RUHR-UNIVERSITÄT BOCHUM

GPU assisted Mutual Information Analysis Attacks on AES

Hans Christoph Hudde

Bachelor Thesis
Chair for Embedded Security - Prof. Dr.-Ing. Christof Paar

Abstract

In 2008 the Mutual Information Analysis (MIA) has been introduced as a generic side-channel key distinguisher suitable for detecting non-linear relations between measurements and hypothetical leakage. It is considered a convenient technique for higher-order scenarios as well as for attacks by an adversary not capable of obtaining an accurate leakage model. This thesis uses CUDA, Nvidia's framework for General Purpose Computations on Graphics Processing Units (GPU), for a fast parallel implementation of MIA that achieved a performance boost of a factor of 4 to 12 compared to a sequential reference implementation. We evaluate different optimization strategies and analyze the constraints to the possible profit that MIA can draw from stream processing parallelizations. More generally our results suggest that GPUs can significantly speed up the computation time of side channel attacks. While the performance gain for our approach is decreased by hitting the memory bandwidth limits without exhausting the computational power of the GPU, attacks having a higher arithmetic intensity, e.g. in higher-order scenarios, are likely to achieve an even higher performance gain.

Declaration

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of the university or other institute of higher learning, except where due acknowledgement has been made in the text.

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, dass alle Stellen der Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, als solche kenntlich gemacht sind und dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegt wurde.

Hans Christoph Hudde

Bochum, April 26, 2010

Contents

1 Introduction

With the Mutual Information Analysis (MIA) Attacks published by Gierlichs et al. [GBTP08] at CHES 2008 a new type of side channel attack with some interesting properties hast been introduced. This thesis combines recent developments in the field of side channel analysis attacks with the performance potential provided by General-Purpose computing on Graphics Processing Units (GPGPU) by proposing a GPU-assisted MIA implementation using Nvidia's CUDA framework.

1.1 Motivation

During the last decade published side-channel attacks convinced cryptographers that the physical security of cryptographic implementations has to be considered as an equally important factor as the mathematical security of the implemented algorithm itself. With the evolution of side-channel attacks, several types of key distinguishers have been proposed. All of them have in common that they investigate the relationship between the measurement of a device's leakage and a simulated leakage behavior that assumes a specific subkey. The distinguisher is then used to decide which model of the leakage behavior, i.e. which subkey, fits best to the observed measurements.

These passive side channel attacks can be performed with cheap equipment, and hence represent a serious threat especially to devices in hostile environments such as in smart cards or RFID applications, where an adversary has unconditional physical access to the device. Furthermore these attacks are usually non-invasive, leaving no trace of the attack on the device.

What distinguishes the MIA from traditional side channel attacks such as Differential Power Analysis (DPA) is the fact that it does not assume a linear dependency between prediction and leakage, nor does it need a profiling step like template attacks. MIA uses a generic approach that is easily adaptable to the multidimensional case, thus representing a promising candidate in complex scenarios, e.g. against devices designed to resist power analysis.

Since performing MIA on huge amounts of data is time-consuming, making the computational power of modern GPUs available for the computation promises to be a rewarding task, provided that it can be parallelized efficiently. GPUs are stream processors, that means they operate by running a single function termed *kernel* on many independent

data records in parallel. This is accomplished by having many cores in each GPU, each of which can run a high number of threads simultaneously, thus offering large performance benefits to applications suited for this kind of architecture.

1.2 Overview

This thesis strives for an efficient implementation of MIA using Nvidia's CUDA framework, thus exploring the opportunities of GPUs to speed up cryptographic computations. A workstation with several recent Nvidia GPUs was available for this thesis in order to test the efficiency of the implementation using multiple devices.

As a first step a reference implementation running on a conventional CPU was created, constituting the basis for later comparisons with the GPU implementation by means of the CUDA interface. In a second step possible parallelization designs have been theoretically and practically evaluated with regard to the potential performance gain on the one hand and the loss of accuracy on the other. After choosing the most promising design, implementation and later optimizations followed as next steps. Finally the results have been evaluated by comparing the achieved performance with the reference implementation as well as by analyzing the performance by combining reference and CUDA implementation and varying available parameters.

Since it was not foreseeable at the beginning of the thesis whether an effective parallelization strategy for a stream processing implementation of Mutual Information Analysis would be found, a performance degradation was considered a possible outcome.

1.3 Organization of this Thesis

This thesis is organized as follows:

In chapter 2 we give an introduction to side channel attacks in general and describe the information-theoretic approach leading to the MIA distinguisher in particular. Moreover the basics of programming Graphics Processing Units (GPU) using Nvidia's CUDA framework is discussed.

The next chapter highlights the general implementation steps for implementing MIA and discusses the applied design decisions. This chapter introduces the terms that are used for both the CPU and GPU implementation.

A reference implementation is presented in chapter 4 to allow a later evaluation of the GPU-assisted implementation. Additionally the design of the implementation allows to combine CPU and GPU power to speed up the total runtime.

Chapter 5 describes the implementation in C for CUDA, particularly emphasizing optimization strategies for the high-throughput kernel design.

The combined results of both implementations are analyzed in chapter 6, before the thesis is concluded in chapter 7 with an outlook on future investigations that could follow this work.

2 Theoretical background

The advent of side channel attacks on cryptographic hardware devices began with the discovery of subtle key-dependent timing variations. Motivated by attacks employing these timing variations to gain information about the key, researchers started examining the power consumption of cryptographic devices. With the publication of an article on *Differential Power Analysis* by Kocher *et al.* [KJJ99] in 1998, the potential of power analysis attacks for revealing key material began to be fully recognized and a widespread development of countermeasures and enhanced analysis methods started.

This chapter gives an overview on methods and countermeasures and aims to provide a motivation for utilizing generic analysis frameworks like the Mutual Information Analysis (MIA).

2.1 AES

This section gives a brief review of the design of the AES (Advanced Encryption Standard, described in [AES01]), today's most commonly used symmetric block cipher, which is targeted by the attack in this thesis.

2.1.1 Structure

AES is built on basis of a substitution-permutation network that operates on blocks of 128 bits called *state*, employing a key size of 128, 192 or 256 bits and a key size dependent number of 10, 12 or 14 rounds. Each round of the encryption consists of a non-linear substitution called `SubBytes`, a cyclic shift operation called `ShiftRows`, a linear mixing transformation called `MixColumns` that is skipped in the final round for perfomance reasons, and finally `AddRoundKey` which is a bitwise XOR of the round key generated by the key schedule and the state. `AddRoundKey` is also performed prior to the first encryption round. All operations can be inverted and performed in reverse order to allow decryption.

The substitution table used in the `SubBytes` operation is called S-Box and was designed to be resistant to linear and differential cryptanalysis by ensuring a complex relationship between key and ciphertext. This property is called confusion and was

identified as one of the most important properties for all secrecy systems by Shannon [Sha49] in 1949.

Although for performance reasons being usually implemented as a lookup-table, the substitution values of the S-Box are not random but derived from the multiplicative inverse over a finite field and also allow a short mathematical description of the S-Box. As detailed in the section on side channel power models (2.2.2) as well as in the implementation chapter, we are only interested in the Hamming weights of the substitution values for the purposes of our side channel attack. Consequently a simplified lookup-table can be used which replaces the S-Box outputs with their Hamming weights (see Algorithm 3 for details).

The linear components of AES are mainly designed to provide diffusion, i.e. they spread the influence of every input bit to many output bits with the goal to hide the distribution of individual plaintext symbols and to complicate the statistical analysis of the cipher. This diffusion property is the second of the two most important properties described by Shannon.

2.1.2 Qualification as cryptographic standard and as attack target

AES was chosen out of fifteen submissions as the new symmetric block cipher standard by the National Institute of Standards and Technology (NIST) in 2001 during an open evaluation process with the intention to replace its predecessor DES, which suffered from a small key size. Thanks to the public election process and the crypto communities high confidence in the security of the AES due to the publicly available and detailed analysis and its good performance in hardware and software it has been quickly adopted for governmental, commercial and private use. It is still considered secure, because there is no computationally feasible publicly known cryptanalysis of the AES with the full round number.

Side channel attacks on specific implementations are possible though, and allow extracting the full key by performing a DPA-like attack against each round to recover the round keys, always deriving the input to the S-Box from the output of the previous round. Figure 2.1 gives an high-level overview on the AES and points out that the intermediate value after the S-Box substitution is a convenient value to use for the attack, as it depends on both the secret key and the input. In this case the diffusion caused by the S-Box is advantageous for the attacker, because it helps making the influence of the key detectable.

Thus, AES is a good example for a cryptographically strong algorithm that is nevertheless very weak in many implementations despite of their correctness. Other block ciphers with similar structures like DES can be attacked analogue.

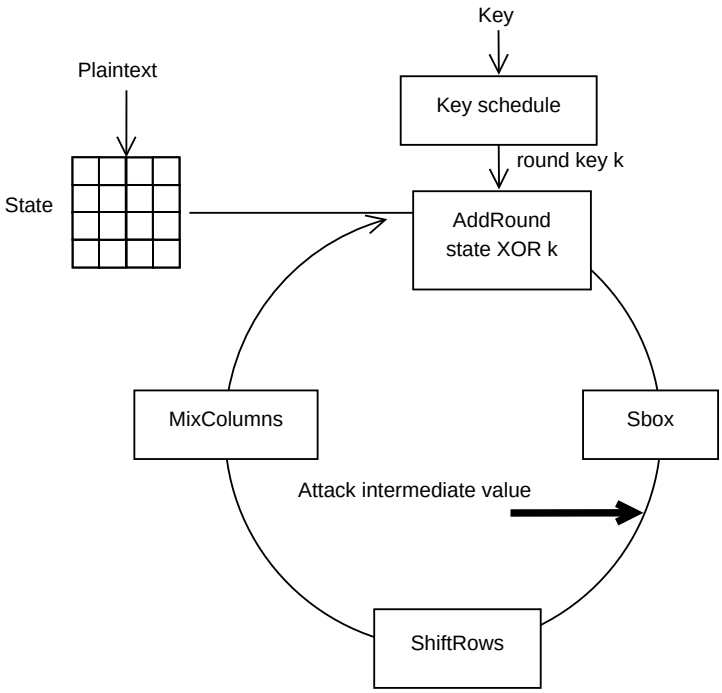

Figure 2.1: High-level view on AES

2.2 Power analysis attacks on cryptographic devices

Power analysis is a type of side channel attack, i.e. it uses information that unintentionally leaks from the physical implementation of a cryptographic device. It is non-invasive and can be conducted with relatively cheap equipment, thus representing a serious security threat for widely used devices such as smart cards.

Power analysis is possible because the power consumption of a device depends on all processed values, including secret keys.[1] The dependency may be very strong and easy to exploit, but often it is necessary to obtain many power traces or to analyse the electrical characteristics of a device in detail before the dependency can be exploited. Countermeasures try to remove the dependency to further complicate an exploit.

2.2.1 Differential Power Analysis

While it may be possible to attack a device just by examining its power consumption over time, which is known as Simple Power Analysis (SPA), more advanced types of

[1] Apart from power analysis, other leakage sources such as electromagnetic emissions can be exploited for side channel attacks.

analyses are required to overcome protection methods and to compute intermediate values of cryptographic computations in black box devices. By statistically analyzing data collected from power traces recorded during multiple cryptographic operations, an adversary can even employ very noisy measurements. Differential Power Analysis (DPA) uses this approach and is said to be the "most popular type of power analysis attack" [KJJ99].

The main idea behind DPA is to relate measured power traces to key-dependent leakage models, evaluating which model leads to the best prediction of the actual leakage. From an abstracted point of view this is equivalent to computing the degree of dependence of two random variables describing model and leakage function. There exist several mathematical methods that accomplish this task in a more or less effective way and can be used with MIA by replacing the originally proposed methods.

To allow a better understanding of the DPA basics, we present a simple DPA attack on an AES encryption device. Therefore we need an intermediate value v that depends on the secret key that we want to extract. The substitution box (S-Box) of AES operates on byte-level and depends on both one round key byte (k) and one plaintext byte (p), thus allowing attackers to recover the key byte by byte. We choose v to be the Most Significant Bit (MSB) of the S-Box output: $v = MSB(S(p \oplus k))$. The next step is to execute a large number (e.g. 1000) of encryption runs with random plaintexts and split the recorded power traces into two groups, one with $v = 1$ and the other with $v = 0$. To extract the key from the recorded power traces the attacker has to correlate the measured data to calculations based on a key hypothesis. For each encryption run an attacker can calculate v, iterating through all 256 possible values of the first round key byte. A simple type of correlation is building the difference between the mean of all power traces that the attacker calculated to have $v = 0$ and those with $v = 1$.

For every wrong key guess, the difference between the power traces is mostly random, so the grouped plots cancel each other out. If there are peaks in the difference, there is a dependency between the calculated value v and the value actually processed in the device. Large peaks should occur only for the correct key guess, thus allowing an attacker to identify the correct round key. The same recorded power traces can be reused to recover the following round key bytes by following the same procedure, and finally to recover the real key from the round keys.

An attack that exploits only one intermediate value is called *first-order* or *univariate* attack, while attacks exploiting the joint leakage of several intermediate values are called *higher-order* or *multivariate* attack.

2.2.2 Power models

Without going into details of the power consumption of electric circuits and logic cells, this section aims to give a basic understanding on why devices leak usable information

and how they can be condensed to a power model. More detailed information can be found in [MOP07].

While a simulation on analog level or logic level, i.e. the simulation of every single component of a device, is the most precise way to simulate the power consumption, it requires a very detailed knowledge of the device and is a complex and time-consuming task. To make a side-channel attack feasible, normally very simple models are used instead, which is possible because an attacker needs to know only relative differences in the power consumption.

As an attacker is interested only in the data-dependency and operation-dependency of a device, every power consumption that does not contribute to the relevant leakage appears as electric noise to him. Often the influence of one relevant part of the device such as a data bus is very strong and allows simple assumptions that can quickly give an rough estimation of the power consumption.

Hamming Distance The Hamming-Distance (HD) model does not measure the absolute power consumption, but only the number of bit flips that occur in a certain time interval for example on a data bus or in a register. It assumes that there is no difference in the power consumption of $0 \rightarrow 1$ and $1 \rightarrow 0$ transitions and that the consumption is distributed equally over all cells. The typical design of microcontrollers promote the validity of this assumption and makes the HD model suitable for most attacks on devices that are not specially protected against side-channel attacks.

Hamming weight The Hamming-Weight (HW) model can be used if the attacker does not know consecutive data values that are transfered on the bus and thus can not apply the HD model. In this case he simply assumes that the power consumption is proportional to the Hamming weight of a data value, which is even less accurate than the HD model but still reasonably well working.

More exact models can be found for specific devices if necessary, e.g. in case that each data value leads to a different power consumption or if countermeasures rendered the standard models useless for a given number of measurements by lowering the signal-to-noise ratio.

2.2.3 Countermeasures

Apart from changing keys so frequently that it is just impossible to get a number of power traces that is high enough, it is possible to counteract side-channel attacks by changing the electrical design or software of a device with the goal to remove any dependency between power consumption and processed data. This can be done by either preventing the device from leaking information or by randomizing the intermediate values of the

used algorithm so that the leakage is useless to an adversary. These concepts are called Hiding and Masking.

Preprocessing functions and Higher-order attacks are typical methods that help an adversary to mount attacks on protected devices.

Although not discussed in this paper, the existence of template attacks [CRR03] should be noted, which are considered to be the strongest known form of side-channel attacks, allowing to defeat a range of countermeasures. The major drawback is the necessity of a training device that the attacker can use in a profiling step before the actual attack.

Hiding Two different approaches are commonly used to hide the leakage of a device:

- Forcing equal power consumption for all operations and all data values
- Randomizing the power consumption for each clock cycle

Both approaches cannot be reached in practice, but complicate an attack. Hiding can affect the time dimension or amplitude of the power consumption, both of which types can be implemented in software or hardware. Further details and hints on the effectiveness of this measures can be found in [MDSM02, MOP07].

Affecting the time dimension destroys the correct alignment of power traces, which is important for DPA attacks. It can be accomplished by shuffling operations of the algorithm that can be executed in arbitrary order or by inserting a constant but random number of dummy operations at different positions. This is possible at software level as well as in hardware, but does "not provide a high level of protection" according to [MOP07]. Manipulating the clock signal or randomly skipping clock cycles are further alternatives for changing the time dimension.

There are several approaches at different architectural levels to affect the amplitude dimension. At lowest level, the application of specially crafted logic cells - e.g. Dual Rail Precharge (DRP) logic which decodes information in differential pairs - can help to reduce the leakage signal. Alternatively the noise can be increased by performing several independent or even useless random operations in parallel, or by inserting filters in the path of the power supply. At software level implementations can be optimized by choosing synonymous operations that leak less information. All this approaches try to lower the Signal-To-Noise ratio.

Masking Masking internally combines intermediate values v with a random value m that varies from execution to execution, e.g. $v_m = v \oplus m$.

It does not prevent leakage, but tries to make it useless to the attacker who does not know the random value. It is usually implemented at algorithm level (hence the algorithm needs to be slightly but not functionally changed), but can also be applied at the cell level. Every intermediate value needs to be masked, and every mask needs to be

removed at the end of the cryptographic operation. Increasing the number of different mask values decreases the performance, hence the number needs to be chosen carefully, particularly in the case of complex non-linear operations such as S-Boxes.

In asymmetric schemes the application of arithmetic masks is called blinding. There exist security proofs for masking schemes that force the masked intermediate values to be independent of both the mask and the original intermediate value; see for example [BMK04, OMPR05, GT03]. Higher-order masking schemes can be used to increase the security, but are effective only if combined with a sufficient amount of noise, as researches published in [SVCO+10] showed.

2.3 Mutual Information Analysis (MIA)

With the notion of a "generic information-theoretic distinguisher for differential side-channel analysis" the authors of [GBTP08] establish a model that is designed to work without any knowledge of the electrical characteristics of the attacked device. Contrary to the typical approach of reducing the number of measurements by enhancing the leakage models, MIA increases the number of measurements to take the advantage of a generally usable model that is said to work effectively in realistic scenarios, at the cost of losing efficiency due to the lack of assumptions.

MIA seems to be particularly promising for higher-order attacks because it is easily adaptable to the multi-dimensional case and allows the detection of arbitrary relationships between measured traces and expected leakage.

The generalization of MIA to higher order attacks was carried out by Prouff and Rivain [PR09], who argue that it is the most efficient known higher-order attack on masked implementations. On the other hand, MIA performs worse than existing standard methods if the leakage is a linear function of the predicted leakage. This goes along with the conclusion of Amir Moradi *et.al.* [MMPS09] who worked out that MIA performs worse for the standard one-dimensional case, as it needs more traces and is more affected by noise.

While Veyrat-Charvillon and Standaert claim in their paper "Mutual Information Analysis: How, When and Why?" [VCS09] that MIA is useful for the one-dimensional case at least if the attacker is not able to find an sufficiently precise leakage model, the authors of [MMPS09] found no evidence that MIA can "work efficiently without any knowledge about the leakage function of the target device" and refuse the idea of an universal attack that works for any unknown device.

An undoubted advantage of MIA is the fact that it does not rely on the Gaussian assumption, i.e. the noise in the leakage samples is not assumed to be additive and its random part does not need to be normally distributed with mean zero and variance σ^2. The Gaussian assumption typically holds for standard DPA attacks, but often does not

apply when countermeasures such as masking are used. This was inquired in a paper named "Unifying Standard DPA Attacks" [MOS09] that confirms the proposition of [GBTP08] that MIA is of greater interest especially in the case of devices employing DPA-resistant logic. On the other hand the same paper states that "for standard univariate DPA attacks [...] the most popular methods are in fact equally efficient" once a good leakage model is found. [SVCO+10] stresses that this conclusion does not hold for multivariate attacks.

Moreover MIA is not limited to distinguishing keys, but has also been proposed as a metric to compare different cryptographic devices [SMY09].

2.3.1 Information Theory

This section introduces some information theoretic definitions.

Entropy Entropy measures the uncertainty of an random variable X on a discrete space $\mathcal{X}$. It is defined as:

Definition 2.3.1

$$H(X) = -\sum_{x \in \mathcal{X}} Pr(X = x) log_2[Pr(X = x)]$$

The uncertainty of a combination of two random variables is called joint entropy and defined as:

Definition 2.3.2

$$H(X, Y) = -\sum_{x \in \mathcal{X}, y \in \mathcal{Y}} Pr(X = x, Y = y) log_2[Pr(X = x, Y = y)]$$

It is equal to the single entropy if Y is a deterministic function of X, and greater otherwise. It holds that $H(X) \leq H(X, Y) \leq H(X) + H(Y)$ if X and Y are independent.

The conditional entropy of a random variable X given an variable Y describes the remaining uncertainty if Y is known.

Definition 2.3.3

$$H(X|Y) = -\sum_{x \in \mathcal{X}, y \in \mathcal{Y}} Pr(X = x, Y = y) log_2[Pr(X = x|Y = y)]$$

Mutual Information Mutual Information measures the dependence between two random variables, thus expressing the reduction of uncertainty about X gained by observing Y. Since it is a symmetrical value, it can be computed in two ways:

Definition 2.3.4

$$\begin{aligned} I(X,Y) &= H(X) - H(X|Y) = H(X) + H(Y) - H(X,Y) \\ &= H(X,Y) - H(X|Y) - H(Y|X) = I(Y,X) \\ 0 &\leq I(X,Y) \leq H(X) \end{aligned}$$

The above equation can be rewritten by replacing every occurrence of entropy functions with the previous definitions. An extensive derivation of these equations can be found in [FFS10].

The equations presented here are valid for discrete values, but can easily be extended to the continuous case. X_i and Y_i are used as short notation of the probability densities $p(x_i)$ and $p(y_i)$.

Definition 2.3.5

$$I(X,Y) = \sum_i X_i(\sum_j Y_i(j)\ log_2(Y_i(j))) - \sum_j(\sum_i Y_i(j)X_i)\ log_2 \sum_i Y_i(j)X_i \tag{2.1}$$

$$I(X,Y) = \sum_i X_i(\sum_j Y_i(j))\ log_2 \frac{X_i Y_i(j)}{\sum_k Y_k(j)X_k}) - \sum_i (X_i\ log_2(X_i)) \tag{2.2}$$

2.3.2 Concept

MIA is based on a side-channel model that describes an attack by means of a leakage function L, a physical observable O, a state transition W and a secret key k from a key space $\mathcal{K} = \{0,1\}^m$. O represents an usually noisy measurements of L, which is estimated as $\hat{L}$ by the attacker. O depends on L which depends on W, i.e. the leakage is observable in the form of state transitions (e.g. bit flips) in the device. As L depends on the key k, L can be explicitly denoted L_k. Figure 2.2 illustrates this concept.

As in standard DPA attacks, the adversary obtains a number of power traces by measuring $O(t)$ while the device processes varied known data x_i (e.g. random plaintexts) together with the key k to compute the targeted intermediate result. Then he guesses the subkey k^* that is a part of k, implying a guess on $\hat{L}_{k^*} = \hat{L}(W_{k^*})$ which is then compared to the real leakage function L. The attacker is looking for the key guess leading to the maximum dependency (i.e. the highest mutual information) between the estimated leakage function and the observed measurements:

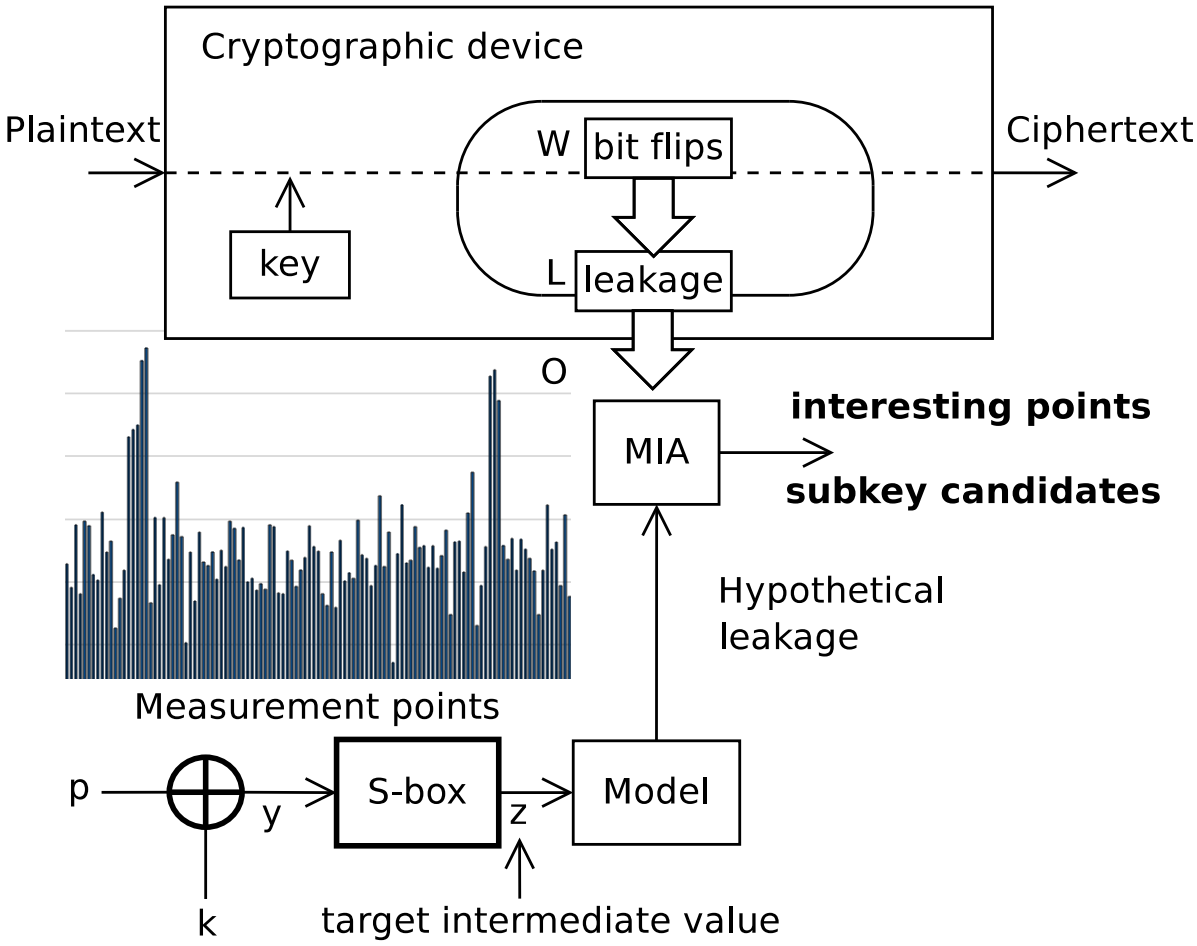

Figure 2.2: Concept of Mutual information analysis, based on figures in [MMPS09, MOS09]

Definition 2.3.6 *$\hat{I}(\hat{L}_{k^*}, O) = \hat{H}(O) - \hat{H}(O|\hat{L}_{k^*})$, where $\hat{H}$ denotes an estimated entropy.*

MIA models the attack by defining a distinguisher $\mathcal{D}$ that outputs a correct key guess given the measurements o_{x_i} and the input values x_i with some probability. MIA also defines an extended distinguisher that also extracts the interesting points in time of the given measurements.

Definition 2.3.7 *$\mathcal{D}(o_{x_i}, x_i) \rightarrow k^*$ iff $\hat{I}(\hat{L}_{k^*}, O) = max_{\hat{k}}(\hat{H}(O) - \hat{H}(O|\hat{L}_{\hat{k}}))$*

Finding an appropriate intermediate value is usually an easy task that depends only on the cryptographic algorithm and not on a specific device. However, choosing $\hat{L}$ is non-trivial and important, because efficiency and effectivity of the MIA distinguisher depend on $\hat{L}$ being close to L. For an unknown L, the adversary first needs to ensure that the hypothetical leakage function $\hat{L}$ does not produce collisions where L does not, hence it is reasonable to choose $\hat{L}$ as a bijective mapping of W, at the cost of losing efficiency because this setting can produce less collisions than L. Furthermore, different key hypotheses must not yield merely permutations of $\hat{L}_{\hat{k}}$, because then $\hat{I}(\hat{L}_{\hat{k}}, O)$ would be independent of $\hat{k}$.

MIA transposes the complexity of choosing a device-specific and fine-tuned $\hat{L}$ to the estimation of a key-dependent probability density function empirically determined by sampling the observed leakage. In the original concept of MIA the estimation relies on histograms and needs very few assumptions, but other statistical methods that may need more assumptions can lead to greater efficiency as [VCS09] points out.

Finally the actual measurements have to be correlated to the predicted probability density function, which is done by the distinguisher by means of Mutual Information. Again, different solutions[2] are possible, with Kullback-Leibler divergence being one of the mentioned alternatives in [VCS09]. The genericity of the framework and diversity of compatible statistical methods that can be "plugged into MIA" motivates seeing MIA as a toolbox, as Veyrat-Charvillon and Standaert stress in their paper.

2.4 Parallel computing on graphics processing units (GPU)

Graphics Processing Units (GPU) are traditionally used to offload graphics rendering from the main processor, but their highly parallel structure and their dedication to fast

[2]It is even possible to correlate the samples directly without an explicit estimation of a probability density function, see [VCS09].

arithmetic stream processing without the overhead of general-purpose CPUs makes them efficient platforms for different types of algorithms. Since there was a growing demand to use the performance offered by GPUs for non-graphical purposes, the largest companies in the field of GPU design began to provide development frameworks that allow running programs on the GPU or - with OpenCL - even on heterogeneous platforms consisting of several types of processors. This type of usage is also called General-purpose computing on graphics processing units (GPGPU).

As the development of CUDA is in a more mature stage than for example OpenCL, we chose CUDA as basis for this work hoping to avoid unnecessary hassle due to framework bugs and limitations, at the price of being bound to one vendor. Converting the code to OpenCL later is possible with a reasonable amount of work that can be partially automatized [Har09b].

2.4.1 Compute Unified Device Architecture (CUDA)

CUDA is a framework for developing software for Nvidia graphic cards, effectively enabling developers to use Nvidia GPUs for arbitrary tasks by giving them access to the native instruction set and GPU memory. Beside 'C for CUDA', which is used for this thesis, there are several other interfaces to access CUDA-enabled GPUs.

As the performance improvement gained from utilizing GPUs mainly stems from fast data-parallel computations, only applications that execute the same computationally intensive program on many data elements in parallel profit from CUDAs benefits.

The bus bandwidth and latency between GPU and CPU constitutes a bottleneck, hence the performance is best if the kernels operate on the same data for a long time before new data needs to be transferred to the GPU. [CUD09a] gives more details on which types of application profit most from the CUDA architecture and how to gain the maximum performance.

Following the term "multi-core", Nvidia named their CUDA architecture "many-core" to illustrate the fact that each GPU can have hundreds of cores that can each run thousands of threads. GPU-threads are designed to be very lightweight compared to CPU-threads, because thread resources stay allocated to the thread for the complete execution time instead of being swapped to the currently active thread. The resources of each core are shared, with the on-chip shared memory allowing parallel tasks to share data without sending it over the system memory bus.

Recent devices have not only more resources than older ones, but also different *compute capabilities*, expressed as a version number that can be accessed along with the device properties by using CUDA's runtime device management functions. For example, before compute capability 1.3 no double-precision arithmetic or atomic functions were available.

2.4.2 Kernels and thread hierarchy

Kernel functions are functions in standard C code that are executed on the GPU instead of on the CPU. They are declared by using the `__global__` keyword, and execute N times in parallel when called. Nvidia introduced the *kernelname* <<<`Gridsize, Blocksize`>>> (*args*) syntax to facilitate the execution configuration for kernel calls, which amongst others defines the number of threads that should be launched for a kernel.

CUDA threads are organized hierarchically into warps, blocks and grids, with warps being the smallest group of threads, controlled by a warp scheduler. Each thread has its own instruction pointer and register set and is identified by an unique *thread ID* that can be used to determine the data that this a thread is working on.

Warps are running most efficiently if all of their threads follow the same execution path because they can execute common instructions in parallel; in case of intra-warp divergent branches, the warp executes each path serially until all threads converge back to the same execution path. The warp-size is 32 in todays architecture.

Threads are organized in three-dimensional blocks that can communicate via shared memory and all reside on the same processor core. A block is identified by its *block id* and holds on current devices up to 512 threads, starting with thread id 0 for every block. Threads in one block can be synchronized by using the `__syncthreads` function.

Multiple equally-sized blocks form a two-dimensional grid, which is not limited by the number of processors. The scheduler executes blocks serially or in parallel and in arbitrary order, scaling automatically to the number of cores in the used device. Hence, the blocks must be independent and cannot be synchronized.

The total number of threads is usually determined by the size of processed data. Thread id, block id and block and grid dimensions are accessible to the programmer as built-in variables, which can easily be used to calculate a thread id across all threads of a grid. A host system can have multiple GPUs, but kernel execution does not scale across all devices. Each device needs to be controlled by a separate host thread and runs independent of all other devices.

2.4.3 Memory management

CUDA kernels can access different memory regions, from global memory - which is accessible by all CUDA threads as well as by the host via the CUDA runtime functions - to private local memory accessible by only one thread. Additionally there are constant and texture memory spaces which are read-only and optimized for special access patterns. Textures are not used as part of this work, hence we will not go into detail.

- Registers: Each multiprocessor has a number of 32-bit registers, which are partitioned among all threads, e.g. 32 per thread. Registers are private to one thread and provide low latency memory.
- Shared memory: Used as a small (e.g. 16kb) but fast user-managed cache layer for global memory, located on-chip. Shared memory is divided into memory banks that can be accessed simultaneously; bank conflicts occur if two addresses of memory requests fall into the same memory bank and must be avoided by the programmer in order to avoid serializing shared memory access.
- Global memory: Provides large quantities of memory on the device, at the price of having a high latency. Furthermore, alignment requirements to 32-, 64- or 128-bit words must be satisfied for full efficiency.
- Local memory: Residing in device memory like the global memory, and thus having the same restrictions and same high latency, local memory is used mainly if a kernel needs more registers than available.

Together with the number of threads per block, the number of registers and the amount of shared memory required by a kernel determines the number of resident warps on one multiprocessor, and hence the occupation. The CUDA Occupancy Calculator provides the means to easily calculate the setup that will achieve the best results, according to the targeted compute capability of the devices.

The CUDA runtime provides functions to allocate *page-locked* (or *pinned*) host memory. Page-locked memory allows memory copies between host and device to be concurrently with kernel executions and provides a higher bandwidth on systems with a front-side bus, but cannot be swapped out by the operating system, thus possibly reducing the performance if insufficient memory is available.

On devices with compute capability greater than 1.2 it is possible to map page-locked host memory to device memory. Data is transfered implicitly from host memory to global memory as needed by the kernel.

If a program uses memory that is never read but written from the host's view, flagging it as not cacheable by allocating it as *write-combining* memory can further improve the performance.

The program `bandwidthTest` from the CUDA-SDK allows to test the memory bandwidth for different parameters. Figure 2.3 plots the data rate of a host-to-device transfer with write-combining pinned memory. On a GeForce GTX 295, best performance is achieved for transfers of 40 MB size with a transfer rate of 5739 MB/s and nearly the same (5696 MB/s) for 4 MB. Smaller memory chunks perform worse due to the overhead, ranging from 3378 MB/s for 64 Kb to 5525 MB/s for 512 Kb. For comparison, on a GeForce GTX 290 the maximum is 2904 MB/s for 30 MB and 2796 for 512 Kb or 2205 MB/s for 64 Kb.

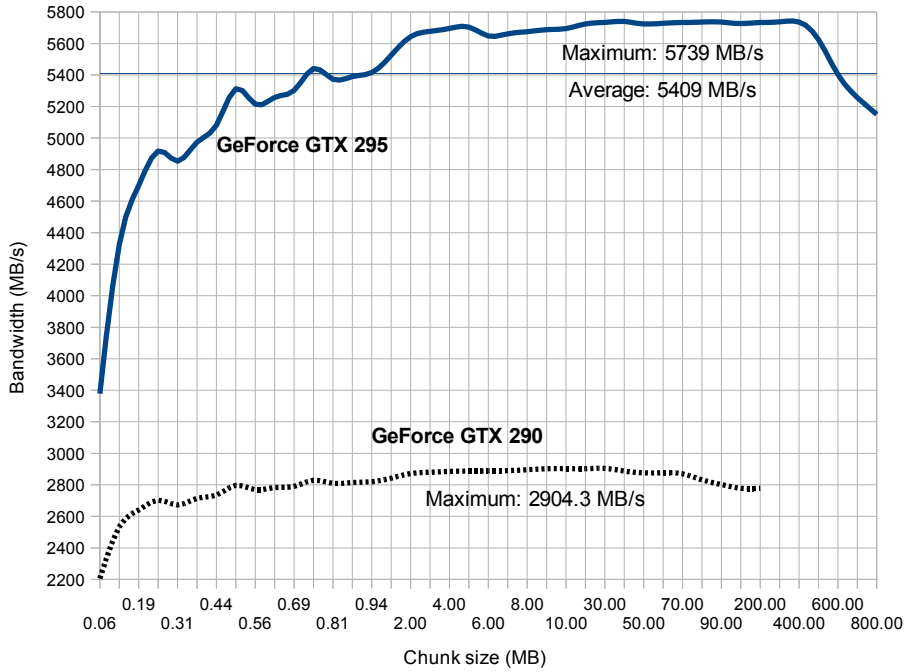

Figure 2.3: Memory bandwidth for a host-to-device transfer with write-combining pinned memory

2.4.4 Asynchronous execution and memory transfers

Kernel launches as well as the set of asynchronous memory copy functions return control to the host thread before the device has completed its task. Concurrent kernel execution is possible on some devices with compute capability 2.0, which were not available for this work, but data transfers on page-locked memory can perform concurrently on some devices since compute capability 1.1, thus considerably improving the performance in many cases.

Concurrency is managed in *streams* that identify commands that need to execute in order with an stream parameter passed to kernel calls and memory copy functions. Commands of different streams can execute concurrently; e.g. if a kernel call of one stream needs to wait for a memory copy operation, a kernel of another stream may run while the memory copy is still unfinished. A typical pattern for using streams is shown in algorithm 1.

2.4.5 Multiple devices

As part of the implicit initialization of the CUDA runtime performed by the first call to a non-managing CUDA runtime function, a CUDA context is created which is current

Algorithm 1 Stream usage pattern

```
cudaStream_t stream[N];
for(int i = 0;i<N;i++)
    cudaStreamCreate(&stream[i]);
for(int i = 0;i<N;i++)
    cudaMemcpyAsync(device_memory + i * SIZE, host_memory+ i * SIZE, SIZE,
        cudaMemcpyHostToDevice, stream[i]);
for(int i = 0;i<N;i++)
    somekernel<<<grid, block, 0, stream[i]>>>(device_memory + i * SIZE);
for(int i = 0;i<N;i++)
     cudaMemcpyAsync(host_memory+ i * SIZE, device_memory + i * SIZE, SIZE,
          cudaMemcpyDeviceToHost, stream[i]);
// wait for all streams to finish
cudaThreadSynchronize();
// or just destroy them (automatically waits until finished)
for(int i = 0;i<N;i++)
    cudaStreamDestroy(stream[i]);
```

only to the calling host thread and restricted to the one used device. To use multiple devices, multiple host threads are needed, each creating their own CUDA context for one of the devices. C for CUDA does not support assigning a context to another host thread.

3 Implementation overview

Although MIA as a generic distinguisher or "toolbox" is perfectly suitable for an abstract modular programming concept, we take another approach for this work in order to exploit all possibilities to optimize a single way of implementing MIA. Consequently we decide to use histograms to sample the observed leakage, thus avoiding any assumption about the unknown distribution.[1] Secondly we decide to use hamming weight as leakage model in order to reduce the number of needed histograms. Later works may expand this to one histogram per oscilloscope value to use more of the available information.[2]

Our target is a generic AES hardware implementation from which we will take approximately one million traces with 20000 data points each.[3] The traces are measured with a digital oscilloscope with 8-bit resolution, thus providing 256 bins for our histograms.

Our application will essentially deal repeatedly with two steps:

- Calculate the hypothetical leakage for each key and sort it into the corresponding histogram.
- Calculate the mutual information.

The first step involves a lookup table and many increments of histogram bin counters, while the second step consists mainly of entropy calculations, i.e. sums and logarithms base two. For an efficient implementation for CUDA, the most important step is to find an optimal parallelization procedure.

In the case of CUDA it is also very important to obey Nvidia's programming recommendations with respect to memory types, memory bank conflicts and other CUDA-specific issues. Before going into detail in chapter 5, we will describe a reference implementation, with the purpose to compare the efficiency of the CUDA implementation and to reduce the total runtime by combining the power of GPU and CPU.

[1] Parametric probability density function estimations tend to be more accurate than generic estimations like histogram approximation; this difference is quantified in [FFS10].

[2] However, [MMPS09] observed that using MIA with high numbers of bins does not deliver good results, but increases the computational overhead more than the same number of bins using for example CPA.

[3] Our architecture scales well for larger amounts of data.

3.1 Prerequisites

Before we discuss the implementation at full length, some basic ideas and terms need to be established.

Constants. We define preprocessor constants for a few often appearing numbers. Wherever this paper assumes concrete numbers without mentioning them explicitly, these numbers are used.

- POINTS: Number of measurement points, e.g. 20000
- TRACENUMBER: Number of traces for each measurement point, e.g. 2^{20}
- HW: Number of Hamming weight values (0-8), i.e. 9
- BINS: Number of histogram bins, i.e. 256
- KEYS: Number of key values for one key byte, i.e. 256
- POINTS_PER_WORKUNIT: Number of measurement points combined to one workunit, e.g. 3

Distributing the workload. As the analysis of every measurement point is completely independent of all other points for univariate attacks, it stands to reason that giving one point at a time to each device or CPU thread is a simple and efficient method of distribution. This task is accomplished by a simple WorkManager class that atomically increments a counter of processed points, and assigns a number of points to each requesting device or thread. This number of points is also denoted as workunit and a single point as subworkunit.

Main function - initialization and threading. As discussed in paragraph 2.4.5 (multiple devices), each GPU needs its own CPU thread to manage its context. Hence, the total number of needed threads (`maxthreads`) is the number of desired stand-alone CPU-threads plus the number of used devices. We use OpenMP[4] as shared memory multiprocessing API for creating threads and managing thread access to critical sections such as incrementing the workunit counter. Algorithm 2 presents a simplified version of the main function that basically shows how the threads are created for each device and for the stand-alone CPU threads. Details on the allocated memory are given in section 4.1.

[4]Open Multi-Processing, http://openmp.org/

Algorithm 2 MIA main function

```
// Initialize WorkManager, get CUDA device count and
// allocate memory for miaresult and miaresult_who

// Iterate over all CUDA devices AND standalone-CPU threads
#pragma omp parallel for num_threads(maxthreads)
for(int device = 0; device < maxthreads; device++)
{
  // Use low thread-ids for GPU threads
  if(device < deviceCount)
  {
    // Skip devices with compute capability < 1.3
    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties(&deviceProp, device);
    if(deviceProp.major>=1 && deviceProp.minor >= 3)
      MIAonGPU(work, tracefile, device, miaresult, miaresult_who);
  }
  else if(cpuUsed<CPU_COUNT)
    MIAonCPU(work, tracefile, device, miaresult, miaresult_who);
}

// All threads finished, now output the results
for(long int i=0;i<POINTS;i++){...}
```

Data and file format. Both the plaintext input to the AES device and the corresponding oscilloscope output are needed for the attack. Both have a size of 1 byte, but accumulate to a data size of $2 \times 20000 \times 2^{20}$ Bytes $= 41943040000$ byte (40 GB). Hence we will read the data from a file and hold only the portion of data needed for the current workunits in memory. The file format was chosen to be a simple sequence of input and output data blocks: [T bytes input data for point 0][T bytes output data for point 0]...[T bytes input data for the last point (P-1)][T bytes output data for the last point (P-1)]

Simulated traces. At the time of writing, our implementation was tested only against simulated power traces. The input values are randomly generated bytes and the output values are defined as Hamming weight of the AES S-Box output, which was chosen either completely random (for measurement points unrelated to the secret key) or according to the XOR of input value and secret round key. Gaussian noise is added to the resulting value as desired.

Algorithm 3 shows the main part of the trace generator. The AES S-Box lookup-table found in [AES01] has been replaced with their Hamming weights, e.g. 0x63 becomes 4, 0x7c becomes 5, etc. Simulation has only been implemented for the first round of AES.

Algorithm 3 Trace simulation

```
for (long int i = 0; i < tracenumber; i++)
{
  unsigned char r = rand() & 0xff; // random plaintext byte
  inputvalues[i] = r;
  if(keydependent) // use S-Box to calculate leakage, add noise
      outputvalues[i] = SboxHamming[ r ^ key ] + ((rand() & MIA_NOISE)-(
          MIA_NOISE/2));
  else // random input to the S-Box
      outputvalues[i] = SboxHamming[ rand() % 256];
}
```

Output, timing and debugging. The raw results along with information on the used parameters are dumped to `stdout` and analyzed by a small python script afterwards. For timing and debugging, additional information can be printed to `stderr` by setting a compiler flag with a negligible performance loss. Another python script analyzes the timing information during the runtime or afterwards and extract step-by-step statistics, e.g. the average, minimum and maximum time to load data from the tracefile or to process a whole workunit. Timings rely on the Cutil library that is issued by Nvidia in addition to the CUDA toolkit.

Linux bash scripts were written to automate the process of producing statistics that vary parameters and collect average timing information from repeated runs. Their output was mainly processed and graphically evaluated with the help of standard office software.

Testing environment. The setup for testing our application consists of a Linux Ubuntu 9.10 server running on two Intel Xeon E5540 CPUs (2.53 GHz) with a total of 8 CPU cores, equipped with a total of 32 GiB RAM, two Nvidia GeForce GTX 295 graphic cards (each providing two GPUs) and one Quadro NVS 290, which remains unused due to its low compute capability. Each GTX 295 GPU has 30 multiprocessors, 240 cores, is clocked at 1.24 GHz and 939327488 bytes (ca. 895 MB) device memory. Each of the 30 multiprocessors provides 16384 32-bit registers and 16 kb of shared memory. The compute capability of the GTX 295 is 1.3, the CUDA driver version is 3.0 with a CUDA runtime version is 2.3.

4 Reference implementation in C

The C implementation is largely a straight-forward implementation of MIA, with optimizations only in the mutual information calculation. Since the combination of multiple measurement points to one workunit has no (positive or significant negative) effect on the performance of MIA on CPU, we assume the workunit size to be exactly one measurement point (i.e. POINTS_PER_WORKUNIT = 1) in this chapter for simplification.

4.1 Memory requirements

- Input (plaintext) and output (oscilloscope) data: Only one measurement point is processed at a time, so we need memory for exactly one subworkunit, i.e. 2^{20} byte for input as well as output values. A variable `unsigned char * iodata` stores both with one memory allocation to allow loading the data from the tracefile in one step. `input` points directly to `iodata` and `output` points to `iodata` with an offset of TRACENUMBER bytes. Overall size is $2 \times 2^{20} = 2097152$ bytes (2 MB).
- We need `HW=9` histograms with `BINS=256` integer bins. Hence `unsigned int * histogram` has a size of 9216 bytes (9 kb).
- Result memory needs to be allocated for POINTS times KEYS results in double format. This is done once in the main function for all threads, so no unneeded memory space is allocated. Consequently `double * miaresult` needs $20000 \times 256 \times 8 = 40960000$ bytes (40 MB) memory.
- `int * miaresult_who` stores which thread processed which measurement point. This requires $20000 \times 4 = 80000$ bytes (80kb) of memory.

This poses no problems even for much higher numbers of points and traces on every standard computer system.

4.2 Implementation overview

Algorithm 4 gives a rough overview of the application flow. The main loop starts by requesting a new workunit from the WorkManager, stores its thread-id in `miaresult_who` and loads the corresponding data from the tracefile by calculating the correct file

offset. Loading the data for one workunit takes less than 1 ms on average which is negligible compared to the total average time of 650 ms for one workunit, mainly caused by the histogram generation (256×2.5 ms). Hence pre-loading the data for the next workunit while working on the current is not worth the effort, the more so as an similar effect can be achieved simply by using more CPU threads than available CPUs.

In the next step a pointer to the application-wide visible `miaresult` memory is declared with an offset that points to the memory allocated for the current workunit.

Afterwards a loop over all possible values (0-255) for one byte of the round key is started, which first of all sets the histogram memory back to zero. A third loop iterates through all TRACENUMBER traces, calculates the XOR of input (plaintext `input[i]`) and key guess and looks up the hypothetical leakage in SboxHamming. There is one histogram for each possible leakage value (0-8 for Hamming weight leakage), and the next step increments the counter of the bin of the oscilloscope output value (`output[i]`) that corresponds to the current plaintext and leakage.

After the third loop finishes, mutual information is calculated based on the histogram as detailed in the next section, and the result is saved to `miaresult` memory.

4.3 Mutual Information

As shown in definition 2.3.5, mutual information is a symmetric value and hence can be calculated two ways. Experimental results - supported by the intuition that less nested sums might be more efficient to implement - suggested to use equation 2.1 for our implementation. In the following we will adapt and modify it to allow an efficient calculation that will also suit the demands of parallelization.

Let O_{hw} be the histogram-approximated probability density of the leakage observed through oscilloscope samples sorted by a hypothetical leakage function (Hamming weight) into histogram hw, where $O_{hw}(x)$ refers to the probability of a single histogram bin x. Let S be the probability distribution of the leakage values, were $S(hw)$ refers to the probability of one of the possible values hw (0-8). Definition 4.3.1 presents the resulting equation derived from equation 2.1.

Definition 4.3.1

$$I(S,O) = \sum_{hw} S(hw)(\sum_{x} O_{hw}(x)\ log_2(O_{hw}(x))) - \sum_{x}((\sum_{hw} O_{hw}(x)S(hw))\ log_2(\sum_{hw} O_{hw}(x)S(hw)))$$

Algorithm 4 MIAonCPU - MIA C implementation overview

```
int workunit;
while((workunit=work->get()) != -1)
{
    // remember who processed this workunit
    miaresult_who[workunit]=threadid;

    // Load data (input/output) for t=workunit from file
    long int file_offset=workunit*TRACENUMBER*2;
    loadOscilloscopeDataFromTracefile(...);

    // Pointer to miaresult with correct offset for workunit
    double * miaresult_workunit= miaresult + workunit*KEYS;

    // Loop over all keys
    for(int key=0;key<KEYS;key++)
    {
        // Set histogram to zero
        memset(histogram, 0, HW * BINS * sizeof(unsigned int));

        for(long int i=0;i<TRACENUMBER;i++)
        {
            // Calculate hypothetical leakage for key guess...
            unsigned char leak = SboxHamming[ input[i] ^ key ];
            // ... and increment oscilloscope outputdata-bin in histogram
                for 'leak'
            histogram[leak][output[i]]++;
        }

        // Calculate Mutual Information
        double temp_mi = mutualinformation(histogram);
        miaresult_workunit[key] = temp_mi;
    }
}
```

Algorithm 5 Distribution of Hamming weights of S-Box output with random input

```
// Calculate hamming weight distribution
void printHammingDistribution()
{
    int distribution[HW]={0,0,0,0,0,0,0,0,0};
    for(unsigned int i=0;i<256;i++)
    {
        distribution[hammingweight(i)]++;
    }
    for(unsigned int i=0;i<HW;i++)
    {
        printf("%f, ",distribution[i]/(double)256);
    }
}
```

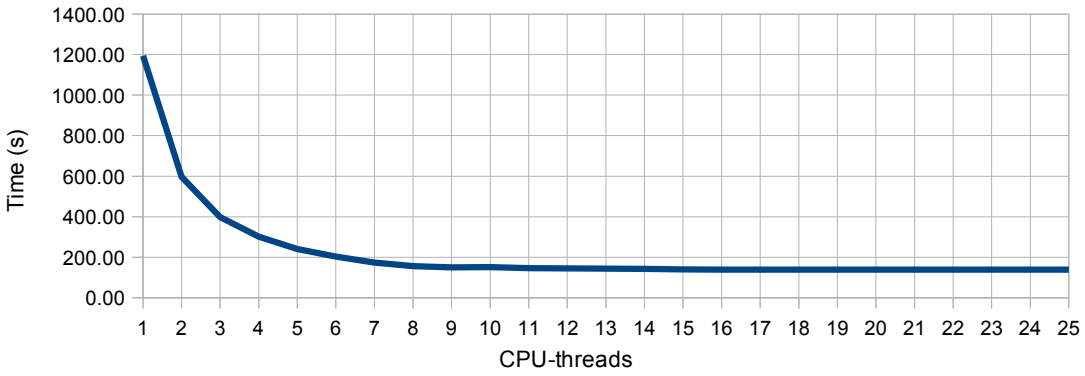

Figure 4.1: Timing of MIA with 2000 measurements points. The minimum time is needed for 23 CPU-threads with a runtime of 138.8 seconds

Since the input to the S-Box can be assumed to be random due to the random plaintext input, S can be precomputed as shown in Algorithm 5.

Moreover we are interested only in *comparing* mutual information values, which does not requires us to know the correct absolute mutual information value. Hence we do not need to employ true probabilities (ranging from 0 to 1) but can use the unnormalized histogram bin counters directly. Instead of calculating $O_{hw}(x) = \frac{O^*_{hw}(x)}{\text{bincountersum}}$ we save a little time by using the bin counter value $O^*_{hw}(x)$. The results can be scaled and compared afterwards as necessary, as done in our python result script.

The resulting algorithm is shown in algorithm 6.

4.4 Results

Although it is probably possible to optimize our CPU-only implementation of MIA, it is suitable for our purpose: it forms a useably data basis for later comparison with the GPU-assisted implementation and it will improve the runtime for combined CPU-/GPU-runs.

Figure 4.1 shows a timing for 2000 measurements points. As expected, experiments showed (see Figure 6.1 in chapter 6) that increasing the number of points increases the runtime by the same factor, so that the runtime for 2000 points allows to assess the runtime for 20000 points.

Listings 1 to 3 present statistics for different numbers of CPU-threads and show timings for several code fragments. The run count of a fragment is indicated in brackets.[1]

[1] The statistics were provided for a run with 200 measurement points. This results in 51200 executions of the histogram and mutual information code, because each workunit is processed for 256 keys, hence $200 \times 256 = 51200$. Each execution of the histogram code creates 9 (HW) histograms.

Algorithm 6 Mutual information on CPU

```
// Fast log2: precompute 1/log(2)
#define LOG2M_CONST 3.32192809488736234787031942949 48
#define _LOG2(x) (log(x) * LOG2M_CONST)

// Calculated hamming weight distribution by printHammingDistribution()
static const double S[HW] = {0.003906, 0.031250, 0.109375, 0.218750,
    0.273438, 0.218750, 0.109375, 0.031250, 0.003906};

// histogram holds 9 histogram arrays and corresponds to O_hw
double cpu_mutualinformation(unsigned int *histogram)
{
  double mutualinformation = 0;
  unsigned int hw,x;
  double temp, tempSum;

  // \sum_{hw}^{} { S(hw) (\sum_{x}^{} { O_{hw}(x) log_2(O_{hw}(x)) } ) }
  for(hw=0;hw<HW;hw++)
  {
    tempSum=0;
    // temp = O_{hw}(x)
    // \sum_{x}^{} { temp log_2(temp) }
    for(x=0;x<BINS;x++)
    {
      temp = histogram[hw][x];
      if(temp>0)tempSum+=temp * _LOG2(temp);
    }

    mutualinformation+= S[hw] * tempSum;
  }

  // \sum_{x}^{} ( ( { \sum_{hw}^{} { O_{hw}(x)  S(hw) } ) log_2( \sum_{hw
      }^{} { O_{hw}(x) S(hw) }  ) }  )
  for(x=0;x<BINS;x++)
  {
    // tempSum = \sum_{hw}^{} { O_{hw}(x)  S(hw) }
    tempSum=0;
    for(hw=0;hw<HW;hw++)
    {
      tempSum+= S[hw] * histogram[hw][x];
    }

    // \sum_{x}^{} { tempSum * log2( tempSum) }
    if(tempSum>0)mutualinformation-=tempSum * _LOG2(tempSum);
  }
  return mutualinformation;
}
```

Notice that 82.9% (21 CPU-threads) to 98.7% (1 CPU-thread) of the runtime of one workunit is spend on histogram generation. The runtime of a single workunit varies dependent on the number of used CPU-threads.

Further analysis will be provided in chapter 6 together with the evaluation of the GPU results.

Listing 1 Runtime of MIA on CPU for 200 measurement points and 1 CPU-thread

```
 CPU MIA Thread [     1] Avg:   119883.91ms
   CPU Workunit [   200] Avg:      599.42ms
CPU Subworkunit [   200] Avg:      599.41ms
  CPU Load Data [   200] Avg:        0.54ms
  CPU Histogram [51200] Avg:        2.31ms
         CPU MI [51200] Avg:        0.02ms
```

Listing 2 Runtime of MIA on CPU for 200 measurement points and 8 CPU-threads (equals number of available CPU cores of our test environment)

```
 CPU MIA Thread [     8] Avg:    16194.01ms
   CPU Workunit [   200] Avg:      647.74ms
CPU Subworkunit [   200] Avg:      647.73ms
  CPU Load Data [   200] Avg:        0.91ms
  CPU Histogram [51200] Avg:        2.49ms
         CPU MI [51200] Avg:        0.03ms
```

Listing 3 Runtime of MIA on CPU for 200 measurement points and 21 CPU-threads (yields best overall performance)

```
 CPU MIA Thread [    21] Avg:    13817.52ms
   CPU Workunit [   200] Avg:     1447.55ms
CPU Subworkunit [   200] Avg:     1447.39ms
  CPU Load Data [   200] Avg:        9.37ms
  CPU Histogram [51200] Avg:        4.69ms
         CPU MI [51200] Avg:        0.43ms
```

5 Implementation in CUDA

The basic steps for the implementation of Mutual Information Analysis in CUDA, as well as the usage of our WorkManager construct that distributes the workload, are the same as in the reference implementation. Apart from that, many tasks require careful design decisions that will greatly differ from the reference implementation, due to the small amount of fast shared memory on the one hand and the great number of parallel threads on the other hand.

5.1 Design decisions

Corresponding to the two essential steps pointed out in the reference implementation chapter - first leakage and histogram, second mutual information calculation - we chose two kernels called `histogramkernel` and `miakernel`. Using one kernel for both tasks would slightly reduce the overhead, but complicate the implementation significantly, for example because the two kernels need different kernel parameters to perform optimally.

5.1.1 Histogram kernel

Since the number of threads is independent of the number of available cores or other hardware parameters, it can be chosen according to the requirements of the implementation task. The natural choice here is to set it equal to the number of traces for one measurement point. Hence, each thread is responsible for exactly one data item of each point, starting with loading it from global memory and calculating the hypothetical leakage under varied key guesses, up to the point where the corresponding histogram bin is incremented. Since we deal with 8-bit words and since the smallest readable unit for CUDA is 32 bit, $\frac{3}{4}$ of the memory bandwidth would be wasted if each thread reads only the 8-bit word it cares for. For this reason we modify the principle of *one thread - one data item* to *one thread - four data items*, so we can load and process a 32-bit word[1]. We will see later that there is a need to further modify the principle, but nevertheless we will use it as a starting point for our implementation.

[1] More exactly, two 32-bit words are loaded by one thread, namely 32 bit plaintext input and 32 bit oscilloscope output, so we could change to two data items per thread. This has no advantage over using four data items but would require us to change the data format for input/output data.

With the number of threads being fixed and the number of threads per block being limited to 512 on todays CUDA architecture, the number of blocks and thus the grid dimensions are determined simply by dividing: $\frac{\text{TRACENUMBER} \times \frac{8}{32}}{BLOCKSIZE} = \frac{1048576/4}{512} = 512$. The CUDA occupancy calculator can be used to determine whether this number of threads leads to an optimal occupation of the GPU, based on the resource usage of the block.

While the register usage of the histogram kernel is low and does not affect the occupation, the amount of shared memory needed by 9 histograms (one for each Hamming weight value) with 256 integer bins (4 byte) is a problem: $9 \times 256 \times 4 = 9216$ bytes fit into the shared memory of 16384 bytes, but reduce the occupation to 50% for compute capability 1.3, because under this conditions only one block instead of the possible number of 2 blocks can be processed by one core at the same time.

There are only two alternatives to holding all needed histograms in shared memory:

- Holding the histograms in global memory instead of shared memory, which is prohibitively slow.
- Holding only a fraction of each histogram in shared memory, or holding less than 9 histograms in memory. This would implicate that during the processing of the data, each result that does not belong to a (part of a) histogram that currently is in shared memory is either thrown away or written to global memory instead. Since the writes to the histogram are not uniformly distributed over all bins and all histograms, holding only those parts in shared memory that will be written most often and writing the remains directly to global memory seems promising, but due to the lack of time has not found its way into this implementation. Throwing away results if they cannot be saved to shared memory seems far less promising, since they need to be calculated again later, thus effectively reducing the occupancy not less than using only one block per core does.

For our implementation we chose to use 512 threads per block and hold all 9 histograms in shared memory.

A completely different approach would be to use one thread for each key instead of looping over all 256 keys in each thread. But in this case, each thread would need 9 histograms, thus making it virtually impossible to hold a reasonable part of the histograms in shared memory. Using registers instead of shared memory is also not possible, since only 16384 registers are available per block. Using local memory instead would be as slow as using global memory.

5.1.2 Mutual information kernel

`miakernel` works on an already reduced data set, hence it is clear that we need a different number of threads than for the histogram kernel. Essentially two options seem to make sense:

- Using one block of 256 threads. Each thread is assigned to one bin in each of the 9 histograms. Choosing 9×256 threads (one thread for each bin of only one histogram) would complicate the situation without improving it significantly, because the dataset is quickly reduced, thus rendering the additional threads useless.
- Choosing 256 blocks with 256 threads each. Each block is responsible for one key, instead of looping over all keys in every thread.

The second option speeds up the calculation at the cost of reading the histogram data 256 times instead of 1 time. Timings showed later that the second option improves the performance by a few seconds.

Since the memory usage of this kernel is very lightweight compared to the histogram kernel, every reasonable implementation can be expected to be usable and to take far less time than the histogram kernel. Unfortunately the GPU occupation is reduced to 50% again, this time not because of the shared memory usage, but because of the register usage, mainly caused by the need to work with double precision floating point numbers needing two registers for one value.

Double precision floating point numbers were introduced with Compute capability 1.3 and offer greater precision but need 64 bit. Shared memory accesses that are larger than 32-bit are split into two 32-bit accesses by CUDA devices, thus generating bank conflicts. CUDA provides functions to manually split up double to two integer values and back, which may improve the performance by avoiding bank conflicts. This has been implemented as an option that can be activated by a compiler flag.

Another minor design decision could be to output integers instead of doubles as results, thus halving the amount of transferred result data without significantly reducing the quality of the results. This was also implemented as an option.

Finally there are intrinsic device functions that implement a function faster but with less precision, such as the `__log2f()` that could be useful for the mutual information calculation. This can be used with our implementation by changing a compiler macro for logarithm calculation, but unfortunately proved to bring a small performance gain at the cost of a huge precision loss.

5.2 Memory bandwidth

As the memory bandwidth constitutes a major bottleneck for programming on a GPU, the data transfer rate is one of the most important performance factors.

Transfers between host and device For any power analysis attack performed on a GPU it is inevitable to transfer all input (plaintext) and output (oscilloscope) data to the GPU, unless one decides to offload the reduction part (in our case: generation of

histograms from the original data) to the CPU. As histogram generation holds by far the highest part of runtime (see chapter 4.4) in our CPU implementation, this is not an option.

The needed data is never read by the host, hence we can use write-combining and page-locked memory (see chapter 2.4.3), which allows the best performance one can get according to Nvidia's CUDA Programming guide [CUD09b].

As seen in Fig. 2.3, we would achieve the best performance (5739.7 MB/s) by copying all needed memory[2] in 40 MB chunks. Thus we can obtain a lower bound[3] on the memory transfer time of our application: $\frac{40960 \text{ MB}}{5739.7 \text{ MB/s}} = 7.14s$. For comparison, on a GeForce GTX 290 with a maximum of 2904 MB/s the transfer time would double to $\frac{40960 \text{ MB}}{2904.3 \text{ MB/s}} = 14.1s$.

Although these numbers only denote the lower bound for transfer times, we conclude that host to device transfers will not cause performance problems for our MIA implementation. Device to host transfers - to pinned but not write-combining memory, because write-combining memory cannot be efficiently read by the CPU - proved to be much slower (arround 1800 MB/s for chunk sizes less than 10 MB), but since only 40 MB need to be transferred and big chunk sizes can be used to avoid any needless overhead, this poses no problems on our implementation, too.

Global and shared memory access For each measurement point the following memory transfers are performed:

1. Leakage and histogram kernel
   a) Every byte of input/output data (2 MB) needs to be read from global memory once. It is not written to shared memory but directly read to registers in aligned 32-bit words.
   b) For every byte of input/output data one histogram bin counter (32-bit word) is written at least 256 times in shared memory. It is not possible to avoid shared memory bank conflicts here, and it is possible that multiple threads need to write to the same bin at the same time. If two threads write concurrently to the same bin, only one thread succeeds, so the write needs to be repeated up to 32 times. Additionally each histogram is initialized to zero 256 times.

[2] 40 GB, see 3.1 for the reasoning of this number. Note that other transfer, e.g. 40 MB of results from device to host are negligible compared to 40 GB and hence not taken into account.

[3] It is a lower bound for several reasons. First of all, data transfers to several devices at the same time can reduce the data rate, depending on whether the device share the same bus. Running `bandwidthTest` on four different devices at the same time reduced the data rate by approximately 25% in our tests. Secondly, data is not transferred in 40 MB chunks, but needs to be split up to an unknown number of devices that operate on an unknown and potentially differing speed, hence needing an unknown and differing amount of workunits.

   c) 512 blocks need to merge 9 (HW) histograms 256 times (key byte) from shared to global memory by using CUDA's atomicAdd function, i.e. a total of $512 \times 256 \times 9 \times 1\text{kb} = 512 \times 2.25 = 1152$ MB. Shared memory access is free of bank conflicts and access to global memory is in aligned 32 bit words, but multiple blocks might write to the same position in global memory at the same time, hence there can be delays caused by atomicAdd.

2. Mutual information kernel
   a) 2.25 MB of histogram data is read from global memory to registers.
   b) 256 results (integer or double, hence 1kb or 2kb) are written to global memory.
   c) Small amounts of shared memory are written approximately 2×256 times with consideration of bank conflicts.

Based on this list the assumption to consider the memory transfers of the mutual information kernel negligible seems certainly justified compared to the requirements of the histogram and leakage kernel. This assumption was confirmed by our timing results later:

- 1a needs less than 1 ms (negligible)
- 1b needs less than 10 ms if concurrent writes are simply ignored, but approximately 350 ms otherwise.
- 1c needs 139 ms with atomicAdd. With a non-atomic addition the time is reduced to 55 ms.
- The histogram kernel without incrementing the histogram bin counters and writing the results to global memory needs less than 10 ms.

As seen from these numbers, the main problem here is either the small size of shared memory, allowing us to hold only a small number of histograms in shared memory, or the low data throughput for global memory, which allows holding a huge number of histograms, but is simply to slow.

Compute capability Shared memory atomic commands and double precision floating point numbers are two important features available only for devices with compute capability 1.3. Although neither of them is strictly indispensable for a MIA implementation, we decided to restrict our program to 1.3 devices for the sake of simplicity and to keep the precision of the results high.

5.3 Fast parallel summation

Calculating mutual information essentially means calculating a series of sums and logarithms. Although it is easy to implement a summation in parallel, there are several

points that need to be considered to obtain an efficient implementation in CUDA.

A parallel summation is a reduction algorithm that takes an array of size N as input and uses N/2 threads to compute the sum of all elements by letting each thread sum up a pair of two elements in each computation step, thus halving the number of remaining elements (and the number of active threads) with every step. Mark Harris, an Nvidia developer, discusses the problem of parallel reduction on CUDA devices in great detail in a presentation [Har09a], describing how to optimize reduction algorithms in several steps.

Algorithm 7 Naive approach: Parallel sum

```
for(unsigned int s=1; s < blockDim.x; s *= 2)
{
  int index = s * threadIdx.x * 2;
  if (index < blockDim.x)
  {
    data[index] += data[index + s];
  }
  __syncthreads();
}
// result is now in data[0]
```

- The naive approach (algorithm 7) produces bank conflicts that can be avoided by using sequential addressing as shown in Fig. 5.1.
- Further optimization is achieved by reducing the instruction overhead introduced by the loop and unnecessary flow control and synchronization. The loop can be unrolled for arbitrary loop counts with the help of C++ templates, but for `miakernel` this is not necessary because we always need to sum up exactly 256 elements. Synchronization is not needed as only 32 or less active threads are left, because all threads of a warp synchronize automatically. Note that this does not work in device emulation mode, because for emulation the warpsize is 1.

The resulting function is shown in algorithm 8. It does not need more operations than a sequential algorithm would need, hence it is work-efficient, and the time complexity is $O(logN)$.

Additionally another version was implemented that operates on two split integers instead of one double.

5.4 Implementation

Before going into the details of the kernel implementations, the basic application flow needs to be explained. As the usage of asynchronous functions cannot be separated from the main workunit cycle, both are described together in the following.

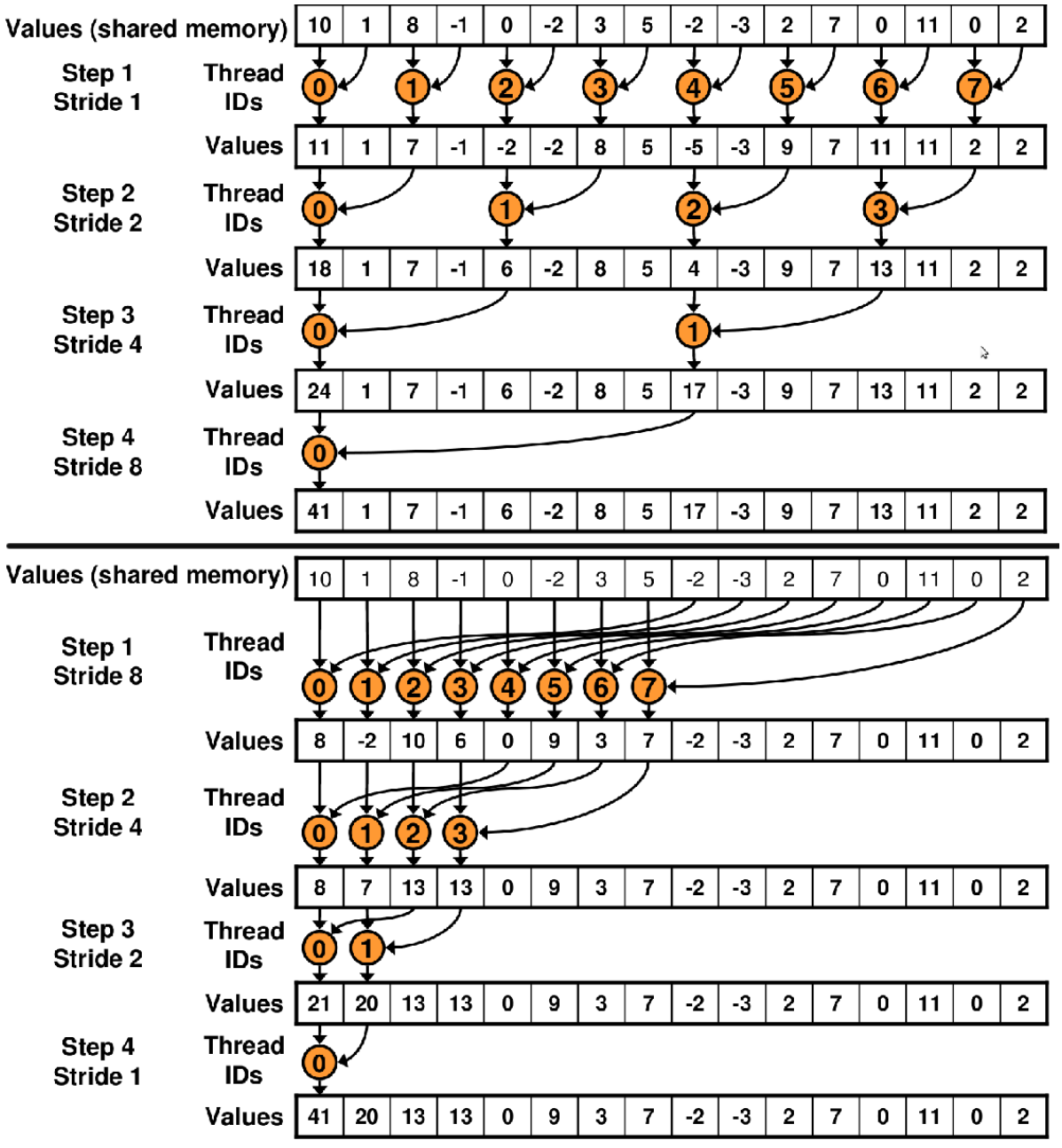

Figure 5.1: Naive and optimized parallel summation, taken from [Har09a]

Algorithm 8 Bank-conflict free parallel unrolled summation, adapted from [Har09a] for 256 elements

```
__inline__ __device__ void optimized_sum(int threadid, double *data)
{
    if (threadid < 128) { data[threadid] += data[threadid + 128]; }
        __syncthreads();
    if (threadid < 64) { data[threadid] += data[threadid + 64]; }
        __syncthreads();
    if (threadid < 32)
    {
        data[threadid] += data[threadid + 32];
        data[threadid] += data[threadid + 16];
        data[threadid] += data[threadid + 8];
        data[threadid] += data[threadid + 4];
        data[threadid] += data[threadid + 2];
        data[threadid] += data[threadid + 1];
    }
    __syncthreads();
}
// result is now in data[0]\textbf{}
```

5.4.1 Asynchronous application flow

In section 2.4.4 we explained the usage of asynchronous memory transfers and kernel calls with the help of streams. These represent the reason for the breakdown of workunits into subworkunit. Before proceeding to the sequence of asynchronous calls, we present some details on subworkunits and the resulting offset calculations.

Subworkunits To allow the usage of asynchronous functions, subworkunits were introduced. This construct is necessary for two reasons:

- The measurement points are distributed across an unknown number of devices that operate on an unknown and potentially differing speed, hence needing an unknown and differing amount of workunits. Each device asks for a new workunit only after finishing the previous one, hence it is not possible to start copying the data needed by the next workunit. Subworkunits are used to circumvent this problem by always giving a device several measurement points (POINTS_PER_WORKUNIT) at the same time, so it knows for which points it can pre-load data.
- It is not possible to allocate memory for all measurement points at the same time. Since processing points asynchronous requires each concurrently processed point to have a separate memory area, a limit on the number of concurrently asynchronously processed points is required. Assigning POINTS_PER_WORKUNIT measurement points to one workunit serves this purpose.

Algorithm 9 outlines the usage of subworkunits. The relationship of the variables `workunit`, `subworkunit` and `s` can be exemplified as shown in listing 4 for POINTS_PER_WORKUNIT=3. The last workunit can have less subworkunits.

Listing 4 Example for the relation of workunit, subworkunit and s

```
workunit:     [     123     ][     126     ]...[    20000    ]
subworkunit: [123,124,125][126,127,128]...[19998,19999]
s:            [0   ,1   ,2  ][0   ,1   ,2  ]...[0     ,1     ]
```

Offsets Memory for one logical unit such as all histograms or all results is always allocated in one block, improving performance as well as brevity. This requires the use of offsets for memory transfers and kernel calls. For example, the offset for the histograms of a subworkunit is calculated as follows: `dev_Histogram` is the start of the histogram memory block. To access the 9 histograms for subworkunit `s`, we need to add `s * KEYS * HW * BINS` to the start address, thus resulting in `unsigned int * dev_Histogram_subworkunit = dev_Histogram + s * KEYS * HW * BINS;`

Offset calculations are left out for the sake of brevity in the examples in this thesis, but [xyz]_subworkunit always refers to the variable xyz with the corresponding offset for the subworkunit s.

Asynchronous processing Making effective use of asynchronous functions proved to be difficult here. We analyze the potential for asynchronous execution step by step.

1. Initialize histograms to zero by asynchronously copying zeros from host. Setting the histogram to zero on the device seems more intuitive, but an extra kernel is needed for this task and proved to be slower. The CUDA-memset function is not available as asynchronous function.
2. While the copy of zeros is still working, input/output data is loaded from the host hard disc.
3. Copy input/output data to the device in the same stream as the previous copy. It would be possible to use another stream here as there is no dependency between the first and second copy, but this proved to be slower due to the overhead, because usually the first copy is already finished before this step begins.
4. Execute histogram kernel; the call returns before the device finished it.
5. Execute mutual information kernel. As this kernel relies on the previous, processing starts only after finishing the previous step. Kernels in different streams can run concurrently on some devices with compute capability 2.0.
6. Wait for all streams to finish before starting the next workunit, as explained above.

Step 2 profits most from the use of asynchronous functions, because it can start loading from a file on the host while copying memory to a device, but this would be possible without the subworkunit construct and even without asynchronous functions by using another host thread for loading the file. The only step that actually profits from the subworkunit construct is step 4, because it can start histogram kernels while it stills copies data to the device for another stream. Step 5 only gains performance on devices with compute capability 2.0, which were not available for our tests.

All in all, a significant gain can be expected from step 2 and minor improvements from the other steps. Algorithm 9 shows the code for the discussed steps.

5.4.2 Histogram kernel

Based on the design proposed in section 5.1.1 the histogram kernel works in the following steps:

- Allocate shared memory for 9 histograms (one for each Hamming weight value).
- Load 32 bit input and 32 bit output data from global memory to registers.
- Initialize histogram in shared memory to zero.
- Loop over all key values (0-255):
  - Split input / output data to byte values, calculate the leakage and increment the corresponding histogram bin using atomicAdd in shared memory.
  - Merge the shared memory histograms of each block to global memory.
  - Set shared memory histogram to zero for the next key.

The signature for the histogram kernel is `__global__ void miakernel_histogram (unsigned int * gIOdata, unsigned int * gHist)`. The `gIodata` parameter for input / output data is declared as `unsigned int *` to allow efficient 32-bit access. `gHist` is the result memory area that holds 9 histograms for each key.

Each block can calculate the position of the data it has to process by using a thread-id that spans across all blocks instead of only inside of one block: `int xtid = BLOCKNUMBER * blockIdx.x + threadIdx.x`

It is used to pre-load global memory data to variables in registers:

```
unsigned int iin = gIOdata[xtid];
```

For the output data we need `TRACENUMBER/4` as offset, because `TRACENUMBER` refers to 1 bytes words, but `gIOdata` is accessed in 4 byte words here:

```
unsigned int iout = gIOdata[(TRACENUMBER/sizeof(int)) + xtid];
```

To initialize the histograms to zero as well as merging the shared memory histograms to global memory, we access them sequentially to achieve the best performance, ignoring the

Algorithm 9 Asynchronous application flow

```
cudaStream_t stream[MIA_POINTS_PER_WORKUNIT];
for(unsigned int i=0;i<MIA_POINTS_PER_WORKUNIT;i++)
 { cutilSafeCall( cudaStreamCreate(&(stream[i])) ); }
while((workunit=work->get())>=0)
{
  unsigned int endworkunit=workunit+POINTS_PER_WORKUNIT;
  if(endworkunit>POINTS){endworkunit = POINTS;}

  // for every subworkunit in this workunit, ...
  for(long int s=0, subworkunit=workunit;subworkunit<endworkunit;
      subworkunit++,s++)
  {
    // Initialize dev_histogram to zero by copying host_Histogram (which
        was initially set to zero)
    cutilSafeCall(cudaMemcpyAsync(dev_Histogram_subworkunit,
        host_Histogram, Histogram_subworkunit_bytes, cudaMemcpyHostToDevice
        , stream[s]));

    // Load input/output data for subworkunit from file
    loadOscilloscopeDataFromTracefile(host_osc_subworkunit, 2 *
       TRACENUMBER, host_oscdata_fileoffset, fname);

    // Copy host input/output data to device.
    cutilSafeCall(cudaMemcpyAsync(dev_osc_subworkunit,
        host_osc_subworkunit, Oscdata_subworkunit_bytes,
        cudaMemcpyHostToDevice, stream[s]));
  }

  for(long int s=0, subworkunit=workunit;subworkunit<endworkunit;
      subworkunit++,s++)
  {
    // Execute histogram kernel.
    miakernel_histogram<<< gridDim, MIA_BLOCK_SIZE_HISTOGRAM_KERNEL, 0,
        stream[s]>>>( (unsigned int*) dev_osc_subworkunit,
        dev_histogram_subworkunit);

    // Execute MIA kernel
    miakernel_mia<<< KEYS, BINS, 0, stream[s]>>>(dev_histogram_subworkunit
        , dev_mia_subworkunit);
  }

  cutilSafeCall(cudaThreadSynchronize());
}

for(unsigned int i=0;i<POINTS_PER_WORKUNIT;i++)
  { cutilSafeCall(cudaStreamDestroy(stream[i])); }
// No more workunits, device finished. Now copy all results in one big
    chunk (very fast).
```

usual access pattern as multidimensional array of the form `sharedmem_Histogram[hw][bin]`. Using a loop and incrementing the index by the number of threads of a block (`BLOCKSIZE`) we utilize the full number of threads efficiently and guarantee a bank-conflict free aligned access.

The main loop of the histogram kernel is presented in algorithm 10.

Algorithm 10 Main part of histogram kernel (with shared memory atomicAdd, minor details stripped for brevity)

```
for(int key=0;key<KEYS;key++)
{
    __syncthreads();
    for(int byte=0;byte<4;byte++)
    {
        // Split iodata to 1 byte
        unsigned char shift =byte*8;
        unsigned char  in   =( iin  >> shift ) & 0xFFU;
        unsigned char out   =( iout >> shift ) & 0xFFU;

        // Hypothetical leakage for key guess according to Hamming Weight
           SBOX
        unsigned char leak = d_AES_SBox_Hamming[ in ^ key ];

        // Increment histogram bin counter in shared memory
        atomicAdd( &sharedmem_Histogram[leak*BINS+out], 1);
    }
    __syncthreads();

    // Merge histogram of this block to histogram in global memory
    i=tid;
    while(1)
    {
        // Global memory holds histogram for each key - choose correct
           offset
        unsigned int gHistOffset = key * HW * BINS;
        if(i<HW*BINS)
        {
            atomicAdd( &gHist[gHistOffset + i], sharedmem_Histogram[i]);
            sharedmem_Histogram[i]=0; // re-init histogram in shared
               memory to 0
        }
        else break;
        i+=BLOCKSIZE;
    }
}
```

Unfortunately this approach did not work stably in our tests, due to frequent infinite hangs in the atomicAdd function on shared memory. It was not possible to determine the reasons without doubts. Replacing the atomicAdd with a non-atomic addition works

without problems, but every time multiple threads write to the same bin at the same time only one write succeeds, thus loosing information and impairing the results. Working directly in global memory, using global memory atomicAdd, works without problems, too, but is prohibitively slow. Since shared memory atomic functions are relatively new to CUDA we cannot exclude with certainty that there is a bug in the device function.

Workaround

Based on the previous solution we implemented a workaround, which will slow down the computation but makes it reliable. The main idea is to partition the histograms so that each warp is responsible for one part of each histogram. Consequently, only intra-warp-conflicts need to be solved, which is done by introducing *tags* for write access, adapted from the histogram256[4] solution provided with CUDA code samples.

Assigning histogram parts to warps Since each thread holds unique input / output data and it seems desirable to avoid redundant reads of data from global memory, the assignment of warps to histogram parts must be cycled, so each warp is working on each histogram part at one point.

The warpsize on today's devices is $32 = 2^5$, hence we compute the warp-id from the thread-id as `warp = tid >> 5`. The number of warps is `WARPS_PER_BLOCK`$= \frac{BLOCKSIZE}{WARPSIZE} = \frac{512}{32} = 16$. With this approach, each data item is processed 16 times instead of 1 time, thus seriously downgrading the performance. On the other hand, the number of collisions is greatly reduced. Although access is not uniformly distributed, the maximum number of 32 collisions per write is seldom reached, because the access of 32 threads is spread over 9×32 bins.

Algorithm 11 shows the cycle of histogram part distribution.

Tag and check write access If several threads of one warp write to the same location in shared memory, only two threads actually perform a write (in unknown order). For this reason we use the five most significant bits (MSB) of each bin counter to store a thread-id-dependent tag that allows to determine whether a thread wrote successfully. With our usual number of 1048576 traces this can never cause a problem, because the 5 MSBs would remain unused even if they all belong to the same bin.

A tag is simply the thread-id shifted to the 5 MSBs, and is applied to the bin counter as bitwise OR. The atomic addition is realized with an read - write - check sequence that is performed up to 32 times in a loop, as demonstrated in algorithm 12. `HIST_TAGMASK`

[4]histogram256 was not directly usable since it works only on one histogram. For MIA the histogram kernel works on several (in our case 9) histograms that additionally have to perform the leakage calculation.

Algorithm 11 Histogram kernel workarround: divide histogram parts

```
for(int key=0;key<KEYS;key++)
{
    for(int p=0;p<WARPS_PER_BLOCK;p++)
    {
        int hist_partStart = ( (warp +  p   )*WARPSIZE ) % BINS;
        int hist_partEnd   = ( (warp + (p+1))*WARPSIZE ) % BINS;

        for(int byte=0;byte<4;byte++)
        {
            unsigned char out   =( iout >> shift ) & 0xFFU;
            // is 'out' in the part of the histogram that this thread is
                working on?
            if(out >= hist_partStart && out < hist_partEnd)
            {
                // calculate leakage, increment in histogram
            }
            // else do nothing (recalculated later)
        }
    }
}
```

is a 32 bit mask with all bits apart from the 5 MSBs set to 1. `sharedmem_Histogram` should be declared `volatile` to avoid undesired compiler optimizations to the read - write - check sequence.

5.4.3 Mutual information kernel

The mutual information kernel has been implemented with a compiler flag to replace all occurrences of shared memory double reads and writes with their split 32-bit equivalent; see 5 for an example. For the sake of readability (and because the performance improvement was insignificant in our tests) we concentrate on the standard double version here.

Another compiler flag is used to decide whether results should be transferred as integers (to save bandwidth) or double values. Thus the mutual information kernel has the signature `__global__ void miakernel_mia(unsigned int * gHist, RESULT * gResult)` and uses an `#ifdef` clause for writing the results to global memory in the correct format. Internally it always uses double values, so the precision loss is marginal and the flag is irrelevant for an explanation of the code.

As mentioned in 5.1.2 there is a pre-compiler macro `_LOG2(x)` that allows to change the used logarithm function easily, mainly to change between the precise standard logarithm and the less precise but fast intrinsic device function.

Algorithm 12 Histogram kernel workarround: Tagged writes

```
// Loop through all keys (0-255)
for(int key=0;key<KEYS;key++)
{
  for(int p=0;p<WARPS_PER_BLOCK;p++)
  { ...
    for(int byte=0;byte<4;byte++)
    { ...
      if(out >= hist_partStart && out < hist_partEnd)
      {
        unsigned char leak = d_AES_SBox_Hamming[ in ^ key ];
        int index=leak*BINS+out;

        // shift threadid to 5 MSBs of 32bit word => blockwide-unique tag
        unsigned int tag = tid << (32-5);
        unsigned int value;

        // Counter prevents eternal hangs, e.g. caused by corrupt memory
        unsigned int sanity_counter=0;
        do {
            sanity_counter++;
            // Save old histogram value (without bits used for tag)
            value = sharedmem_Histogram[index] & HIST_TAGMASK;
            // Increment and save tag to 5 MSBs of value
            value = (value + 1 ) | tag;
            // Write value to shared memory.
            sharedmem_Histogram[index] = value;
        }
        // check whether write succeeded or maximum number is reached
        while(sanity_counter < 32 && sharedmem_Histogram[index] != value);
      }
    }
  }
  // Merge histogram of this block to histogram in global memory as before
      . Only difference: strip tag from value before adding to global
     memory
  while(...)
  {
    value=sharedmem_Histogram[i] & HIST_TAGMASK;
    // global memory atomic add works fine
    atomicAdd( &gHist[gHistOffset + i], value);
  }
}
```

Listing 5 Avoiding bank conflicts: accessing double values in 32-bit words

```
double temp = __hiloint2double(shared_hi[tid], shared_lo[tid]);
shared_lo[tid]=__double2loint(temp);
shared_hi[tid]=__double2hiint(temp);
```

Recall that each block calculates the mutual information of one key (`int key=blockIdx.x`) and each thread of a block is responsible for one bin in all histograms, but also for general purpose (`int tid = threadIdx.x`). The basic histogram offset is calculated as `histogramOffset = key * HW * BINS`.

Two shared memory areas are declared: `__shared__ double s_temp[BINS]` mainly for fast parallel reduction as explained in section 5.3 and `__shared__ double s_hwtemp[HW]` as temporary variable for intermediate results.

S(hw)=`HW_DISTRIBUTION[hw]` refers to the pre-calculated Hamming weight distribution as described in algorithm 5.

Algorithm 13 shows the first loop of the mutual information kernel. Definition 5.4.1 shows the equation from chapter 2.3.1 adapted to the kernel implementation and illustrates the progress of the algorithm after the first loop. Each thread reads its corresponding $O_{hw}(tid)$ value from the histogram in global memory; this happens exactly once for each value, so there is no need to read it to shared memory first. Note how the line marked with [*] already calculates a part of the subtrahend during the calculation of the minuend, i.e. tempB holds $\sum_{hw} O_{hw}(tid)S(hw)$.

Algorithm 13 Mutual information kernel: first part

```
for(int hw=0;hw<HW;hw++)
{
  // read histogram value
  double temp=global_Histogram[histogramOffset + hw * BINS + tid];

  // Sum up O_hw(tid) for all hw=0...8 [*]
  tempB+=temp * HW_DISTRIBUTION[hw];

  // O_hw(tid) * log( O_hw(tid) )
  if(temp>0)temp=temp * _LOG2(temp);
  s_temp[tid]=temp;

  // Fast parallel reduction (syncthreads included in optimized_sum)
  optimized_sum(KEYS,tid,s_temp); // \sum_{tid} of minuend

  // s_temp[0] holds the sum of all s_temp elements
  if(tid==0){ s_hwtemp[hw]=HW_DISTRIBUTION[hw] * s_temp[tid]; }
}
```

Definition 5.4.1

$$I(S,O) = \sum_{hw}\sum_{tid} O_{hw}(tid)S(hw)\ log_2(O_{hw}(tid)) - \sum_{tid}(\sum_{hw} O_{hw}(tid)S(hw))\ log_2(\sum_{hw} O_{hw}(tid)S(hw))$$

The second part of the mutual information kernel calculates all remaining parts of the equation, using the parallel reduction again to compute the outer sum of the subtrahend. Finally, only one thread calculates the outer sum of the minuend, thus finishing the whole calculation, and writes it to global memory as integer or double value.

Algorithm 14 Mutual information kernel: second part

```
// calculate inner part of subtrahend
if(tempB>0)tempB=tempB * _LOG2(tempB);
s_temp[tid]=tempB;

// outer sum of subtrahend
optimized_sum(KEYS,tid,s_temp);

if(tid==0)
{
  /// s_hwtemp[hw] holds inner sum of minuend
  tempB=0;
  for(unsigned char hw=0;hw<HW;hw++)
  {
      tempB+=s_hwtemp[hw];
  }

  /// s_temp[0] holds outer sum of subtrahend
  /// Write to global result memory
  #ifdef RESULT_INT
  *(gResult+key)=__double2int_rn(tempB-s_temp[tid]);
  #else
  *(gResult+key)=tempB-s_temp[tid];
  #endif
}
```

6 Results

There are mainly two criteria to analyze the success of our implementation: The accuracy of the mutual information results and the efficiency of the computation compared to the CPU implementation. After the presentation of our results we make an attempt at interpreting the results and illuminating the reasons of our findings in section 6.3.

6.1 Precision

Although the steps to compute mutual information on GPU and on the host CPU are mathematically equivalent they yield different results due to accuracy problems with floating point numbers. For example the GPU implementation moves a floating point constant into a summation for performance reasons, hence the imprecisely binary represented factor is multiplied with each summation element instead of once with the summation result, thus giving an significantly different result. We found discrepancies between CPU and GPU computations of up to 30%. However, the difference between maximum and minimum value of the GPU and the CPU results was very close, so the results do not seem to loose much information, but mainly loose their direct comparability.

Consequently results of mixed GPU/CPU computations must be scaled before comparison, which can be accomplished by finding the minimum and maximum for all GPU (respectively CPU) results, computing the result range and use the same factor to scale all GPU (respectively CPU) results to the desired range. However, this can induce further inaccuracy.

Applying a different mutual information kernel to the histograms generated by the usual histogram kernel proved that the lost accuracy is actually caused by the different calculation steps. Our test kernel used only one thread and exactly the same calculation steps as the CPU implementation, accordingly it was very slow but produced the same results as the CPU.

Even more inaccuracy was caused by using the intrinsic device function `__log2f()` for base-2 logarithm, without bringing a significant performance gain. `__log2f()` operates on single precision floating point numbers, which first of all explains the low precision and secondly might explain the performance, since the conversion of double to single precision and back causes an additional overhead.

6.2 Timings

This section presents timings of our implementation, which were acquired from the timer values output with the debugging information unless otherwise noted. All timings are average values based on multiple executions.

To give a feel for the magnitude of the MIA runtime the following list shows a few approximate values for 20000 measurement points with 1048576 traces, before the next paragraphs go more into the detail.

- CPU implementation:
  - 1 thread: 199 minutes
  - 8 threads (8 CPU cores): 26 minutes
  - 23 threads (8 CPU cores): 23 minutes (best result)
  - Histogram generation for 256 keys: 637 ms to 1200 ms (82.9% to 98.7% of the time needed for one measurement point, see Listing 1 - 3)
  - Mutual information for 256 keys: 5 ms to 110 ms
- GPU implementation:
  - 1 GPU: 48 minutes
  - 2 GPUs - 4 GPUs: 42 minutes
  - Histogram generation for 256 keys: 479 ms (97.4% of the time needed for one measurement point)
  - Mutual information for 256 keys: 17 ms

6.2.1 Number of measurement points

As each measurement point can be processed independently one can expect that increasing the number of measurement points increases the total runtime by the same factor. If the number of points is high enough (e.g. greater than 200) the overhead should be irrelevant. This assumption was experimentally confirmed as seen in Figure 6.1 and was helpful to reduce the time needed to generate timing statistics. The relation is valid for CPU-only, GPU-only and all mixed executions.

6.2.2 Number of traces

Since the number of traces was initially specified as 1048576, the implementation was not build for much higher numbers, thus limiting the explorable range for tracenumbers by causing integer overflows. Nevertheless the results of this measurement allow a

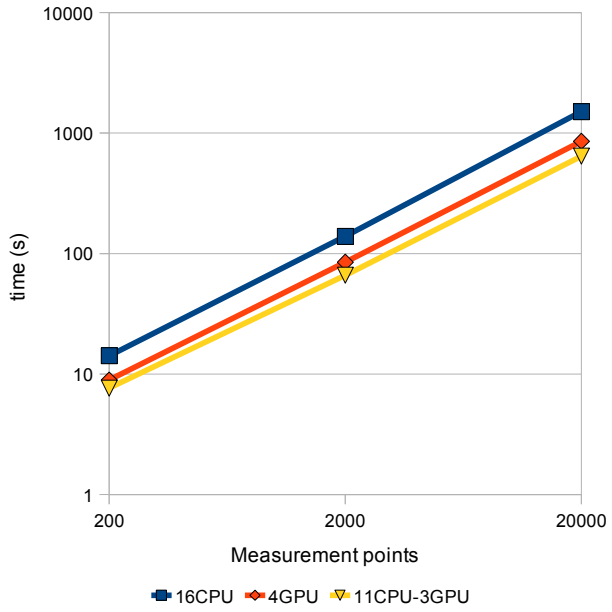

Figure 6.1: Increasing the number of measurement points increases the runtime by the same factor.

good estimation of results for higher numbers, which can be reached by modifying the implementation to use long integers.

Increasing the tracenumber for the CPU implementation causes a linear increase in the runtime by roughly the same factor. Interestingly the GPU implementation shows a smaller factor for tracenumbers higher than 1048576, indicating that the GPU implementation is especially suited for attacks employing a high number of traces per measurement point. The difference is based on the reduction of overhead in the memory management and a greater effectiveness of parallelization caused by the usage of larger workunits. See Figure 6.2 and Table 6.2.2 for a comparative measurement with 200 points. The factor shown in the table show the ratio of each runtime compared to the respective runtime for 1048576 traces. `CPU=x, GPU=y` denotes the usage of `x` CPU-threads and `y` GPU devices.

Traces	Exp.	Time (s) CPU=1	Factor	Time (s) GPU=1	Factor	Time (s) GPU=4 CPU=38	Factor
6291456	6×2^{20}	736.9	5.83	162.3	5.34	58.7	5.34
4194304	4×2^{20}	505.1	4.00	107.6	3.54	39.5	3.6
2097152	2×2^{20}	252.1	1.99	55.9	1.84	20.8	1.9
1048576	1×2^{20}	126.4	1.00	30.4	1.00	11.0	1
524288	0.5×2^{20}	63.8	0.51	14.0	0.46	5.9	0.54

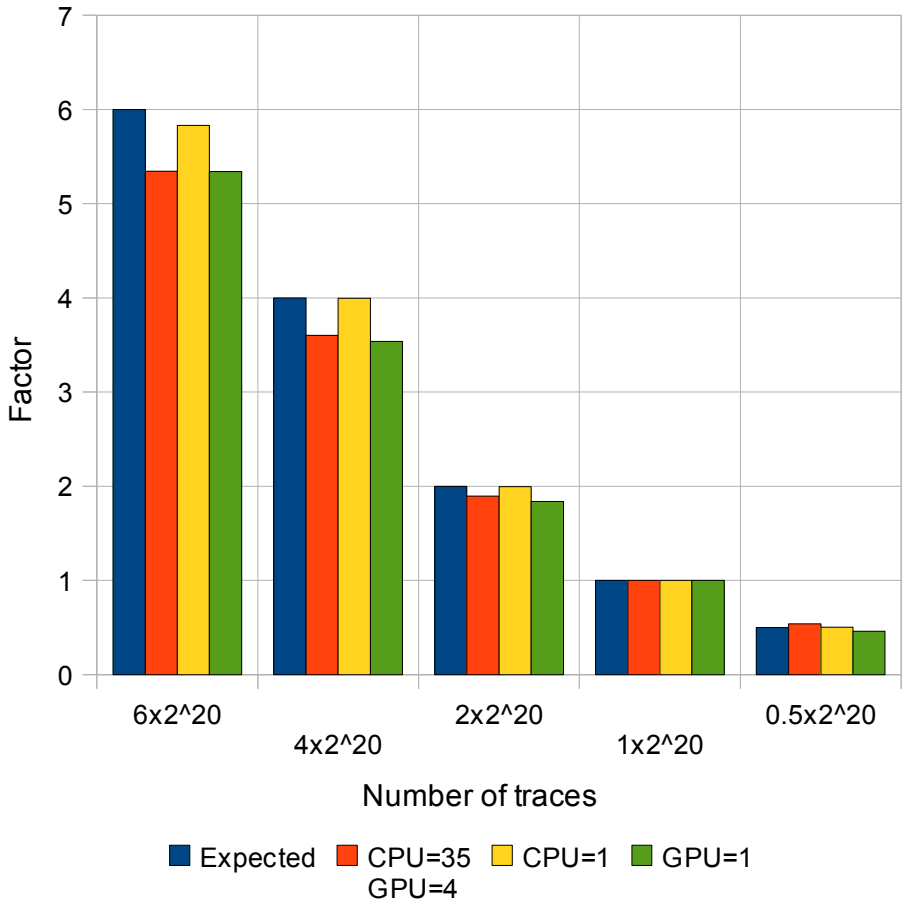

Figure 6.2: Increasing the number of traces increases the runtime by approximately the same factor. However the factor is lower for GPUs, thus increasing the runtime difference between CPU and GPU implementation in favor of the GPU. The higher the number of traces the higher the performance gain achieved from the GPU-assisted implementation.

6.2.3 Runtime of mixed GPU / CPU executions

Based on a measurement with 2000 points, Figure 6.3 shows a comparison of runtimes for mixed executions of CPU and GPU implementation. Several important conclusion can be drawn from this figure:

- The GPU implementation was successfully designed to effectively using the stream processing architecture.
  - While 1 CPU-thread (i.e. 8 CPU core) needs 1193.86 seconds for MIA with 2000 points, 1 GPU (i.e. one Nvidia GeForce GTX 295 with 2×240 cores, as detailed in 3.1) needs only 290.56 seconds, thus improving the speed by a factor of 4.11.
  - Combining the computational power of all 8 CPU cores using 38 CPU-threads with 4 GPUs providing a total of 1920 cores yielded the best result in our measurements: 98.74 seconds, giving an performance boost of factor 12.09.
  - Nearly the same factor can be reach already with 21 CPU-threads and 2 GPUs, scoring with 101.34 seconds.
  - These results allow an accurate estimation for measurements with 20000 points just by multiplying the runtime with 10.
- Using two GPUs (113.29 seconds with 30 CPU-threads) yields a serious improvement to using 1 GPU (138.80 seconds with 23 CPU-threads), but increasing the number of devices further only reduces the variability of the timing without lowering the runtime significantly. The minimal time of approximately 98 seconds is reached by two GPUs (with 34 CPU-threads and 98.95 seconds) and three GPUs (39 CPU-threads and 98.90 seconds) as well as four CPUs (38 CPU-threads and 98.74 seconds).

6.2.4 Other parameters

This section summarizes the results of measurements that showed no significant impact on the runtime.

Asynchronous execution: number of points per workunit

The implementation chapter describes a design for improving the efficiency of using asynchronous function calls for memory transfers and kernel calls (see section 5.4.1). At the same time it already suggested that no significant gain could be expected from this technique because the margin for improvements was very limited. This was confirmed by our experiments, showing that the overhead of the used loops outweighs any improvement or actually downgrades the performance by a few milliseconds. All operations (e.g.

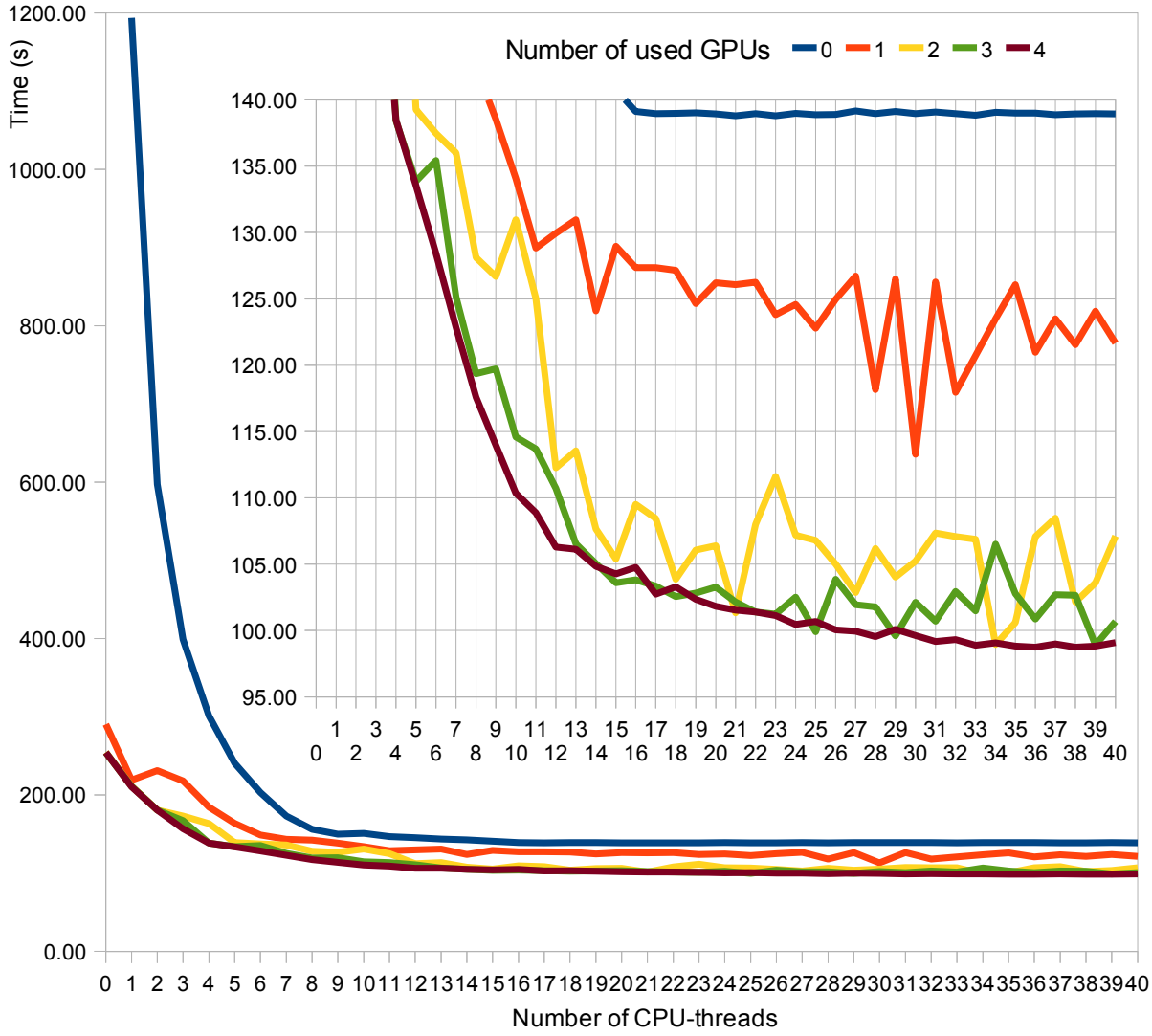

Figure 6.3: Runtime for 2000 points. Both charts show the same data, differing only in the scaling of the y-axis.

loading data from file and copying it to the device) that could profit from the design are performed too fast to make a difference.

Result type: double or integer

The GPU implementation allows transferring the final mutual information results as integer values instead double precision floating point numbers from the device's global memory to host main memory, hence halving the amount of data. While this actually saved a few seconds in an older version of the implementation that copied the results in small chunks directly after a workunit had finished, the improvement is largely irrelevant now, because even copying the double precision results of 20000 measurement points takes just a few milliseconds. Moreover the effect is getting even less for multiple devices in our test environment, probably because the devices shared the same memory bus.

Further tests

- Disabling the output of timing information does not significantly improve the performance. (Few milliseconds, measured with the linux `time` function.)
- Compiling for 64-bit architecture instead of 32-bit caused a slowdown of a approximately one second for 20000 points (negligible).
- Using more intrinsic device functions such as `umul24` has only been briefly tested without achieving a performance gain. This was to be expected considering the already low arithmetic complexity. However it might be possible to speed up the mutual information kernel if a greater effort is put into a full optimization of the usage of intrinsic functions.
- Splitting up each shared memory access of double precision numbers to avoid bank conflicts caused no difference in timing at all.

6.3 Interpretation

While a performance boost by a factor of 4 to 12 definitely is an improvement, the question arises under which conditions a further improvement would be possible.

To answer this question the metric of *arithmetic intensity* can be used. It is defined as the ratio of operations per data word. If the arithmetic intensity is high, the performance has an upper bound determined by the possible number of operations per second, e.g. measured in Flops (Floating Point Operations Per Second). If it is low, the limiting factor is the memory bandwidth, because the full number of operations per second can

only be exploited if the bandwidth is high enough to provide data as quickly as it is processed.

As the histogram kernel takes up most of the time of one workunit, it can be considered the key issue for our MIA implementation. Histogram generation applies only a few (computationally cheap) operations to each data item: addition, bitwise operations such as XOR and shifts, and some multiplications for address arithmetic. On the other hand it needs one read from global memory, at least one write to shared memory (shared memory atomicAdd or workarround), one atomicAdd to global memory and additional writes for re-initializing histograms back to zero. Obviously the histogram kernel has a very low arithmetic intensity and is limited by the memory bandwidth.

The mutual information kernel has a higher arithmetic density, among other things due to the use of the logarithm, but this is relatively irrelevant considering the fraction of the processing time of one workunit it needs.

Consequently a significantly better performance can not be expected as long as histogram generation is used to sample the observed leakage.

7 Conclusion

In this chapter we summarize what we have done and give an outlook on what could be addressed by later works.

7.1 Summary

In this thesis we presented an approach for implementing a fast parallel version of Mutual Information Analysis for Graphics Processing Units. It was designed to profit from both the CPU power and all available GPU devices. After giving an overview on power analysis of AES devices, the information theoretic reasoning of MIA and the basics of CUDA programming in chapter 2, we explained the groundwork of both the CPU and GPU implementation in chapter 3. Chapter 4 describes a straight-forward C-implementation of MIA for CPU that mainly serves the purpose to allow later comparison but can also be used in combination with the CUDA-implementation presented in the chapter 5 to speed up the total runtime. Histogram generation proved to be the biggest challenge in terms of shared memory requirements and performance. Chapter 6 presents the performance gain achieved by using a varied number of GPUs in addition to or instead of the stand-alone CPU-threads and analyzes the influence of different parameters.

Comparing one CPU-core to one GPU, performance is improved by a factor of 4.11. Best performance was achieved by using a combination of all 8 CPU-cores running 38 threads and 4 GPUs. In this case processing 20000 measurement points with 1048576 traces takes 17 minutes instead of 3 hours and 19 minutes, which is 12 times faster than one CPU-core. This factor increases for measurements with a higher number of traces: 20000 points with 6291456 traces takes more than 20 hours and 38 minutes on one CPU-core but only 4 hours 16 minutes on one GPU (factor 4.83). Using two GPUs reduces the time to less than 4 hours and the combined computational power of 4 GPUs and 8 CPU-cores processes the same data in 96 minutes (factor 12.88).

More than 4 GPUs were not available for our tests, but as using more than two GPUs already provided no major performance benefit, additional devices can only be expected to enhance the performance if a different set-up is applied, e.g connecting them to different memory buses.

Although our implementation was only tested against simulated traces, the results suggest that it is ready for real-world usage and speeds up the MIA significantly. Fur-

thermore it allows the assumption that GPU-assisted implementations are especially well applicable in scenarios having higher computational demands. For attacks requiring more computations per data record an even higher performance gain can be expected.

7.2 Future Work

Since this thesis was meant to be a first evaluation of the usability of GPUs for side channel attacks in general and Mutual Information Analysis in particular, the positive results may very well be a motivation for future work in similar fields. GPUs would be even more effective in cases that have higher computational demands than MIA has.

There are also opportunities to continue and improve our MIA implementation. Finding an efficient **and** accurate implementation of mutual information on GPU might be subject for further investigations, as well as finding a more efficient solution for histogram generation, maybe by determining a way to use the shared memory atomicAdd function.

Finally the expansion to a different leakage model with a higher number of different values could be tried. When sticking to the Hamming weight model, the outsourcing of less frequently used parts of histograms to global memory with the goal to increase the GPU occupation by using less shared memory promises to be rewarding, and possibly usable for other models as well.

The evaluation of openCL and the effort necessary for converting applications from CUDA to openCL could be another task for future works, which would allow to address a wider range of GPUs.

Bibliography

[AES01] Specification for the advanced encryption standard (AES). Federal Information Processing Standards Publication 197, 2001.

[BMK04] Johannes Blömer, Jorge Guajardo Merchan, and Volker Krümmel. Provably secure masking of AES. In *In SAC*, pages 69–83. Springer, 2004.

[CRR03] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template attacks. In *CHES '02: Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, pages 13–28, London, UK, 2003. Springer-Verlag.

[CUD09a] Nvidia cuda c programming. best practice guide, 2009.

[CUD09b] Nvidia cuda. programming guide, 2009.

[FFS10] Jean-Luc Danger Moulay Aziz Elaabid Houssem Maghrebi Florent Flament, Sylvain Guilley and Laurent Sauvage. About probability density function estimation for side channel analysis. In Sorin Huss Werner Schindler, editor, *COSADE 2010 Workshop Proceedings*, 2010.

[GBTP08] Benedikt Gierlichs, Lejla Batina, Pim Tuyls, and Bart Preneel. Mutual information analysis. In *CHES '08: Proceedings of the 10th international workshop on Cryptographic Hardware and Embedded Systems*, pages 426–442, Berlin, Heidelberg, 2008. Springer-Verlag.

[GT03] Jovan Dj. Golic and Christophe Tymen. Multiplicative masking and power analysis of aes. In *CHES '02: Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, pages 198–212, London, UK, 2003. Springer-Verlag.

[Har09a] Mark Harris. Optimizing parallel reduction in CUDA, 2009. `http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf`.

[Har09b] Matt Harvey. Experiences porting from CUDA to OpenCL, 12 2009.

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. *Lecture Notes in Computer Science*, 1666:388–397, 1999.

[MDSM02] Thomas S. Messerges, Ezzat A. Dabbish, Robert H. Sloan, and Senior Member. Examining smart-card security under the threat of power analysis attacks. *IEEE Transactions on Computers*, 51:541–552, 2002.

[MMPS09] Amir Moradi, Nima Mousavi, Christof Paar, and Mahmoud Salmasizadeh. A comparative study of mutual information analysis under a gaussian assumption. pages 193–205, 2009.

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards (Advances in Information Security)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[MOS09] Stefan Mangard, Elisabeth Oswald, and Francois-Xavier Standaert. One for all - all for one: Unifying standard dpa attacks. Cryptology ePrint Archive, Report 2009/449, 2009. `http://eprint.iacr.org/`.

[OMPR05] Elisabeth Oswald, Stefan Mangard, Norbert Pramstaller, and Vincent Rijmen. A side-channel analysis resistant description of the AES s-box. In *FSE*, pages 413–423, 2005.

[PR09] Emmanuel Prouff and Matthieu Rivain. Theoretical and practical aspects of mutual information based side channel analysis. In *ACNS*, pages 499–518, 2009.

[Sha49] Claude E. Shannon. Communication theory of secrecy systems. *Bell Systems Technical Journal*, 28:656–715, 1949.

[SMY09] François-Xavier Standaert, Tal G. Malkin, and Moti Yung. A unified framework for the analysis of side-channel key recovery attacks. In *EUROCRYPT '09: Proceedings of the 28th Annual International Conference on Advances in Cryptology*, pages 443–461, Berlin, Heidelberg, 2009. Springer-Verlag.

[SVCO+10] Francois-Xavier Standaert, Nicolas Veyrat-Charvillon, Elisabeth Oswald, Benedikt Gierlichs, Marcel Medwed, Markus Kasper, and Stefan Mangard. The world is not enough: Another look on second-order dpa. Cryptology ePrint Archive, Report 2010/180, 2010.

[VCS09] Nicolas Veyrat-Charvillon and Francois-Xavier Standaert. Mutual information analysis: How, when and why? In Christophe Clavier and Kris Gaj, editors, *CHES*, volume 5747 of *Lecture Notes in Computer Science*, pages 429–443. Springer, 2009.