. One solution is to give each thread a different data set to compute. Another solution is to give each thread a step to compute. Both scenarios are implemented in the following sections.

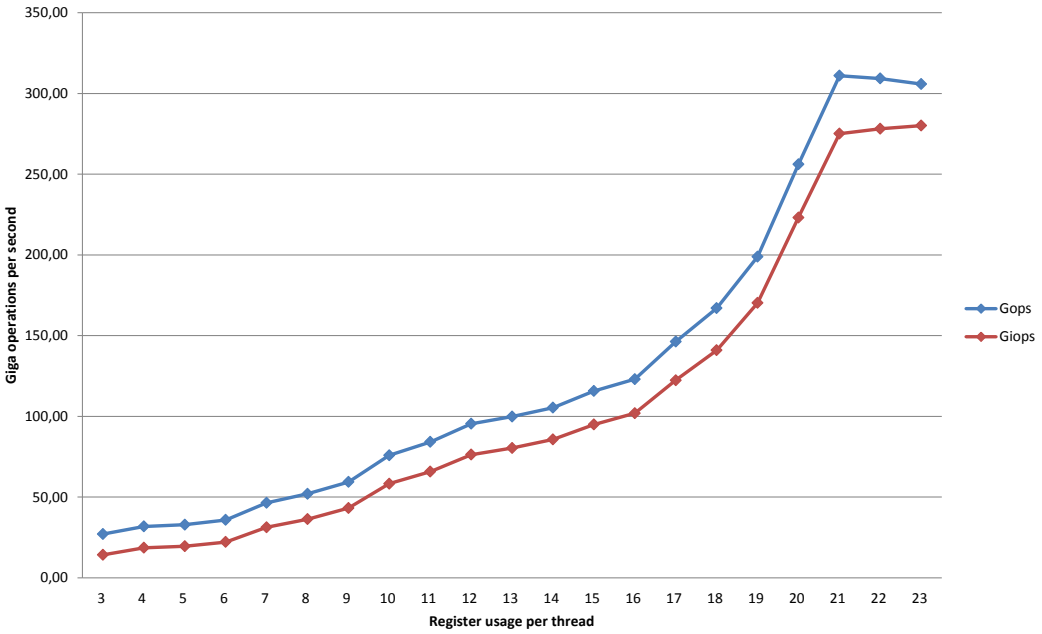

Figure 6.1: Different runtime behavior dependent on register usage per thread and usage of local memory per thread

6.4.1 Preparation of Input Data

The different messages are forming the input data. Each message is composed of 16 32-bit message words. The message words with the lower indices are ”fixed”. There are no degrees of freedom in the ”fixed” message words. In other words, all fixed message words are similar for each data set. Implementation wise it makes sense that the ”fixed” message words are stored into constant memory.

The part of the input data which is not fixed determines the number of different data sets the algorithm has to compute. For example if 32 bits in the input message are not ”fixed” the algorithm has to compute 2^{32} different input messages.

For the two approaches described in Section 6.6 and Section 6.7 the degrees of freedom determine the thread structure. The distribution of the input data has also an influence on the efficiency of the algorithm. Tests showed that the best results are achieved when the input message is similar as long as possible in one thread block. That means that the message word with the highest index which is not ”fixed” determines the number of threads one thread block contains. The ”not fixed” message words with the second and third highest indices determine the grid structure. The remaining ”not fixed” message words are handled by the host in a way that the host starts a separate kernel for each possible remaining variation of the input message.

In case of the approach described in Section 6.8 the distribution of input data differs from the two other approaches. Also for that case, the distribution of the input data has no effect on the runtime of the algorithm.

6.4.2 Development Environment

As operating system served Microsoft Windows 7 in the 32-bit version. Microsoft Visual Studio 2008 with its Visual C++ compiler served as platform for the host code. It is possible that different versions of the CUDA compiler produce different results in terms of runtime. All benchmarks were conducted with the CUDA compiler in the 3.0.14 version.

Also some compiler flags are important for the efficient and correct computation:

- **-maxrregcount** (sets the maximum number of registers one thread has at his disposal)
- **-arch=sm_13** (the code is compiled for devices with compute capability 1.3)

6.5 Precomputed Values Approach (CPU)

The Precomputed values approach on the CPU has a very straightforward implementation structure. This boils down to a few principles:

- Precomputation of values. (For example *W[17] = S1(W[14] xor W[9] xor W[3] xor W[1])* instead of *W[step] = S1(W[step-3] xor W[step-8] xor W[step-14] xor W[step-16])*).
- Avoidance of function calls on instructions which are often executed

Tests showed that this implementation runs about two times faster than an implementation where these principals were ignored. A more detailed elaboration of the runtime behavior of these implementations is given in Section 6.9

6.6 Precomputed Values Approach (GPU)

Such a ”CPU performance optimized” code generally executes also very well on GPUs. One important condition for the implementation to run well on a GPU is that all threads follow the same path. Collision search attacks in general exhibit a non-deterministic behavior. In case of a multi-threaded environment it means that each thread follows an own path dependent on certain conditions. Different paths viewed on a deeper more hardware oriented level mean that the hardware has to issue different instructions for the threads. In CUDA different instructions in a warp at the same time equals serialization of computation. Figure 6.2 shows a possible scenario for diverging threads. Furthermore it is very likely that each thread has a different runtime.

6.6.1 Different Runtime Problem

It is very likely that threads have different runtimes. This is because of the fact that steps are computed dependent on the input data. Figure 6.3 shows the same algorithm with different input data. One box represents a computation of one step with the evaluation of the corresponding conditions. Each column represents a thread in a warp. In a single threaded environment each box would be computed in serial. The number of steps that have to be computed would be identical for the two input data-sets.

In CUDA the runtime of the algorithm is different for each data set. The total runtime of the warp is equal to the runtime of the thread which finishes last within the warp. As Figure 6.3 depicts the algorithms runtime, with input data 1, is two times as long as the algorithms runtime with input data 2.

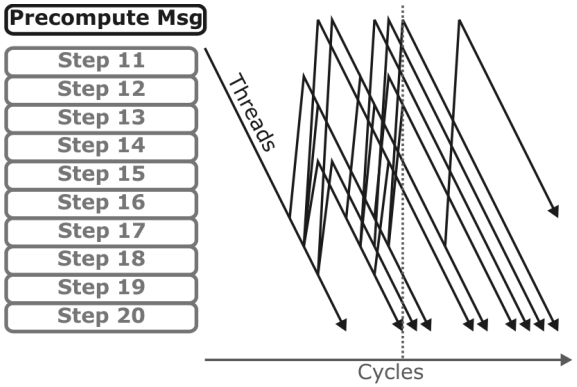

Figure 6.2: Same cycle different instructions ⇒ divergence ⇒ sequential computation

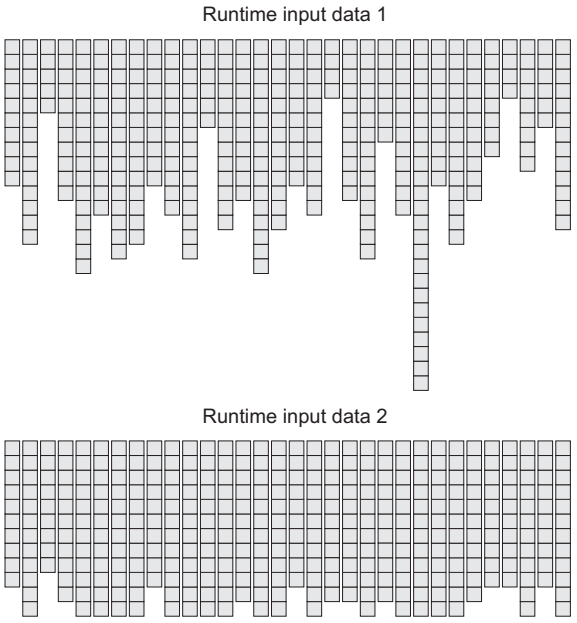

Figure 6.3: Different runtime behavior dependent on input data

6.6.2 Divergence Problem

As mentioned before CUDA capable GPUs are based on a SIMT architecture. Always 32 threads are grouped into a so called warp. In order to execute all instructions in parallel these threads have to execute the same instruction. The worst case scenario would be if every thread in the active warp computes a different instruction. As a result of that all threads would execute their instructions in serial. In the case of the CPU optimized approach the runtime behavior is tightly bound on how much the threads diverge in average. Also important is how sophisticated the hardware mechanisms are to join divergent threads together if possible.

Because of the diverging threads the hardware has to serialize certain computations. This results on one hand into idle threads and on the other hand in a different over all runtime behavior. Figure 6.4 shows the same algorithm with different input data. One box represents a computation of one step with the evaluation of the corresponding conditions. Each column represents a thread in a warp. In a single threaded environment each box would be computed in serial. The amount of steps that have to be computed is similar for the two input data-sets.

In CUDA the runtime of the algorithm is different for each data set. The total runtime of the warp is equal to the runtime of the thread which finishes last within the warp. As Figure 6.4 depicts the algorithms runtime with input data 1 is four times as long as the algorithms runtime with input data 2.

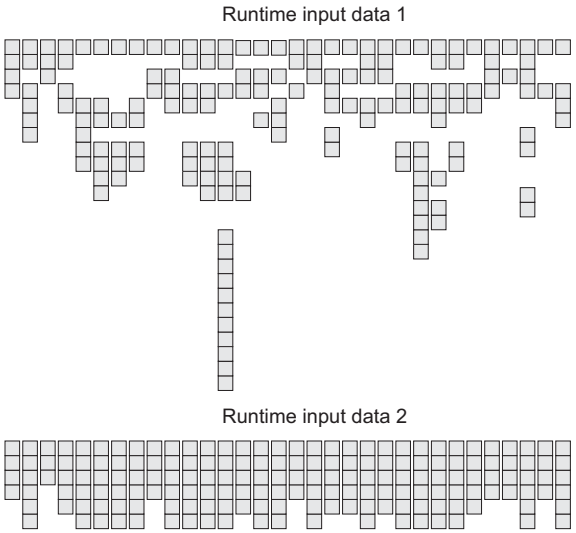

Figure 6.4: Different divergence behavior dependent on input data

6.6.3 Implementation Principles

There are different implementation principles which have a big impact on the overall runtime.

Distribution of the Input Data

As mentioned in Sections 6.6.1 and 6.6.2 the runtime of the algorithm and the degree of how much the threads diverge is dependent on the input data. In the case of CUDA, SIMT is just limited to threads within the same warp. This means that if possible the input data for threads of the same warp should be chosen in such a manner that the computational load is as uniformly distributed as possible.

In the case study the degrees of freedom in W determine the input data. Tests showed that the computational load is most uniform when the input data is chosen in such a manner that W is similar within a warp as long as possible. In concrete terms this means that W_{15} determines the size of the block. Furthermore W_0 till W_{14} is similar in each block and therefore also in each warp.

Memory Usage

The number of registers a thread uses determines the occupancy of the streaming multiprocessor. The occupancy rate again has a big influence on how good the hardware is able to hide latencies of global memory accesses. A big advantage of this approach is that the threads access the global memory only at the beginning and at the end of the computation. Therefore the occupancy rate has almost no impact on the runtime because of the fact that there is almost no latency to hide.

For this approach the best results are achieved when all variables are placed into registers. In this specific case the implementation needs 36 32-bit registers per thread. This results in an occupancy rate of 25% for devices with compute capability of 1.x and 50% for devices with compute capability 2.x.

Compared to variables, constants can be stored in three different ways. Constants can be stored in register, shared memory and constant memory. Tests showed that the usage of the constant memory achieves the best results. The better results compared to register usage are probably due to the fact that if using the registers all constants have to be loaded separately into the registers from the global memory. With the usage of shared memory the ”separate load disadvantage” would no longer be applicable. The usage of shared memory on the other hand has the tendency to generate additional instructions on the hardware. Therefore the use of constant memory is the best choice for constants

For variables, which are similar for each thread within a block, it has proven to be effective to load the values into the shared memory first. After they are stored into the shared memory the values can be distributed to the registers. This has the big advantage that each thread has to load only one value from the global memory.

6.6.4 Benchmarks and Limitations

The advantages are that this approach doesn't need any local or global memory with the exception at the start and the end of the algorithm. Therefore memory latency is not an issue. Another advantage is that all values in in the code which can be precomputed are precomputed. This means that there is no additional computational overhead.

A disadvantage is that thread divergence is an issue. For the instance of the case study there is a 38% loss of efficiency. Benchmarks showed that this approach provides an average ratio of about 37 in comparison to the fastest CPU implementation. A more precise analysis is given in Section 6.9.

6.7 Parameterized Approach

The basic idea of the parameterized Approach is to avoid thread divergence by trying to uniform the instructions as much as possible. The key for this approach to work is to decouple the input data entirely from the instructions. That means the threads only compute the same instructions over and over again, without knowing which step, dataset or condition they are computing (see Figure 6.5).

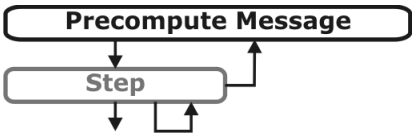

Figure 6.5: Decoupling data from instructions ⇒ almost no divergence, but additional control flow.

6.7.1 Different Runtime Problem

A big problem which still remains is that it is not predictable how much steps each thread has to compute until he has completed his task. As also mentioned in Section 6.6 if one thread in the warp takes a very long time to finish the other threads have to wait until this particularly thread finishes its computation. One possible scenario would be to give each thread more than one data set to compute. Because of the fact that the time to compute one data set should be random the overall time consumption of threads within a warp has to be more uniform. Figure 6.6 shows the thought behind this idea. The first scenario depicts an approach where each thread is responsible for one input data set. The second scenario shows an approach where each thread has a couple of input data sets to compute. With only five data sets per thread the overall execution time is not that much shorter. It is easy to imagine though that

with increasing the input data sets per thread the overall ratio in view of execution time is also increasing.

The worst case scenario would be if one thread gets always the data sets which take the longest time to compute. In this case the overall execution time would be equal to the execution time of the first scenario.

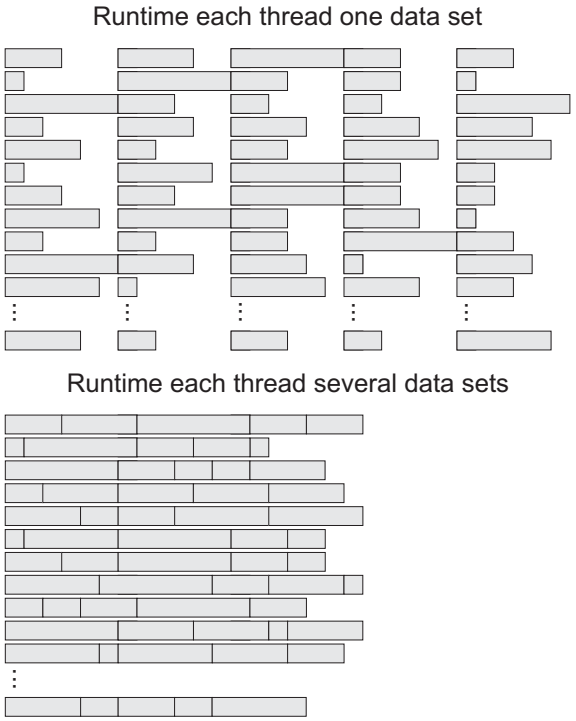

Figure 6.6: Two different execution patterns

For an implementation without divergence the second scenario has with the exception of the worst case scenario always the better overall execution time.

6.7.2 Divergence Problem

In the case of this implementation there are two possibilities for divergence to occur. First, for steps 15 and below message expansion is not an issue. Second, each step has a different number of conditions to check. The checking of the conditions is not computationally intensive. Therefore the focus in sense of runtime is on the message expansion.

Test showed that divergence occurs more often for scenarios where threads compute more than one input data set. This is probably because of the divergence in the message expansion. Furthermore the computational overhead due to divergence outweighs the benefits. Therefore the first execution pattern shown in Figure 6.6 was chosen for the implementation. For other attacks this behavior may be different and execution pattern two may be better suited.

6.7.3 Memory Usage

This implementation uses registers, shared memory, constant memory and local memory. The constant memory is used for the lookup tables. One big problem is that the evaluation of the active step is only possible at runtime. This means that the arrays which hold W and A are also indexed at runtime dependent on the currently active step. Per definition CUDA can only place arrays into register space if their indices are determined at compile time. In the case of a runtime determination of the indices the compiler spills the values of the arrays into local memory. Local memory resides on global memory and is therefore high latency. Furthermore because of the random accesses the memory transactions are not coalesced. The consequences of non-coalesced memory accesses are described in Section 5.5.7

Another problem is that the arithmetic intensity is too low to hide all memory latencies. To measure the impact on the computational time this approach was implemented in three different ways. The first implementation uses only local memory. In other words the compiler spills out both arrays to the local memory. The second implementation uses shared memory for the array which holds the values of W. The third implementation uses shared memory for the array which holds the values of A. Because of the fact that A is more often accessed the third implementation has the best efficiency. A fourth possibility is to put A and W into the shared memory space. As mentioned in Section 6.3.5 shared memory accesses are producing additional instructions. Tests showed that the third implementation delivered the best results in view of runtime. This is because the arithmetic intensity for this case is high enough to hide the occurring latencies due to memory accesses of W.

6.7.4 Benchmarks and Limitations

One limitation is the register spillage because of the indexing at runtime. Another problem is that thread divergence cannot be entirely avoided. Also there is additional overhead because certain values can only be computed at runtime which normally could be precomputed.

Benchmarks showed that this approach provides an average ratio of about 9 in comparison to the fastest CPU implementation. For the instance of the case study there is a 65% loss of efficiency. A more precise analysis is given in Section 6.9.

6.8 Kernel Scheduled Approach

The third approach avoids thread divergence completely for the computational intensive parts. This is possible by dividing the different steps into autonomous kernels. Each kernel has a list of data sets for which he is responsible at a certain time. The principal is shown in Figure 6.7. More precisely the list only consists of the starting addresses of the individual data sets.

A kernel scheduler which is controlled by the host thread is responsible to handle the kernel invocations. The kernel scheduler always looks which kernel has the largest amount of data sets to compute. This kernel is than invocated by the kernel scheduler.

Figure 6.7: No divergence, but many load and store instructions from global memory ⇒ memory latency

6.8.1 Implementation Principles

At the beginning always a so called initial kernel has to be computed. The initial kernel creates all the data sets necessary for the computation. The data sets are stored in the global memory. Figure 6.8 shows how the memory and the address lists are arranged after the computation of the initial step.

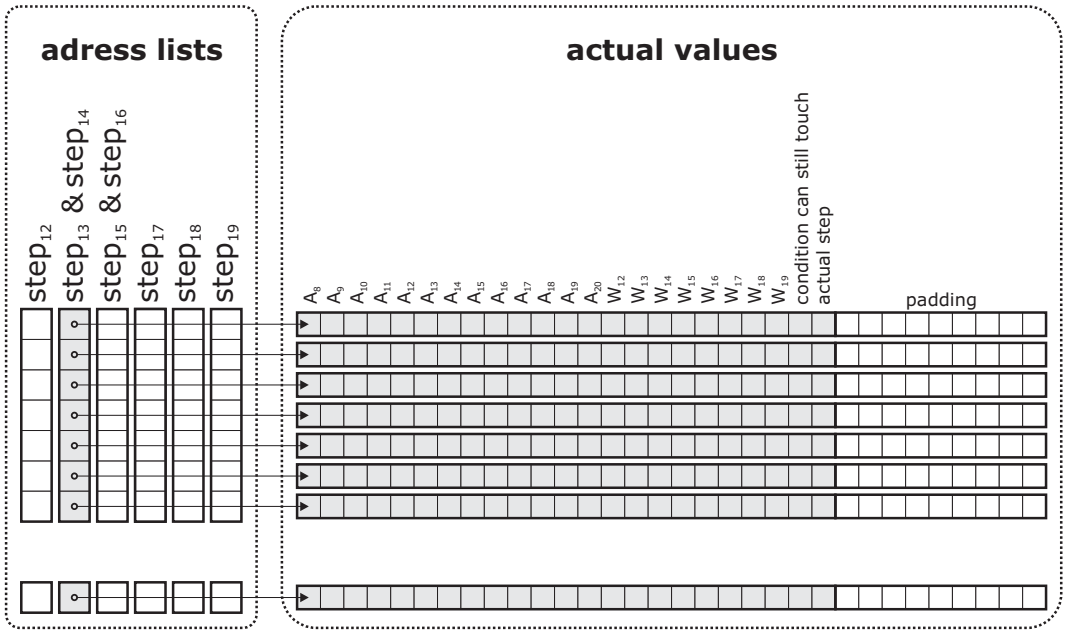

Figure 6.8: Arrangement of addresses and memory after the computation of the initialization kernel

After the initialization the address list with the starting step is the only one which contains addresses. The kernel scheduler than starts this kernel which computes the step or steps and stores the addresses into the corresponding address lists. After that a thread synchronization is called to ensure all threads are

finished before the kernel scheduler chooses the next kernel to execute.

The kernel scheduler always checks which kernel has the most threads to compute and invokes it. After a couple of kernel invocations a typical state is that every address list contains a number of addresses as shown in Figure 6.9. The kernel scheduler invokes kernels as long as there are addresses stored in the address lists.

There are two scenarios in which the computation is completed. One scenario is if there are no addresses left at all to compute. This would mean that no data set fulfills the required conditions. The second scenario is if all addresses are stored into the last step. All these data sets fulfill the conditions up to the last step.

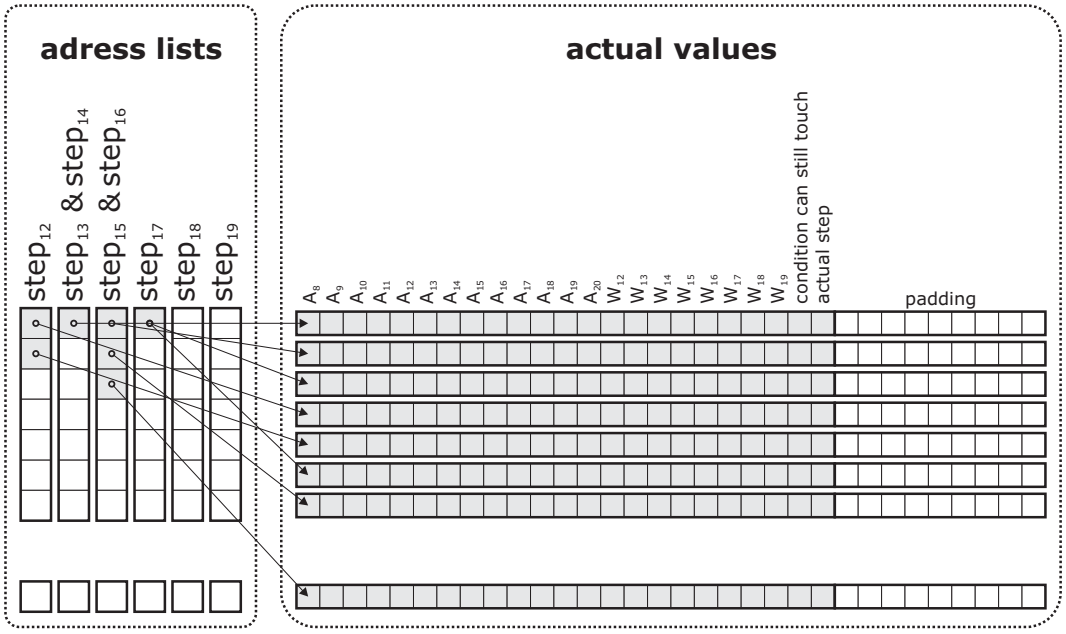

Figure 6.9: Arrangement of addresses and memory after the computation of a few kernels

6.8.2 Memory Usage

The only way to share data between different kernels is to use the global memory. This has the huge disadvantage that the implementation cannot benefit from faster memory spaces like for example the shared memory space. For global memory accesses to be performant certain requirements have to be fulfilled.

One important point is the arithmetic intensity which is described in Section 5.5.1. The arithmetic intensity of this implementation is very low. This basically means that the streaming multiprocessors often have to wait until certain values are available for further processing.

Another problem is that global memory accesses are many times slower if they are not coalesced. After a few runs of the kernel scheduler the addresses of the data sets are almost randomly distributed across the address lists of the different kernels. For this reason it is impossible to create coalesced memory accesses for this approach.

6.8.3 Runtime Analysis

The initial kernel takes a big part of the overall execution time. This is because of the fact that all known values have to be written into the data structure. The arithmetic intensity for this kernel is particularly low because there is almost no computation. As a point of reference, the parameterized approach needs about the same time for the entire computation as the initial kernel.

Although this approach has the longest execution time it should be mentioned that if memory structures will change in the future it could very well be the fastest approach. This is because of the fact that there is no thread divergence in the computational intensive parts and also only little computation overhead because of the kernel scheduler. Also the fact that the threads are very lightweight, and therefore fast to create, supports this theory.

6.8.4 Benchmarks and Limitations

In general there is only one limitation present for this approach. All the variables necessary have to be loaded from and stored to the global memory for the computation of each step. Also there is no coalescing of the memory accesses possible.

Benchmarks showed that this approach provides an average ratio of about 1 in comparison to the fastest CPU implementation. For the instance of the case study there is a 98% loss of efficiency. A more precise analysis is given in Section 6.9.

6.9 Comparison of the Different Approaches

There are 2 CPU implementations and 3 GPU implementations which are compared in the following. The second CPU implementation is a result of the fact that in order to get representative results, the influence of the computational overhead regarding the parameterized approach has to be known.

6.9.1 Memory Usage

The memory usage of the different kernels is an important key figure. Different devices have a different amount of memory at their disposal. For that reason the memory consumption of a thread or a kernel contributes also to the runtime of the algorithm. It is very likely that the ratio between achieved Gflops and theoretically possible Gflops changes with different devices. Table 6.6 shows the consumption of registers, shared memory, constant memory and local memory. The first value in the shared memory column is the memory which is allocated in the source code. The second value in the column is because of the parameters a kernel passes. The constant memory is divided into memory banks. The first value in the column shows memory bank 1. The second value in the column shows memory bank 2.

Noticeable is that the kernel from the ”precomputed values approach” uses significantly more registers than the other kernels. This kernel has not much latency to hide therefore the higher usage of registers has almost no effect on the runtime. There is also no significant usage of shared memory.

Interesting to see is that the ”parameterized approach” uses local memory. The usage of local memory is due to register spillage of the W array. Also interesting is that only 13 registers are used. This is because of the register spillage of W and the usage of shared memory for A.

The kernels of the ”kernel scheduled approach” all use a relatively low number of registers. All kernels which compute steps use less than 17 registers which results in a full occupancy rate. The only kernel which uses more than 16 registers is the initialization kernel.

Table 6.6: Memory usage of threads and kernels

Approach	Kernel name	Register usage	Shared memory	Constant Memory	Local memory
Precomputed values approach (GPU)	app1KernelCollision	28	96 16	568 108	
Parameterized approach (GPU)	app2KernelCollision	13	6752 16	3568 20	48
Kernel scheduled approach (GPU)	app3KernelCollisionStepInitialize	17	36 16	3568 4	
Kernel scheduled approach (GPU)	app3KernelCollisionStep12	14	24 16	3568 8	
Kernel scheduled approach (GPU)	app3KernelCollisionStep13_14	13	24 16	3568 12	
Kernel scheduled approach (GPU)	app3KernelCollisionStep15_16	15	4132 16	3568 24	
Kernel scheduled approach (GPU)	app3KernelCollisionStep17	11	6188 16	3568 48	
Kernel scheduled approach (GPU)	app3KernelCollisionStep18	13	6188 16	3568 48	
Kernel scheduled approach (GPU)	app3KernelCollisionStep19	11	6188 16	3568 48	

6.9.2 Benchmarks

Figure 6.10 shows a benchmark comparison of all five implementations. Dependent on the input data the ratio varies. The measurements show clearly that the occurring differences in runtime are marginal.

The ratio is calculated by taking the runtime of the fast CPU implementation dividing it through the runtime of the measured implementation. The ratio is a better performance indicator than the execution time. This is because of the fact that the GPU and the CPU used in the benchmarks are ”similar”. Similar in a sense, that they were purchased at the same time and also similar in sense of their ”hardware generations”. The fast CPU implementation was chosen as a reference value because it is fastest existing CPU implementation. The ratio now shows how many times faster or slower the other implementations compute the algorithm.

Table 6.7 shows the different execution time ratios between the implementations. The ratios were computed with the help of the mean values of a series of measurements which are shown in Figure 6.10. The order of computation is always columns through rows.

One essential ratio is the ratio between the ”precomputed values approach (CPU)” and the ”precomputed values approach (GPU)”. This ratio compares the fastest CPU implementation with the fastest GPU implementation. Table 6.7 shows that the GPU implementation is in average about 37 times faster

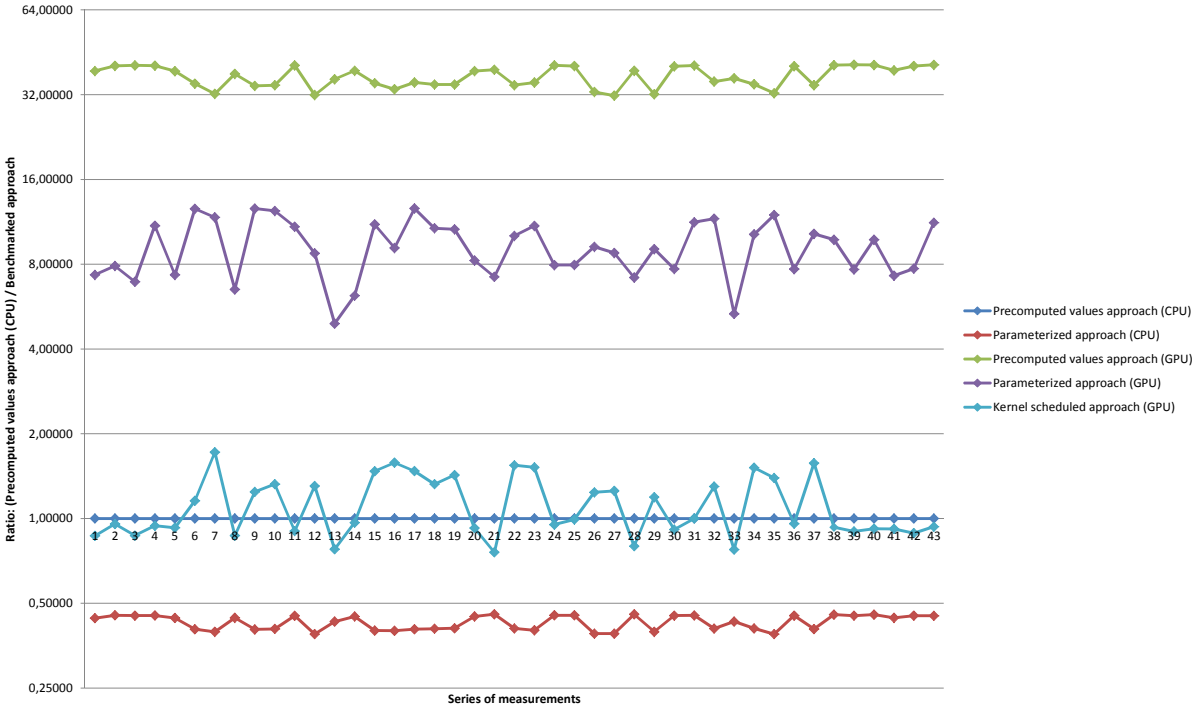

Figure 6.10: Benchmark comparison for a series of measurements

than the CPU implementation. With regards to the benchmark measured in Section 6.3.5 this means that the ”precomputed values approach (GPU)” works at about 62% of efficiency.

Also interesting is the ratio between the ”parameterized approach (CPU)” and the ”parameterized approach (GPU)”. The ”parameterized approach (GPU)” avoids divergence better than the ”precomputed values approach (GPU)”. Therefore it might be a good alternative for attacks which have a high number of diverging paths. Table 6.7 shows that the GPU version is about 22 times faster than the CPU version. Compared with the ”precomputed values approach (CPU)” it is about 9 times faster. A ratio of 9 is not that high compared to the ratio of the ”precomputed values approach (GPU)”. Measurements however showed that the register spillage in this case slows down the execution at about a factor of 2. With the possibility that in future versions of the compiler the register spillage does not occur anymore this implementation would be in range of the ”precomputed values approach (GPU)” in sense of efficiency for this implemented attack.

A look at the ratio between ”precomputed values approach (CPU)” and ”kernel scheduled approach (GPU)” shows that GPU implementation and CPU implementation have nearly the same execution times. This implementation avoids divergence completely for the computational intensive parts and should therefore be very effective. The problem is that the arithmetic intensity is that low that the occurring latency cannot be nearly hidden. Therefore this implementation is simply no viable substitute for existing CPU implementations. In future generations of devices the memory accesses might be a lot faster. In consequence of this, this implementation might compute more efficiently.

Table 6.7: Execution time ratios between different implementations

	Precomputed values app. (CPU)	Parameterized approach (CPU)	Precomputed values app. (GPU)	Parameterized approach (GPU)	Kernel scheduled app. (GPU)
Precomputed values approach (CPU)	1.00	0.43	37.16	9.24	1.12
Parameterized approach (CPU)	2.34	1.00	86.88	21.60	2.62
Precomputed values approach (GPU)	0.03	0.01	1.00	0.25	0.03
Parameterized approach (GPU)	0.11	0.05	4.02	1.00	0.12
Kernel scheduled approach (GPU)	0.89	0.38	33.15	8.24	1.00

6.10 Concluding Remarks

For an efficient implementation of a differential shortcut collision search, the best possible outcome would be to take the advantages of each part and combine these three approaches. In a broad sense the kernel scheduled approach is an extended version of the precomputed values approach. They are basically based on the same principles with the exception that the kernel scheduled approach avoids the divergence problem because of the memory pools. The parameterized approach takes a whole different path and suffers from computational overhead which is not present in the other two approaches.

In general it can be said that GPUs are very promising substitutes for CPUs in view of shortcut collision attacks. There are a few limitations such as memory latency and SIMT which hinder better results. It is also a fact that GPUs perform better when they have to compute floating point operations instead of integer operations. All in all the advantages outweigh the disadvantages. The fact that the fastest GPU implementation is about 37 times faster than the fastest CPU implementation underlines the previous statement.

Chapter 7

Outlook

Streaming processors are around for a while now. For a long time they were overshadowed by the traditional CPUs. With the rapidly growing gaming industry a new field of exploration emerged. The high demand on processing power for graphics processing was responsible that GPUs were developed to support and relieve the CPU. Over time the requirements on GPUs became more and more complex. In consequence of that also the structure of the GPUs became more complex. This development is responsible for the possibility to use these GPUs for general purpose processing. In the foreseeable future this process of development will continue.

For CPUs there is also an interesting development to detect. For a long time CPUs had only one processing core. One main effort of the manufacturer was to increase the clock rate of this core. With clock rates growing higher, cooling became more and more an issue. As a consequence of this CPUs with more than one core were produced. Looking at the development now, the number of cores in CPUs is increasing steadily, while the clock rates don't change that much over time.

Looking at the fact that GPUs grow more complex in their structure and CPUs are getting more cores or kernels, it is noticeable that these two fields of operation grow more and more together. It seems likely that in the future these two fields of operation will merge together.

7.1 Ideas for Future Work

CUDA with its associated GPUs is a relatively young field of exploration. Therefore the hardware at this time has different limitations which influence the efficiency of a program. The implemented approaches in the case study can be seen as design patterns for other implementations of collision attacks. There is also the fact that GPUs get a more complex structure with every generation. Limitations such as memory latency and SIMD won't possibly be there for later generations of devices. In consequence the efficiency of the implementations will rise in the future.

Chapter 8

Concluding Remarks

The findings presented in this thesis clearly state that streaming processors are very suitable to solve shortcut collision attacks on SHA-1. Furthermore due to their similar structure shortcut collision attacks on other hash algorithms are also very suitable to compute with a high degree of certainty. Results of the case study show that two out of three implemented approaches achieve better results in respect of runtime compared to ”traditional” CPU implementations. What has to be highlighted, is that the fastest GPU implementation is about 37 times faster than the fastest CPU implementation

Due to their non-deterministic runtime behaviors it stands to reason that shortcut collision attacks cannot be solved on today's GPUs without losses in efficiency. Although a 40% loss in efficiency seems to be a lot at first glance it is negligible in contrast to the overall performance gain.

Results in the case study show also that each implemented approach has its benefits and its limitations regarding to the features the used hardware provides. Overall it can be said that the precomputed values approach will work very efficiently for attacks which display a ”homogeneous path behavior” for different data sets. The parameterized approach will work very efficiently when the path behavior is not so homogeneous for different data sets.

In view of the high complexity, in sense of runtime, most shortcut collision attacks display, using GPUs for the computation is a good way to reduce computation time. It also gives cryptographers a viable alternative to distributed computing efforts.

In conclusion can be said that the gain in performance outweighs the higher development effort by far which is certainly necessary to implement a collision attack on GPUs.

Appendix A

Benchmarks GPU CPU

Tables A.1, A.2 and A.3 list the exact data which was used to compile Figure 4.6 and Figure 4.7 in Section 4.5.4

Table A.1: The exact benchmark data of NVIDIA GPUs

NVIDIA				
Model	Codename	Release Date	Bandwidth (GB/s)	Gflops
Ti 500	NV20	Oct-01	8 [har10b]	10 [har10b]
Ti 4600	NV25	Feb-02	10.4 [har10b]	16 [har10b]
FX5950 ULTRA	NV38	Oct-03	30.4 [har10b]	29 [har10b]
6800 ULTRA	N45	Mar-05	33.6 [har10b]	75 [har10b]
7800 GTX	G70	Jun-05	38.4 [har10b]	200 [har10b]
7900 GTX	G71	Mar-06	51.2 [har10b]	301 [har10b]
8800GTX	G80	Nov-06	86.4 [NVI08]	518 [NVI08]
9800 GTX	G92	Apr-08	70.4 [NVI10a]	648 [NVI10a]
GTX 280	GT200	Jun-08	141.7 [NVI08]	933 [NVI08]
Fermi	GF100	Mar-10	192 [Nat09a]	1500 [Nat09a]

Table A.2: The exact benchmark data of ATI GPUs

ATI				
Model	Codename	Release Date	Bandwidth (GB/s)	Gflops
8500	R200	Aug-01	8.8 [har10a]	12 [har10a]
9800 XT	R360	Dec-03	23.4 [har10a]	43 [har10a]
X 850 XT PE	R480	Dec-04	37.8 [har10a]	102 [har10a]
X 1950 XTX	R580+	Jul-06	64 [har10a]	374 [har10a]
HD 2900 XT	R600	May-07	105.4 [har10a]	474,88 [har10a]
HD 3870	RV670XT	Nov-07	72 [har10a]	496 [har10a]
HD 4870	RV7770XT	Jun-08	115.2 [Nea08]	1200 [AMD10a]
HD 5870	CypressXT	Sep-09	153.6 [AMD10b]	2720 [AMD10b]

Table A.3: The exact benchmark data of Intel CPUs

Intel				
Model	Codename	Release Date	Bandwidth (GB/s)	Gflops
Pentium 4 EE	Prescott	Feb-04	3.2 [Int01]	7,5 [Tom04]
X6800 E	Conroe	Jul-06	10.7 [Int07a]	18,8 [Tom07]
QX6800	Kentsfield	Jun-07	8.5 [Int07b]	38,7 [Tom07]
QX9770 E	Yorkfield	Mar-08	12.4 [Int09a]	47,4 [Tom07]
i7-975 E	Bloomfield	May-09	25.6 [Int09b]	72 [Tom09]

A.1 Synthetic Benchmarks GPU CPU

The following listings show the complete source code of the synthetic benchmarks described in Section 6.3.4

```
1  __global__ void kflop()
2  {
3
4    float a = result[threadIdx.x];
5    float b = 1.01f;
6
7    #pragma unroll
8    for (int i = 0; i < 1024; i++)
9    {
10     a = b+a+b;
11   }
12
13   result[threadIdx.x] = a+b;
14 }
```

```
1  __global__ void kflopMAD()
2  {
3
4    float a = result[threadIdx.x];
5    float b = 1.01f;
6
7    #pragma unroll
8    for(int i = 0; i < 1024; i++)
9    {
10     a= b*a+b;
11     a= b*a+b;
12   }
13
14   result[threadIdx.x] = a+b;
15 }
```

```
1  __global__ void kflopMULMAD()
2  {
3
4    float a = result[threadIdx.x];
5    float b = 1.01f;
6
```

```
7     #pragma unroll
8     for (int i = 0; i < 1024; i++)
9     {
10      a = b * a + b;
11      a = a * b;
12    }
13
14    result[threadIdx.x] = a+b;
15  }
```

```
1   __global__ void kflopMULMADMAD()
2   {
3
4     float a = result[threadIdx.x];
5     float b = 1.01f;
6
7     #pragma unroll
8     for (int i = 0; i < 1024; i++)
9     {
10      a = b * a + b;
11      a = a * b;
12      a = a * b;
13    }
14
15    result[threadIdx.x] = a+b;
16  }
```

```
1   __global__ void kiop()
2   {
3
4     unsigned int a = result[threadIdx.x];
5     unsigned int b = threadIdx.x;
6
7     #pragma unroll
8     for (int i = 0; i < 1024; i++)
9     {
10      a =  b & a + b;
11    }
12
13    result[threadIdx.x] = a+b;
14  }
```

```
1  __global__ void kiopMULMAD()
2  {
3
4    unsigned int a = result[threadIdx.x];
5    unsigned int b = threadIdx.x;
6
7      #pragma unroll
8    for (int i = 0; i < 1024; i++)
9    {
10     a = b * a + b;
11     a = a * b;
12     a = a * b;
13   }
14
15   result[threadIdx.x] = a+b;
16 }
```

Bibliography

[ACK+02] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. Seti@home: an experiment in public-resource computing. *Commun. ACM*, 45(11):56–61, November 2002. (Cited on page 25.)

[AHMP08] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and Raphael C.-W. Phan. Sha-3 proposal blake. Submission to NIST, 2008. (Cited on pages 8 and 9.)

[AMD06] AMD. Ati ctm guide - technical reference manual, 2006. (Cited on page 1.)

[AMD10a] AMD. Ati radeon hd 4870 graphics, 2010. (Cited on page 77.)

[AMD10b] AMD. Ati radeon hd 5870 gpu feature summary, 2010. (Cited on page 77.)

[BBG+08] Ryad Benadjila, Olivier Billet, Henri Gilbert, Gilles Macario-Rat, Thomas Peyrin, Matt Robshaw, and Yannick Seurin. Sha-3 proposal: Echo. Submission to NIST, 2008. (Cited on page 8.)

[BC04] Eli Biham and Rafi Chen. Near-collisions of SHA-0. In *CRYPTO: Proceedings of Crypto*, 2004. (Cited on page 15.)

[BCCM+08] Emmanuel Bresson, Anne Canteaut, Benoît Chevallier-Mames, Christophe Clavier, Thomas Fuhr, Aline Gouget, Thomas Icart, Jean-François Misarsky, Marìa Naya-Plasencia, Pascal Paillier, Thomas Pornin, Jean-René Reinhard, Céline Thuillet, and Marion Videau. Shabal, a submission to nistś cryptographic hash algorithm competition. Submission to NIST, 2008. (Cited on page 9.)

[BCJ+05] Biham, Chen, Joux, Carribault, Lemuet, and jalby. Collisions of SHA-0 and reduced SHA-1. In *EUROCRYPT: Advances in Cryptology: Proceedings of EUROCRYPT*, 2005. (Cited on pages 12 and 19.)

[BD08] Eli Biham and Orr Dunkelman. The shavite-3 hash function. Submission to NIST, 2008. (Cited on page 9.)

[BDP97] Antoon Bosselaers, Hans Dogbbertin, and Bart Preneel. The RIPEMD-160 cryptographic hash function. *Dr. Dobb's Journal of Software Tools*, 22(1):24, 26, 28, 78, 80, January 1997. (Cited on page 6.)

[BDPA08] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Keccak specifications. Submission to NIST, 2008. (Cited on page 9.)

[Ber08] Daniel J. Bernstein. Cubehash specification (2.b.1). Submission to NIST, 2008. (Cited on page 8.)

[CJ98] Florent Chabaud and Antoine Joux. Differential collisions in SHA-0. In *CRYPTO: Proceedings of Crypto*, 1998. (Cited on pages 13, 14, 15, 16 and 18.)

[Cor09] Microsoft Corporation. Microsoft directx, 2009. (Cited on page 1.)

[CR06] Christophe De Cannière and Christian Rechberger. Finding SHA-1 characteristics: General results and applications. In Xuejia Lai and Kefei Chen, editors, *Advances in Cryptology - ASIACRYPT 2006, 12th International Conference on the Theory and Application of Cryptology and Information Security, Shanghai, China, December 3-7, 2006, Proceedings*, volume 4284 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006. (Cited on pages 21 and 24.)

[CRDI07] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. Cell broadband engine architecture and its first implementation: a performance view. *IBM J. Res. Dev.*, 51(5):559–572, 2007. (Cited on page 28.)

[CSW08] Christophe De Canniere, Hisayoshi Sato, and Dai Watanabe. Hash function luffa: Specification. Submission to NIST, 2008. (Cited on page 9.)

[CW76] H. J. Curnow and B. A. Wichmann. A synthetic benchmark. *The Computer Journal*, 19(1):43–49, February 1976. (Cited on page 54.)

[Dam90] Ivan B. Damgaard. A design principle for hash functions. In *Advances in Cryptology (CRYPTO '89)*, pages 416–427, Berlin - Heidelberg - New York, August 1990. Springer. (Cited on page 4.)

[Dob96] H. Dobbertin. Cryptanalysis of MD4. *Lecture Notes in Computer Science*, 1039:53–69, 1996. (Cited on page 12.)

[FFG09] Ewan Fleischmann, Christian Forler, and Michael Gorski. Classification of the SHA-3 candidates. In Helena Handschuh, Stefan Lucks, Bart Preneel, and Phillip Rogaway, editors, *Symmetric Cryptography*, number 09031 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2009. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany. (Cited on page 9.)

[FIP01] FIPS. *Advanced Encryption Standard (AES)*. National Institute for Standards and Technology, pub-NIST:adr, November 2001. (Cited on page 8.)

[FLS+08] Niels Ferguson, Stefan Lucks, Bruce Schneier, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas, and Jesse Walker. The skein hash function family. Submission to NIST, 2008. (Cited on page 9.)

[Fol] Folding. Folding@home distributed computing. http://folding.stanford.edu/. (Cited on page 25.)

[FSYA07] Wilson W. L. Fung, Ivan Sham, George L. Yuan, and Tor M. Aamodt. Dynamic warp formation and scheduling for efficient GPU control flow. In *MICRO*, pages 407–420. IEEE Computer Society, 2007. (Cited on page 46.)

[GKK+08] Danilo Gligoroski, Vlastimil Klima, Svein Johan Knapskog, Mohamed El-Hadedy, Jorn Amundsen, and Stig Frode Mjolsnes. Cryptographic hash function blue midnight wish. Submission to NIST, 2008. (Cited on page 8.)

[GKM+08] Praveen Gauravaram, Lars R. Knudsen, Krystian Matusiewicz, Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S. Thomsen. Grøstl – a sha-3 candidate. Submission to NIST, 2008. (Cited on pages 8 and 9.)

[har10a] hardware-infos.com. Ati-radeon-serie, 2010. (Cited on page 77.)

[har10b] hardware-infos.com. Nvidia-geforce-serie, 2010. (Cited on page 77.)

[HHJ08] Shai Halevi, William E. Hall, and Charanjit S. Jutla. The hash function fugue. Submission to NIST, 2008. (Cited on page 8.)

[Int01] Intel. Desktop performance and optimization for intel pentium 4 processor, February 2001. (Cited on page 78.)

[Int02] Intel Corporation. Creating a pci express interconnect, 2002. (Cited on page 36.)

[Int04] Intel Corporation. Improve video quality with the pci express x16 graphics interface, 2004. (Cited on page 36.)

[Int07a] Intel. Intel core 2 extreme processor x6800 and intel core 2 duo desktop processor e6000 and e4000 sequences, October 2007. (Cited on page 78.)

[Int07b] Intel. Intel core 2 extreme quad-core processor qx6800, April 2007. (Cited on page 78.)

[Int09a] Intel. Intel core 2 extreme processor qx9000 series, intel core 2 quad processor q9000, q9000s, q8000 and q8000s series, August 2009. (Cited on page 78.)

[Int09b] Intel. Intel core i7-975 processor extreme edition, 2009. (Cited on page 78.)

[Int10a] Intel. Intel core i7-900 desktop processor extreme edition series and intel core i7-900 desktop processor series, February 2010. (Cited on page 52.)

[Int10b] Intel. Intel microprocessor export compliance metrics, July 2010. (Cited on page 54.)

[JP07] Antoine Joux and Thomas Peyrin. Hash functions and the (amplified) boomerang attack. In Alfred Menezes, editor, *Advances in Cryptology - CRYPTO 2007, 27th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2007, Proceedings*, volume 4622 of *Lecture Notes in Computer Science*, pages 244–263. Springer, 2007. (Cited on page 20.)

[KDH+05] James A. Kahle, Michael N. Day, H. Peter Hofstee, Charles R. Johns, Theodore R. Maeurer, and David J. Shippy. Introduction to the cell multiprocessor. *IBM Journal of Research and Development*, 49(4-5):589–604, 2005. (Cited on page 28.)

[KDK+01] Brucek Khailany, William J. Dally, Ujval J. Kapasi, Peter R. Mattson, Jinyung Namkoong, John D. Owens, Brian Towles, Andrew Chang, and Scott Rixner. Imagine: Media processing with streams. *IEEE Micro*, 21(2):35–46, 2001. (Cited on pages 25 and 28.)

[KDR+03] Brucek Khailany, William J. Dally, Scott Rixner, Ujval J. Kapasi, John D. Owens, and Brian Towles. Exploring the vlsi scalability of stream processors. In *HPCA '03: Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, page 153, Washington, DC, USA, 2003. IEEE Computer Society. (Cited on page 25.)

[KKS00] John Kelsey, Tadayoshi Kohno, and Bruce Schneier. Amplified boomerang attacks against reduced-round MARS and Serpent (abstract only). In NIST, editor, *The Third Advanced Encryption Standard Candidate Conference, April 13–14, 2000, New York, NY, USA*, pages 10–10, pub-NIST:adr, 2000. National Institute for Standards and Technology. (Cited on page 20.)

[KRD+03] Ujval J. Kapasi, Scott Rixner, William J. Dally, Brucek Khailany, Jung Ho Ahn, Peter Mattson, and John D. Owens. Programmable stream processors. *Computer*, 36(8):54–62, 2003. (Cited on pages 25 and 26.)

[LBF08] Gaëtan Leurent, Charles Bouillaguet, and Pierre-Alain Fouque. Simd is a message digest. Submission to NIST, 2008. (Cited on page 9.)

[Mer79] Ralph Charles Merkle. Secrecy, authentication, and public key systems. Technical report, Stanford University, June 1979. (Cited on page 4.)

[Mer90] Ralph C. Merkle. A certified digital signature. In *Advances in Cryptology (CRYPTO '89)*, pages 218–238, Berlin - Heidelberg - New York, August 1990. Springer. (Cited on page 4.)

[MR] Florian Mendel and Christian Rechberger. Clustered truncated differentials and fast collision search. unpublished manuscript. (Cited on page 22.)

[MRR09] Florian Mendel, Christian Rechberger, , and Vincent Rijmen. Sha-1 collision search graz, 2009. (Cited on page 22.)

[MvOV96] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanston, editors. *Handbook of Applied Cryptography*. CRC Press, 3rd edition, June 1996. (Cited on pages 1, 3, 5 and 11.)

[Nat93] National Institute of Standards and Technology (NIST). *FIPS Publication 180: Secure Hash Standard (SHS)*, May 11, 1993. (Cited on page 6.)

[Nat09a] Nathan Brookwood. Nvidia solves the gpu computing puzzle, 2009. (Cited on page 77.)

[Nat09b] National Institute of Standards and Technology. Status report on the first round of the sha-3 cryptographic hash algorithm competition. Technical report, National Institute for Standards and Technology, pub-NIST:adr, September 2009. (Cited on page 8.)

[NBB+] James Nechvatal, Elaine Barker, Lawrence Bassham, William Burr, and Morris Dworkin. Report on the development of the advanced encryption standard (AES). (Cited on page 8.)

[NCBF+02] Naffziger, S.D. Colon-Bonet, G. Fischer, T. Riedlinger, R. Sullivan, T.J. Grutkowski, T. Hewlett-Packard, and Fort Collins. The implementation of the itanium 2 microprocessor. In *Solid-State Circuits*, pages 1448 – 1460. IEEE Solid-State Circuits Society, 2002. (Cited on page 25.)

[Nea08] Neal Robison. Understanding graphics, 2008. (Cited on page 77.)

[NVI06] NVIDIA. Technical brief: Nvidia geforce 8800 gpu architecture overview, November 2006. (Cited on pages 29 and 31.)

[NVI08] NVIDIA. Nvidia geforce gtx 200 gpu architectural overview, 2008. (Cited on pages 54 and 77.)

[NVI09a] NVIDIA. Nvidia cuda architecture, 2009. (Cited on pages 1, 35 and 36.)

[NVI09b] NVIDIA. Nvidias next generation cuda compute architecture: Fermi, 2009. (Cited on page 37.)

[NVI09c] Corporation NVIDIA. *Programming Guide 2.2*, April 2009. (Cited on pages 27, 35, 37 and 43.)

[NVI10a] NVIDIA. Geforce 9800 gtx specifications, 2010. (Cited on page 77.)

[NVI10b] NVIDIA. Nvidia cuda best practices guide 3.0, 2010. (Cited on pages 37 and 47.)

[NVI10c] Corporation NVIDIA. *Programming Guide 3.0*, Feb 2010. (Cited on pages 39, 40, 41, 42, 43, 45 and 46.)

[oST02] National Institute of Standards and Technology. Secure hash standard. Federal Information Processing Standards Publication (FIPS PUB) 180-2, August 2002. February 2004 specifies SHA-224 and discusses truncation of the hash function output in order to provide interoperability. (Cited on pages 1, 5, 6, 7, 8 and 9.)

[oST10] National Institute of Standards and Technology. Email announcing finalists, 2010. (Cited on page 9.)

[PCI07] PCI-SIG. Pci-sig delivers pci express 2.0 specification, 2007. (Cited on page 36.)

[Pet09] Peter N. Glaskowsky. Nvidias fermi: The first complete gpu computing architecture, September 2009. (Cited on page 25.)

[PF05] Matt Pharr and Randima Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional, 2005. (Cited on page 27.)

[Pre94] B. Preneel. Design principles for dedicated hash functions. *Lecture Notes in Computer Science*, 809:71–82, 1994. (Cited on page 3.)

[RD99] Edward Roback and Morris Dworkin. Conference report: First Advanced Encryption Standard (AES) Candidate Conference, Ventura, CA, August 20–22, 1998. *Journal of research of the National Institute of Standards and Technology*, 104(1):97–105, January/February 1999. (Cited on page 8.)

[Rec09] Christian Rechberger. *Cryptanalysis of Hash Functions*. PhD thesis, Graz University of Technology, 2009. (Cited on page 22.)

[Riv92a] R. L. Rivest. *RFC 1320: The MD4 Message-Digest Algorithm*. Internet Activities Board, April 1992. (Cited on page 6.)

[Riv92b] R. L. Rivest. *RFC 1321: The MD5 Message-Digest Algorithm*. Internet Activities Board, April 1992. (Cited on pages 1 and 6.)

[Rix02] Scott Rixner. *Stream processor architecture*. Kluwer Academic Publishers, Norwell, MA, USA, 2002. (Cited on pages 26 and 27.)

[RO05] Rijmen and Oswald. Update on SHA-1. In *CTRSA: CT-RSA, The Cryptographers' Track at RSA Conference, LNCS*, 2005. (Cited on pages 16 and 17.)

[SGI92] SGI. OpenGL - reference manual, the official reference documantation for openGL, release 1. *Addison Wesley, ISBN 0-201-63276-4*, 1992. (Cited on page 1.)

[Tom04] Tom's Hardware. Review intel, January 2004. (Cited on page 78.)

[Tom07] Tom's Hardware. Review intel core 2 extreme qx9770, November 2007. (Cited on page 78.)

[Tom09] Tom's Hardware. Review core i7 975, February 2009. (Cited on page 78.)

[Wag99] Wagner. The boomerang attack. In *IWFSE: International Workshop on Fast Software Encryption, LNCS*, 1999. (Cited on page 20.)

[Wan97] Xiaoyun Wang. The collision attack on sha-0. www.infosec.edu.cn, 1997. (Cited on page 18.)

[Wei84] Reinhold P. Weicker. Dhrystone: a synthetic systems programming benchmark. *Commun. ACM*, 27(10):1013–1030, 1984. (Cited on page 54.)

[WFLY04] Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu. Collisions for hash functions MD4, MD5, HAVAL-128 and RIPEMD. Cryptology ePrint Archive, Report 2004/199, 2004. (Cited on page 12.)

[Wu08] Hongjun Wu. The hash function jh. Submission to NIST, 2008. (Cited on page 9.)

[WYY05a] Xiaoyun Wang, Andrew Yao, and Frances Yao. New collision search for sha-1. Rump Session CRYPTO 2005, 2005. (Cited on pages 19, 20 and 21.)

[WYY05b] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding collisions in the full SHA-1. *Lecture Notes in Computer Science*, 3621:17–??, 2005. (Cited on pages 12 and 17.)

[WYY05c] Xiaoyun Wang, Hongbo Yu, and Yiqun Lisa Yin. Efficient collision search attacks on SHA-0. *Lecture Notes in Computer Science*, 3621:1–??, 2005. (Cited on page 17.)

[YG07] Jason Yang and James Goodman. Symmetric key cryptography on modern graphics hardware. In *ASIACRYPT*, pages 249–264, 2007. (Cited on pages 30 and 31.)

[Yuv79] G. Yuval. How to swindle Rabin. *Cryptologia*, 3:187–189, July 1979. (Cited on page 11.)

[zK08] Özgül Kücük. The hash function hamsi. Submission to NIST, 2008. (Cited on page 9.)

[ZPS93] Y. Zheng, J. Pieprzyk, and J. Seberry. HAVAL — a one-way hashing algorithm with variable length of output. *Lecture Notes in Computer Science*, 718:83–104, 1993. (Cited on page 6.)