RUHR-UNIVERSITÄT BOCHUM

GPU Assisted Side-Channel-Evaluation

Thomas Klostermann

Master's Thesis. July 7, 2016.
Chair for Embedded Security – Prof. Dr.-Ing. Christof Paar
Advisor: Falk Schellenberg

Eidesstattliche Erklärung

Ich erkläre, dass ich keine Arbeit in gleicher oder ähnlicher Fassung bereits für eine andere Prüfung an der Ruhr-Universität Bochum oder einer anderen Hochschule eingereicht habe.

Ich versichere, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Die Stellen, die anderen Quellen dem Wortlaut oder dem Sinn nach entnommen sind, habe ich unter Angabe der Quellen kenntlich gemacht. Dies gilt sinngemäß auch für verwendete Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen.

Ich versichere auch, dass die von mir eingereichte schriftliche Version mit der digitalen Version übereinstimmt. Ich erkläre mich damit einverstanden, dass die digitale Version dieser Arbeit zwecks Plagiatsprüfung verwendet wird.

Statutory Declaration

Hereby I declare, that I have not submitted this thesis in this or similar form to any other examination at the Ruhr-Universität Bochum or any other Institution of High School.

I officially ensure, that this paper has been written solely on my own. I herewith officially ensure, that I have not used any other sources but those stated by me. Any and every parts of the text which constitute quotes in original wording or in its essence have been explicitly referred by me by using official marking and proper quotation. This is also valid for used drafts, pictures and similar formats.

I also officially ensure, that the printed version as submitted by me fully confirms with my digital version. I agree that the digital version will be used to subject the paper to plagiarism examination.

Not this English translation, but only the official version in German is legally binding.

Datum / Date

Unterschrift / Signature

Abstract

Correlation Power Analysis (CPA) as a passive power analysis based side-channel attack is a common technique to reveal the secret key of generally every encryption scheme in the context of embedded devices. To counteract this circumstance, a lot of effort has been expended by researchers on the development of countermeasures to either enlarge the computational effort or to decrease the probability of success for an attack. To ensure that a proposed countermeasure works properly, it needs to be verified by attacking the protection scheme, which in fact can become infeasible. It is of great interest for researchers and moreover for the industry to facilitate the examination of those countermeasures with regards to time consumption and costs. This thesis demonstrates the superior performance of Graphics Processing Unit (GPU) with respect to parallel computation power and specifically correlation power analysis by providing a GPU based framework to parallelize first and higher order CPA attacks. The framework's property to be scalable enables us to evenly distribute the work of the attack among an arbitrary large cluster of servers and moreover to distribute the work for each server among an arbitrary large number of GPUs. By combining this fact with the use of robust one-pass formulas, we are able to negate the limiting factor of bounded memory. This allows us to execute the attack on arbitrary large trace pulls at — for all practical purposes — any order.

Contents

1 Introduction

This chapter covers the motivation of the thesis in order to highlight the importance of the focused research subject. Additionally, a short overview of previous work done on this research field is provided. Thereby, we emphasize the problems and gaps in the current research state, which we want to address through our contribution. Moreover, the structure and organization of the thesis is described at the end of this chapter.

1.1 Motivation

Passive side-channel attacks are capable of breaking cryptographic schemes in a short period of time, regardless of the fact that the underlying cryptographic algorithm is considered to be cryptanalytically secure. This is based on the fact that implementation itself is being attacked rather than the mathematical structure of the algorithm. This problem has been addressed frequently by the scientific community over the last years. As a result, different countermeasures like masking, hiding, or trace misalignment have been proposed. Those countermeasures aim to reduce the effectiveness of side-channel attacks by either increasing the computational effort or by lowering the probability of success for an attack.
However, to ensure that a certain protection scheme is working properly, it has to be tested in a real world scenario by evaluating the computational effort, the success rate as well as the economical factor for an attack. If this evaluation process has to be done for a wide range of devices, it may become impractical, particularly in the case of higher order attacks, where several millions of traces have to be preprocessed. This preprocessing can be challenging since the trace pull has to be parsed at least two times. To overcome this problem, robust one-pass formulas [SMG15] allow the parallel computation of correlation based attacks at an arbitrary order by processing each trace only once.
Modern computer systems contain multiple Central Processing Unit (CPU) cores and thus, are suitable to perform parallel computations. However, with regards to floating point performance, a crucial factor in correlation based side-channel attacks, many-core GPU systems have led the race over multi-core CPU based systems since 2003. In 2009, the ratio for peak floating point calculation was approximately 10 to 1 in favour of GPUs and is increasing further[1]. In addition to that, GPU support for the Institute of Electrical and Electronics Engineers (IEEE) floating point standard as well as performance loss with regards to double precision execution speed has become comparable to that of CPUs. [KWm12]

[1] Comparing the Intel Core i7-3930 with the Tesla K80, the ratio is approximately 48:1 in favour of the Tesla K80. [i7] [K80]

The parallelization of correlation based side-channel attacks can be accomplished in multiple dimensions: For instance, in the dimension of the traces, the sample points per trace, or the key hypothesis, which predestine them to be conducted on systems with massive parallel computation capabilities as realized by GPUs. Since the introduction of the Compute Unified Device Architecture (CUDA) in 2007, GPUs are programmable in the same way as CPUs by adding a small set of extensions to the C / C++ programming language. Previously, GPU applications were developed in the context of General Purpose Computation on Graphics Processing Unit (GPGPU) by the use of an unnatural language like the Open Graphics Library (OpenGL), which had major drawbacks, see Section 2.1.1.

1.2 Related Work

Although they are not directly associated to CPA, several papers on the parallel computation of the Pearson correlation have been published. In 2009, a parallel algorithm to compute the Pearson correlation on GPU was proposed by Chang et al. [CDOR09]. We used this algorithm with minor modifications to adapt it to CPA. In 2011, Kijsipongse et al. [KNT+11] extended the work of Chang et al. to overcome its limitations. Their approach allows the computation of Pearson correlation for large datasets using a hybrid approach of Message Passing Interface (MPI) and CUDA on a cluster of multiple GPUs. Gembris et al. [GNG+11] investigated the performance of Pearson correlation for different architectures. They achieved a speed-up of 10 for the Field Programmable Gate Array (FPGA), respectively 15 for the GPU implementation compared to a single core Pentium 4 CPU.

The enormous parallel computation capabilities of GPUs with respect to side-channel attacks have been demonstrated by numerous publications. Bartkewitz et al. [BLR11] proposed a high performance implementation of Pearson correlation based Differential Power Analysis (DPA) obtaining a speed-up of approximately 100 compared to a common four core CPU. In 2014, the performance of CPA on a NVIDIA Tesla C2075 graphics card was analysed by Gamaarachchi et al. [GRJ14], resulting in a speed-up of approximately 66 compared to a 32 threaded high performance CPU server. Also in 2014, Swamy et al. [SSL+14] examined the performance of a serial first order CPA attack to a parallel one implemented in CUDA and Open Computing Language (OpenCL). For a dataset of 80000 traces and 1000 sample points per trace, the CUDA implementation on a combined system of CPU and a GeForce GTX 480 was approximately 6 times faster than the serial implementation on a Intel core I7-3820. Simultaneously, the OpenCL version on a AMD A10-6700 Accelerated Processing Unit (APU) in conjunction with a Radeon 8670 HD graphics card was approximately 15 times faster than the serial implementation.

Schneider et al. [SMG15] published formulas for robust and one-pass parallel computation of correlation-based attacks at arbitrary orders. Those formulas are a fundamental component of our work since they allow us to iteratively evaluate the correlation in a CPA attack at any order in both, univariate and multivariate settings with i) stable results, ii) each trace has to be processed only once while the result of the attack can be obtained at any time, and iii) the computation can efficiently be parallelized.

1.3 Contribution

Although the performance of GPUs with respect to side-channel attacks were evaluated and compared to that of CPUs by precedent publications, the experimental results are - in our opinion - not flawless.
The authors did not consider the overhead caused by reading the necessary data from the hard drive. They either assumed that the data resides in the CPU memory [GRJ14] or argued that the influence of reading the data should not be considered since it effects the GPU and the CPU implementation in the same way [BLR11]. This argument is invalid since if the overhead is constant for both implementations, obviously the speed-up of the GPU implementation decreases by taking it into account. We cannot say, whether [SSL+14] considered this overhead in their evaluation. For a fair comparison, it is desirable to include all overhead into the measurements.
Moreover, it is essential to set the speed-up in relation to the costs of the used hardware, which was done in none of the publications. Apart from that, the comparison between a parallel GPU based first order CPA attack to a serial CPU based one [SSL+14] is not meaningful in the era of multi-core processors. As a last point, there exists — as far as we know — no performance evaluation for higher order side-channel attacks.
Our contribution is to analyse the performance of GPU assisted side-channel attacks for first and higher orders under, as far as possible, fair conditions by including any overhead for the attack into the measurements. Furthermore, we set the speed-up in relation to the costs to conclude, if it is cost economical to run side-channel attacks on GPUs.

1.4 Outline

The thesis is organized as follows. Chapter 2 focuses on background information, which are necessary to understand the successive chapters. By this, we provide information on the CUDA programming model, execution model, and memory hierarchy. In the second place, we explain the concept of leakage occurring in integrated circuits and how to exploit this leakage through correlation based side-channel attacks. Furthermore, we introduce robust one-pass formulas and finish the chapter by presenting masking and trace misalignment as two examples of countermeasures.
After that, the implementation platform is presented in Chapter 3. Next, we discuss various GPU-assisted CPA attacks for first and higher orders in conjunction with code optimizations in order to sustain an optimal runtime in Chapter 4. Their performance is discussed and compared to CPU-based implementations in Chapter 5. Finally, the work is summarized by our conclusion and ideas on further work.

2 Background

The background information provided in the following are vital to understand the GPU assisted first and higher order CPA attacks discussed in Chapter 4. The chapter is split into two parts, a comprehensive overview of CUDA followed by a in-depth look at the concept of correlation based side-channel attacks.

2.1 CUDA

Both, CPUs and GPUs are suitable for sequential and parallel programming. However, due to their different architecture, one does not perform as well on a certain task as the other. The common approach for parallel programming is to use the CPU and the GPU in a heterogeneous fashion. Since every parallel program contains sequential parts, those parts are worked off by the CPU, while the parallel fragments are computed by the GPU. The CUDA programming model was designed for exactly this purpose.

2.1.1 The CUDA Programming Model

In 2003, researchers began to explore the use of GPUs to solve computational-intensive science and engineering problems. However, GPUs were designed to match the features required by the graphics Application Programming Interface (API). In the programming model of GPGPU, the programmer had to express the specific problem in a complex language like OpenGL. [KWm12]
The computation had to be written as a pixel shader. The collection of input data had to be stored in texture images and issued by submitting triangles to the GPU. The output had to be casted as a set of pixels generated from the raster operations. [KWm12]
Additionally, GPGPU had memory access problems and a massive performance overhead of the graphics layer APIs as well as a lack of support for bitwise and integer operations. Everything changed with the introduction of CUDA in 2007. CUDA is a parallel computing platform and programming model with a small set of extensions to the C programming language. In addition to sharing many abstractions with other parallel programming models, CUDA provides the following special features to harness the computing power of GPU architectures. [CGM14]

1. A way to organize threads through a hierarchical structure, see Section 2.1.2.
2. A way to access memory through a hierarchical structure, see Section 2.1.3.

From a high level point of view, CUDA provides features to program a heterogeneous computer platform. The CPU and its memory, called the "host", is complemented by

the GPU and its memory, the "device". Host and device are able to communicate over the Peripheral Component Interconnect Express (PCI-E) bus and operate independently from each other. [CGM14]
A key component of the CUDA programming model is the kernel, i.e., the code that is executed on the device. The kernel is a sequential program invoked by multiple threads to achieve parallelism. The CUDA programming model exposes a two level thread hierarchy as illustrated in Figure 2.1. [CGM14]

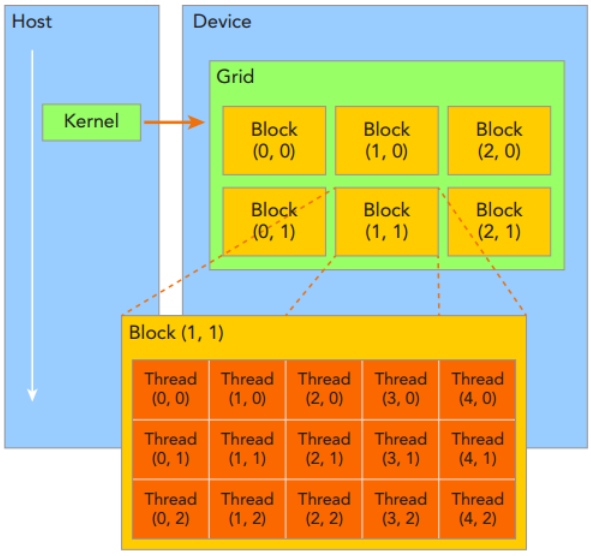

Figure 2.1: Illustration of the CUDA thread hierarchy. [CGM14]

All threads spawned by a kernel are called a grid. A grid consists of several blocks, where each block contains multiple threads that are able to communicate with each other. Grids and blocks can be one-, two-, or three-dimensional. Figure 2.1 shows a two-dimensional grid, which consists of six two-dimensional blocks where each block contains 15 threads. Threads and blocks are distinguished by identifiers. Every thread within a block has its own thread identifier, while every block inside the grid has a unique block identifier in the corresponding dimension. Figure 2.2 illustrates a one-dimensional grid with four blocks and eight threads per block.

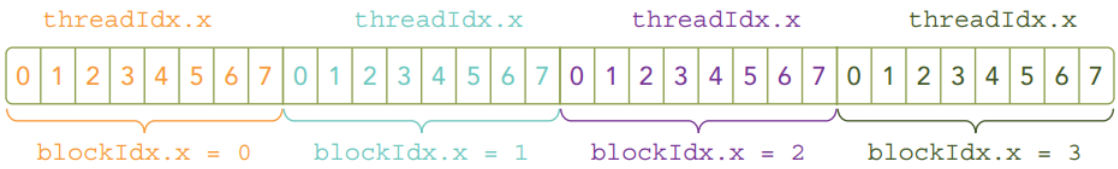

Figure 2.2: The concept of the identifiers threadId and blockId. [CGM14]

The nth thread of each block has the thread identifier `threadIdx.x = n` for $0 \leq n \leq 7$.

Likewise, the kth block has the block identifier `blockIdx.x = k` for $0 \leq k \leq 3$. The global thread identifier in the x-dimension can be computed by the following formula.

$$id_{global} = blockIdx.x \cdot blockDim.x + threadIdx.x$$

Thereby, `blockDim.x` denotes the number of threads per block in the x-dimension. The extension of this concept to multiple dimensions is straightforward. There exists an identifier for every dimension - threadIdx.y / blockIdx.y for the y-dimension, respectively threadIdx.z / blockIdx.z for the z-dimension. Additionally, it is possible to request the size of each block as well as the size of the grid by blockDim.y / gridDim.y for the y-dimension, respectively blockDim.z / gridDim.z for the z-dimension.

The CUDA programming model exposes an abstraction of memory from the GPU architecture in a hierarchical fashion, which is — as a simplified version — illustrated in Figure 2.3.

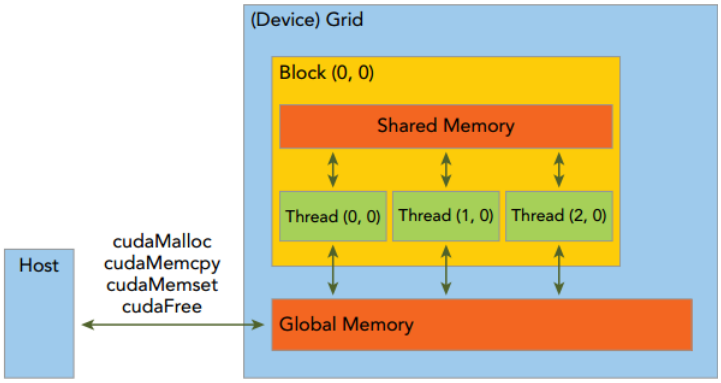

Figure 2.3: A simplified model of the CUDA memory hierarchy. [CGM14]

While every thread inside the grid has access to the global memory in equal measure, only those threads that reside in the same block can access the shared memory of this specific block. CUDA APIs like cudaMalloc, cudaMemcpy, or cudaFree are used to allocate / deallocate global memory or to copy data between the host and the device. A comprehensive view on the CUDA memory hierarchy is provided in Section 2.1.3.

2.1.2 The CUDA Execution Model

To understand the nature of the CUDA execution model, a basic knowledge of the GPU architecture is necessary. Parallelism in GPUs is achieved through the replication of a building block called Streaming Multiprocessor (SM) [Far11]. Figure 2.4 shows the block diagram of the Kepler GPU architecture.

Figure 2.4: An illustration of the Kepler K20X architecture. [CGM14]

The chip consists of 15 SMs, six 64-bit memory controllers, a coherent L2 cache shared by all SMs, a PCI-E 3.0 host interface, and a GigaThread global scheduler, which distributes work to the SMs. Based on the number of blocks and threads per block defined in the kernel's execution configuration, this scheduler allocates one or more blocks to each SM [Far11]. Figure 2.5 shows the composition of one SM.

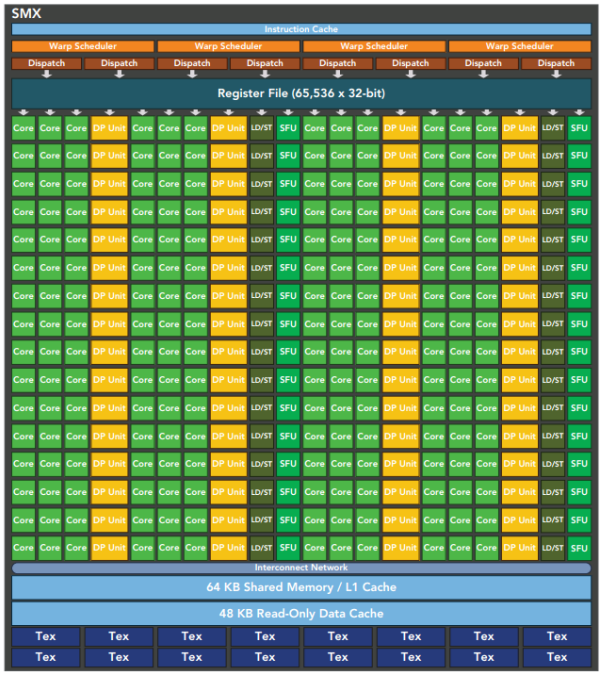

Figure 2.5: The composition of a SM for the Kepler architecture. [nvi12]

Each Kepler SM unit contains 192 single-precision CUDA cores, 64 double-precision units, 32 Special Function Unit (SFU) for fast approximate transcendental operations as well as 32 load / store units. Additionally, four warp schedulers and eight instruction dispatchers allow four warps to be issued and executed concurrently. [nvi12]
All units are connected through an interconnect network and share different memories like register files, shared memory, L1 cache, and texture memory.
As mentioned in Section 2.1.1, all threads execute the statements defined in the kernel. This is only true from a logical point of view. Inside the hardware, not all threads can physically be executed at the same time. When a kernel is invoked, the GigaThread global scheduler distributes the work by assigning a certain number of blocks to each SM. Those blocks are divided into warps, which are groups of 32 threads and can be seen as the basic execution units of a SM. All threads that belong to the same warp execute the same instruction on their own private data. [CGM14] This concept is referred to as Single Instruction Multiple Thread (SIMT) and can be considered as the equivalent to Single Instruction Multiple Data (SIMD) on multi-core CPUs, see [Fly72].
Threads in a warp executing different instructions due to branches is referred to as warp divergence. If warp divergence occurs, each branch is executed sequentially, which leads to an overall performance decrease. [CGM14]
Grids and blocks can be one-, two-, or three-dimensional, cf. Section 2.1.1. Again, this is only true from a logical point of view. From the hardware perspective, all threads are arranged in a one-dimensional fashion. The logical two- or three-dimensional layout is converted to a one-dimensional physical layout and partitioned into warps. [CGM14]
Since the execution context of a warp, which consists of program counters, registers, and shared memory, is maintained on-chip during the lifetime of the warp, switching between contexts has no cost [CGM14]. This is referred to as zero-overhead thread scheduling [KWm12].
Once the execution context has been allocated to a block, this block becomes active and its warps are called active warps. Active warps are further divided into:

- selected warp - a warp that is actively executing.
- stalled warp - a warp that is not ready for execution.
- eligible warp - a warp that is ready for execution but currently not actively executing.

The warp schedulers on a SM select active warps on every cycle and dispatch them to the execution units. [CGM14] Eligible warps are selected for execution based on a prioritized scheduling policy. Running multiple warps per SM is the only way a GPU can hide both, Arithmetic Logic Unit (ALU) and memory latencies in order to keep the execution units busy [Far11].

2.1.3 The CUDA Memory Hierarchy

CUDA exposes a memory hierarchy similar to that of CPUs, where memories with different sizes and performance are implemented to hide latency. Figure 2.6 provides an

overview of the programmable memory types of the CUDA memory hierarchy. Besides that, CUDA offers various types of non-programmable memories like L1 cache, L2 cache, read-only constant cache, and read-only texture cache [CGM14].

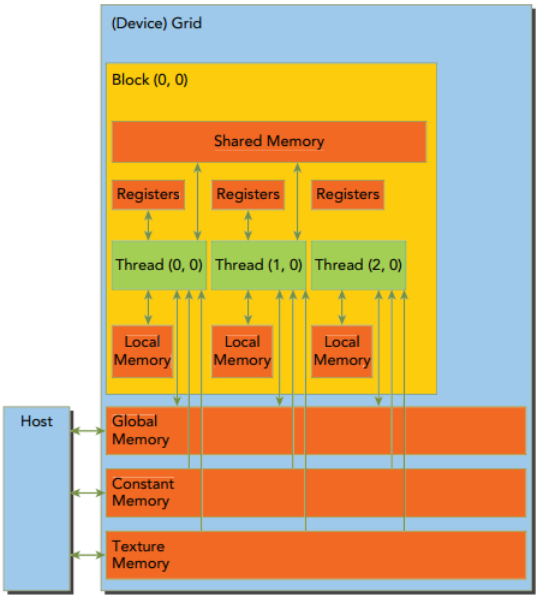

Figure 2.6: Overview of the CUDA programmable memory hierarchy. [CGM14]

Every thread has its own local memory and registers. Blocks do have their own shared memory with which threads that belong to the same block are able to communicate with each other. All threads have access to the global-, constant-, and texture-memory. While the global memory is a read / write memory, constant- and texture-memory are read only memories optimized for different use cases. [CGM14]
Registers are the fastest memory on GPUs and are a very precious resource since they are the only memory with sufficient bandwidth and low latency to deliver peak performance [Far11].
If not explicitly stated otherwise, variables are generally stored in registers. Arrays with constant indices that can be determined at compile time are stored in registers as well. For the Kepler architecture, up to 255 registers can be assigned to each thread. Variables that are eligible for registers but do not fit into the register space, for instance arrays whose indices can not be determined at compile time respectively large local structures or arrays, reside in local memory. Values spilled to local memory reside in the same physical location as global memory and are cached in a per-SM L1 cache and a per-device L2 cache. [CGM14]
Constant memory is predestined to be used, if all threads in a warp have to read the same memory address. Constant memory is capable of broadcasting 32-bit per warp per two clocks per SM [Far11]. A typical application is the involvement of a coefficient in the evaluation of a mathematical formula. Since every thread has to read the same coefficient

and therefore, the same memory address, it is advantageous to store the coefficient in the constant memory.

Texture memory is used for visualization and is optimized for two-dimensional spatial locality. Textures do have limited processing capabilities that can efficiently unpack and broadcast data. Moreover, 9-bit computational units are able to perform out-of-bound index handling, interpolation, and format conversion. [Far11]

In the following, we provide a comprehensive overview on global- and shared-memory since they are the most commonly used memories from the perspective of the programmer. In order to achieve an outstanding computation performance, it is essential to pay attention on the optimal use of those memories. As mentioned before, registers are the only memory type that can deliver peak performance. In converse argument, most CUDA applications are memory bounded rather than being computational bounded. The incorrect use of global- and shared-memory leads to an enormous decrease in performance.

Global memory was designed to quickly stream memory blocks of data into the SM. From the developer's perspective, global-memory accesses need to be perfectly coalesced. A coalesced memory access means that the hardware can coalesce or combine the memory requests from the threads into a single memory transaction. From a hardware perspective, memory requests are issued in the context of warps. Thus, the 32 addresses of a warp should ideally address a continuous, aligned region to stream data from global memory at the highest bandwidth. [Far11]

Load operations from global memory can either be cached or non-cached, depending on whether the L1 cache is involved or not. For cached load operations, the SM firstly attempts to find the data in the L1 cache and, in case of a cache-miss, in the L2 cache. Failing this as well, a 128-byte cache-line load is issued. In case of non-cached loads, the SM does not look for the data in the L1 cache. If the data does not reside in the L2 cache, a 32-byte global-memory load is issued. This can lead to a better performance if a 128-byte cache-line fetch would be wasteful. [Far11]

There are two characteristics that should be considered for efficient global memory use, aligned memory-accesses and coalesced memory-accesses. An aligned memory-access occurs, if the first address of a memory transaction is a multiple of the cache-line size. A coalesced memory-access occurs, if all 32 threads in a warp access a continuous chunk of memory. [CGM14]

Therefore, an optimal access pattern is an aligned and coalesced memory-access, which is illustrated in Figure 2.7.

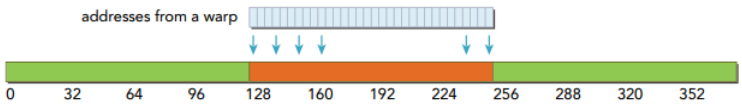

Figure 2.7: An example for an aligned and coalesced memory load-operation. [CGM14]

Because all threads in the warp access a continuous memory location and the first address

is a multiple of the L1 cache-line size of 128 bytes[1], the memory access is aligned and coalesced. The 128 bytes can be served with one memory transaction. Figure 2.8 shows the case of a coalesced but misaligned memory load-operation.

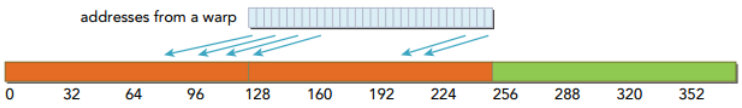

Figure 2.8: An example for a coalesced but misaligned memory load-operation. [CGM14]

Because the requested addresses fall into two 128 byte cache-lines, two memory transactions are required to fulfill the request. For an uncached memory load-operation, where only the L2 cache with a cache-line size of 32 bytes is involved, the memory request is aligned and coalesced. This holds under the assumption that the first requested address is 64. Figure 2.9 illustrates an aligned but uncoalesced memory load-operation.

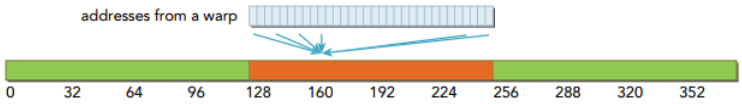

Figure 2.9: An example for an aligned but uncoalesced memory load-operation. [CGM14]

Because all threads access the same memory address, only one transactions has to be conducted to fulfill the request. However, only four out of 128 bytes are payload. Therefore, the bus utilization is merely 3.125%. Finally, Figure 2.10 represents the worst case, a misaligned and uncoalesced memory-access.

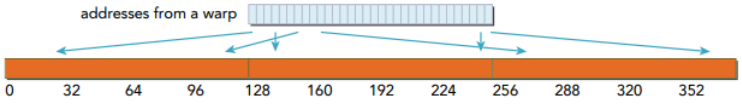

Figure 2.10: An example for a misaligned and uncoalesced memory load-operation. [CGM14]

If the requested 128 bytes fall within N different cache lines, N memory requests have to be issued to serve all 128 bytes. The mechanics work the same for memory store-operations with the difference that only the L2 cache is involved. Therefore, the alignment has to be assured with regards to 32 bytes. [CGM14]

[1] We assume a cached memory load-operation.

In contrast to global memory, shared memory is an on-chip memory and can be viewed as a user-controlled L1 cache. The L1 cache and the shared memory share a 64 KByte block per SM. This block is, in the case of the Kepler architecture, configurable in 16k blocks in favour of either shared memory or L1 cache. With a throughput of 1.5 TB/s, shared memory is approximately seven times faster than global memory, but still approximately five times slower than registers. [Coo12]

There are numerous applications for shared memory. First of all, it facilitates intra-block communication and cooperation between threads. Moreover, it can be used as a program managed cache for global-memory data to reduce global-memory bandwidth required by the kernel. Apart from that, shared memory is able to act as a scratchpad memory to improve global-memory access patterns, namely aligned and coalesced memory accesses. [CGM14] For the last point, we refer to [nvi10].

Shared memory accesses are issued per warp and one shared-memory request is ideally served in one transaction. In the worst case, each request is issued sequentially, resulting in 32 transactions for one request. If multiple threads in a warp access the same location in shared memory, one thread fetches the value of that location and broadcasts it to all other threads. [CGM14]

Shared memory is a bank-switched architecture with 32 equally sized banks, where each bank has a size of four bytes[2]. A bank can serve one operation per cycle and if every thread operates on a different bank, a shared-memory access by a warp is issued in one cycle. [Coo12]

However, if multiple addresses of a shared-memory request fall into the same bank, a bank conflict occurs. This means that the request is served by multiple transactions [CGM14]. Bank conflicts can be viewed as the equivalent to misaligned / uncoalesced global-memory accesses.

There are three shared memory access-patterns. The most common pattern is the parallel access, where multiple addresses across multiple banks are accessed. Secondly, a serial access occurs, if multiple addresses within the same bank are accessed, which results in a bank conflict. The last access pattern is the broadcast access, where all threads in a warp read the same address in a single bank. This kind of request is satisfied in one transaction and the value of the address is broadcasted to all threads. [CGM14] Figure 2.11 and Figure 2.12 show examples for bank conflict-free shared-memory accesses.

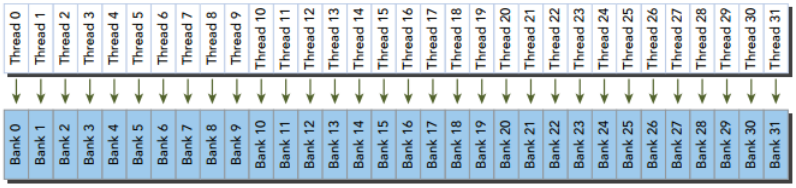

Figure 2.11: A regular bank conflict-free shared-memory access. [CGM14]

[2]The Kepler architecture has a special operation-mode in which eight byte wide banks are used.

The shared memory access is bank conflict-free since every thread accesses a different bank.

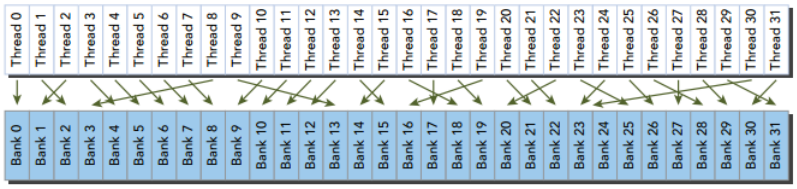

Figure 2.12: An irregular bank conflict-free shared-memory access. [CGM14]

The shared-memory access is conflict-free for the same reason as above. However, in this case the access is random rather than continuous. Figure 2.13 illustrates the occurrence of a bank conflict.

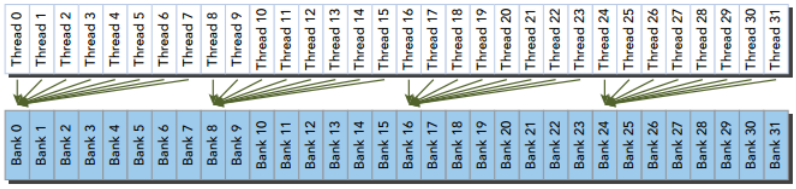

Figure 2.13: A shared memory access resulting in a bank conflict. [CGM14]

Eight consecutive threads of the warp access a single bank. Under the assumption that not all threads access the same address in the corresponding bank, a bank conflict occurs. Multiple transactions are required to satisfy the memory access, depending on the number of bank conflicts. [CGM14]

Memory padding is a common technique to prohibit the occurrence of shared-memory bank conflicts. Figure 2.14 shows an example of shared-memory padding for five memory banks.

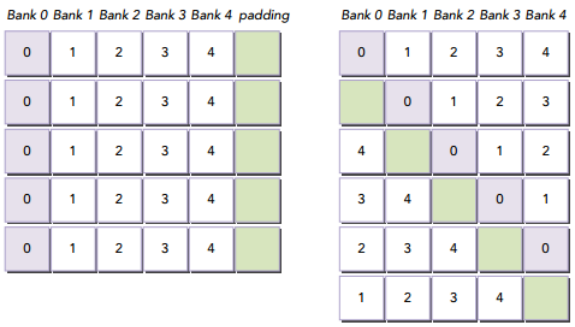

Figure 2.14: An example of shared memory padding to avoid bank conflicts. [CGM14]

We suppose that a two-dimensional array, which consists of five columns and five rows, resides in shared memory. The mapping of the array members to memory banks is shown on the left-hand side. If all threads access a different address within bank x, a five-way bank conflict occurs.
By appending an additional column to the array, we force CUDA to map the array members to the memory banks as shown on the right-hand side. Threads previously accessing different addresses within the same bank, now access different addresses inside different banks, which reflects in a bank conflict-free shared-memory access. Obviously, shared memory padding produces overhead as the padding column does not contain useful data. [CGM14]
As already mentioned, shared memory enables the possibility for threads in the same block to communicate and cooperate with each other. If multiple threads intend to read / write the same data into / from shared memory, race conditions can arise. In order to thwart the occurrence of race conditions, synchronization has to be applied. For this, CUDA provides two approaches, namely barriers and memory fences. As soon as a thread reaches a barrier, it stops executing further until all other threads in the block have reached the same barrier. At a memory fence, all threads stall until any modification to the memory is visible to all other threads. [CGM14]

2.2 Side-Channel Attacks

Unlike classical cryptanalysis, which investigates the mathematical structure of a cryptographic primitive, side-channel attacks focus specifically on the implementation. Even the implementation of strong ciphers like Advanced Encryption Standard (AES) that are considered to be secure against classical cryptanalysis — linear cryptanalysis, differential cryptanalysis, lattice based attacks — can be broken through side-channel attacks.
We differentiate between active (Bellcore attack [BDL01], Lenstra's attack [Len96]) and passive (Simple Power Analysis (SPA), DPA, timing attacks) attacks. In an active attack, the adversary interferes the encryption through power spikes, lasers, chemicals, or x-rays to force a malfunction during the computation, whereupon conclusions on the key can be drawn. Passive attacks measure the data dependent power consumption of a device or the number of clock cycles for a specific operation to exploit information about the secret key.

2.2.1 Mean, Variance, Covariance, and Correlation

In this section, we introduce all necessary formulas for a correlation based side-channel attack. First, we define mean, variance, covariance, and Pearson correlation. Secondly, we highlight how to efficiently compute the Pearson correlation for higher order attacks by the use of robust one-pass formulas.
For higher order attacks, the traces have to be preprocessed. This preprocessing involves the computation of mean-free values for each sample point, which are then raised to the power of d for a dth order attack. In addition to the problem of preprocessing, the traces become more prone to noise. Thus, the number of required traces to successfully run a

dth order attack grows exponentially in d with respect to the noise standard deviation. [SMG15]

The common strategy for trace preprocessing, known as three-pass, parses each trace three times to i) obtain the means, ii) combine points by their mean-free products, and iii) compute the correlation. This strategy has crucial disadvantages, namely i) the traces have to be processed three times, ii) by adding more traces, the last two steps have to be repeated, and iii) it is not easily possible to parallelize the computation by splitting the entire trace pull into smaller subsets. [SMG15]

Robust one-pass formulas can overcome these shortcomings. The following formulas are mainly based on the work of Schneider et al. [SMG15].

We define the dth order raw statistical moment of X by $M_d = E(X^d)$ with $E(\cdot)$ being the expectation operator. Furthermore, we define the dth order central moment for $d > 1$ by $CM_d = E((X-m)^d)$ and the dth order standardized moment as $SM_d = E\left(\left(\frac{X-m}{s}\right)^d\right)$ for $d > 2$.

The first order statistical moment M_1 is the mean and in the discrete domain, we can write

$$M_1 = \mu = \frac{1}{n}\sum_{i=1}^{n} X_i$$

Furthermore, the second order central moment is the variance and in the discrete domain, we can write

$$CM_2 = s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

The covariance in the discrete domain is defined as

$$cov_{x,y} = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x}) \cdot (y_i - \bar{y})$$

Finally, the Pearson correlation coefficient is the normalized covariance

$$\rho = \frac{cov(T, L)}{s_t s_l}$$

We rewrite the Pearson correlation coefficient for a first order attack as follows

$$\rho = \frac{\frac{1}{n}\sum_{i=1}^{n}(t_i - \mu_t) \cdot (l_i - \mu_l)}{\sqrt{\frac{1}{n}\sum_{i=1}^{n}(t_i - \mu_t)^2 \cdot \frac{1}{n}\sum_{i=1}^{n}(l_i - \mu_l)^2}} = \frac{\frac{1}{n}ACS_1}{\sqrt{\frac{1}{n}CS_{2,T} \cdot \frac{1}{n}CS_{2,L}}}$$

where $CS_{d,X} = \sum_{i=1}^{n}(x_i - \mu_x)^d$ is the dth order centralized sum and

$ACS_1 = \sum_{i=1}^{n}(t_i - \mu_t)(l_i - \mu_l)$ is the first order adjusted centralized sum.

For the iterative formulas, we assume two sets Q_1 and Q_2 with cardinality $|Q_1| = n_1$ and $|Q_2| = n_2$. The mean of the set $Q = Q_1 \cup Q_2$ is defined as

$$M_{1,Q} = \frac{M_{1,Q_1} \cdot n_1 + M_{1,Q_2} \cdot n_2}{n}$$

The dth order centralized sum $CS_{d,X}$ for the set $Q = Q_1 \cup Q_2$ is

$$CS_{d,Q} = CS_{d,Q_1} + CS_{d,Q_2} + \sum_{p=1}^{d-2}\binom{p}{d}\left[\left(\frac{-n_2}{n}\right)^p CS_{d-p,Q_1} + \left(\frac{n_1}{n}\right)^p CS_{d-p,Q_2}\right]\Delta^p$$
$$+ \left(\frac{n_1 n_2}{n}\Delta\right)^d \left[\left(\frac{1}{n_2}\right)^{d-1} - \left(\frac{-1}{n_1}\right)^{d-1}\right] \ , \Delta = M_{1,Q_2} - M_{1,Q_1}$$

Furthermore, the dth order adjusted centralized sum for the set $Q = Q_1 \cup Q_2$ is

$$ACS_{d,Q} = ACS_{d,Q_1} + ACS_{d,Q_2} + \frac{\Delta_l}{n}\left(n_1 CS_{d,Q_2} - n_2 CS_{d,Q_1}\right) + \sum_{p=1}^{d-1}\binom{p}{d}\left(\frac{\Delta_t}{n}\right)^p$$
$$\left[(-n_2)^p ACS_{d-p,Q_1} + (n_1)^p ACS_{d-p,Q_2} + \frac{\Delta_l}{n}\left((-n_2)^{p+1} CS_{d-p,Q_1} + (n_1)^{p+1} CS_{d-p,Q_2}\right)\right]$$

$$+ \frac{(n_1(-n_2)^{d+1} + n_2(n_1)^{d+1})}{n^{d+1}}(\Delta_t)^d \Delta_l \ , \Delta_t = \mu_{t,Q_2} - \mu_{t,Q_1}, \Delta_l = \mu_{l,Q_2} - \mu_{l,Q_1}$$

For $|Q_2| = n_2 = 1$ where Q_2 consists of the element y, the iterative formulas become incremental ones. $M_{1,Q}$, $CS_{d,Q}$, and $ACS_{d,Q}$ can be simplified to

$$M_{1,Q} = M_{1,Q_1} + \frac{\Delta}{n} \ , \Delta = y - M_{1,Q_1}$$

$$CS_{d,Q} = CS_{d,Q_1} + \sum_{p=1}^{d-2}\binom{p}{d} CS_{d-p,Q_1} + \left(\frac{-\Delta}{n}\right)^p + \left(\frac{n-1}{n}\Delta\right)^d \left[1 - \left(\frac{-1}{n-1}\right)^{d-1}\right]$$

$$ACS_{d,Q} = ACS_{d,Q_1} + CS_{d,Q_1}\left(-\frac{\Delta_l}{n}\right) + \sum_{p=1}^{d-1}\binom{p}{d}\left(\frac{-\Delta_t}{n}\right)^p \left[ACS_{d-p,Q_1} + CS_{d-p,Q_1}\left(-\frac{\Delta_l}{n}\right)\right]$$
$$+ \frac{(-1)^{d+1}(n-1) + (n-1)^{d+1}}{n^{d+1}}(\Delta_t)^d \Delta_l$$

Finally, the Pearson correlation coefficient for a dth order CPA attack with $d > 1$ can be expressed as

$$\rho = \frac{\frac{1}{n} ACS_d}{\sqrt{\frac{1}{n}\left(CS_{2 \cdot d,T} - \frac{{CS_{d,T}}^2}{n}\right) \frac{1}{n} CS_{2,L}}}$$

Schneider et al. [SMG15] provide explicit iterative and incremental formulas for computing the dth order centralized sum and adjusted centralized sum up to $d = 5$.

The approach for computing the Pearson correlation for a dth order attack is as follows. Let D be the number of processed traces and T the number of sample points per trace. For each sample point, we initialize $CS_{d,T}$, $CS_{2,L}$, μ_T, μ_L, and ACS_d with zero. For each trace x for $1 \leq x \leq D$, we apply the incremental formulas to update the values of $CS_{d,T}$, $CS_{2,L}$, μ_T, μ_L, and ACS_d, based on their previous values and the trace x. After D traces are processed, we compute the Pearson correlation based on formula above.
To distribute the work across two GPUs, we apply the approach for the traces $1 \leq x_1 \leq D_1$ on the first GPU and for the traces $D_1 + 1 \leq x_2 \leq D$ on the second GPU to obtain the values of $CS_{d,T}$, $CS_{2,L}$, μ_T, μ_L, and ACS_d for the sets Q_1 respectively Q_2. Subsequently, we apply the iterative formulas to obtain the values of $CS_{d,T}$, $CS_{2,L}$, μ_T, μ_L, and ACS_d for the set $Q = Q_1 \cup Q_2$ to finally compute the Pearson correlation based on the formula above. The extension of this approach to distribute the work among n GPUs is straightforward.

2.2.2 Leakage and Power Models

In this section, we discuss the concept of leakage occurring in integrated circuits and point out, how an attacker can exploit this leakage in a CPA attack. Furthermore, several power models, used to estimate the power consumption of the attacked device, are presented. The discussion is mainly based on [MOP08].

The power consumption of a Complementary Metal Oxide Semiconductor (CMOS) circuit is the sum of the power consumption of all logic cells of the circuit. The power consumption of a logic cell is the accumulation of the static and dynamic power consumption, $P_{cell} = P_{static} + P_{dynamic}$. Thereby, P_{static} is the constantly consumed power if there is no switching activity, while $P_{dynamic}$ is the short-time consumed power if an internal signal or the output of the cell switches. Typically, $P_{dynamic}$ is the dominant factor in the total power consumption and depends on the processed data. Figure 2.15 shows the switching effect of the output signal on the power consumption of a CMOS inverter.

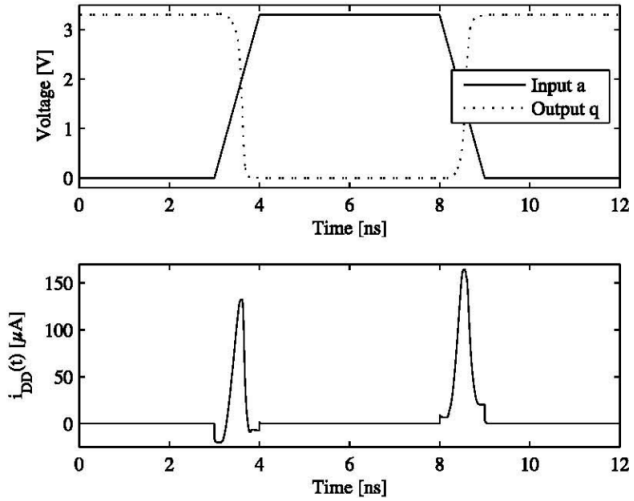

Figure 2.15: The switching effect of the output value of a CMOS inverter on its power consumption. [MOP08]

As soon as the output value of the inverter switches from $1 \rightarrow 0$ respectively from $0 \rightarrow 1$, the power consumption of the cell increases significantly, caused by the additionally consumed dynamic power $P_{dynamic}$. For the $0 \rightarrow 1$ transition, the peak of the current is higher since the capacity must be charged, which is not necessary for a $1 \rightarrow 0$ transition.
In a CMOS circuit, many combinational logic cells are connected to each other to form a multi-stage combinational circuit. Due to the fact that the propagation delay of the signals vary, input signals at a combinational cell have different times of arrival. This leads to temporary states of the output of the cell, so called glitches. Glitches are data dependent and can become the dominant factor in the total power consumption since they propagate like an avalanche through the stages of the multi-stage combinational circuit.
In a CPA attack, a power model is applied to estimate the power consumption of the target device, which is then compared to the real power consumption by the Pearson correlation, see Section 2.2.3. We introduce the Hamming-weight model and the Hamming-distance model, being the two most commonly used power models.
The Hamming-weight model is the most basic power model and is commonly used when the attacker does not have a lot of knowledge about the design. For instance, he does not know the consecutive values of the data in some part of the process. [MBMT13]
Let D be a binary number with $D = \sum_{j=0}^{m-1} d_j 2^j$. The Hamming-weight calculates the number of "1" in the binary number and is defined as $HW(D) = \sum_{j=0}^{m-1} d_j$. The Hamming-weight model works best for software implementations, where the power consumption mainly depends on how many bits are switched on the bus of the device [Hna10]. The more bits are set to "1" on the bus, the more power is consumed.

The Hamming-distance model exploits the fact that a transition of the output value from $1 \rightarrow 0$ respectively $0 \rightarrow 1$ leads to a higher power consumption than the transition $0 \rightarrow 0$ respectively $1 \rightarrow 1$. The Hamming-distance model works best for hardware implementations, where the power consumption mainly depends on the switching of bits inside registers. [Hna10]
The Hamming-distance is defined as $HD(N_1, N_2) = HW(N_1 \oplus N_2)$. Because of the fact that the Hamming-distance model requires a reference-value N_1, the attacker has to have knowledge about two intermediate values rather than one as in the Hamming-weight model.
There are more power models used in practice. The glitch model counts the number of glitches occurring in each clock cycle and therefore, requires a lot of knowledge about the target device. The zero-value model is a simple but yet efficient power model. It outputs "0" if the intermediate values is zero and "1" for all other values. The idea is that the output of the AES Sbox is "0", if the input value is "0". For all other input values, the output is never "0". The Bit-model outputs a specific bit of the intermediate value. Often the most / least significant bit is considered. Weighted models are equivalent to the Hamming-weight / Hamming-distance model. In contrast, they weight each bit with a factor. Stochastic models are a special case of weighted models, where a least square estimator is used as the weighting factor. For template models, the attacker estimates the power consumption by averaging over many recorded traces using a known secret key.

2.2.3 Correlation Power Analysis

CPA is an advanced passive side-channel attack that is capable of revealing information from extremely noisy traces. A large number of traces is recorded whereupon statistical methods are applied on them. In order to be able to apply those statistical methods, the attacker has to have knowledge about either the plaintexts or the ciphertexts. Apart from this, the key has to be the same for every execution of the targeted algorithm. The idea is to correlate the measured power consumption with the power model in order to reveal the used secret key. Thereby, the power model is the estimated power consumption of the device for every possible key. The following approach is based on [MOP08].

Select an Intermediate Value To select a suitable intermediate value, the attacker has to ensure that i) the value depends on some bits of the secret key, ii) the value depends on either the plaintexts or the ciphertexts, and iii) the dependency should be non-linear whenever possible.
The first condition provides the ability to apply the divide and conquer principle to extract only some bits of the key at a time. Since the key stays the same for every execution, a known input value that influences the intermediate values is required. Non-linear dependencies lead to better results of the statistical methods conducted in the last step of the attack since a change of one bit of the input value implies numerous bit changes in the intermediate value, the so called avalanche effect. In the case of AES or the Data Encryption Standard (DES), the chosen intermediate value is typically the output of

the S-Box. We denote the intermediate value as a function $f(c, k)$ where c is a known non-constant value and k denotes a small part of the key.

Measure the Power Consumption In the second step, the attacker measures the power consumption of the device while it encrypts / decrypts D different data blocks. The conjunction of this data is written as a vector $p = (p_1, \ldots, p_D)$ where p_i for $i = 1, \ldots, D$ represents the data of the ith encryption / decryption. The power trace that corresponds to data block p_i is expressed as a vector $t_i = (t_{i,1}, \ldots, t_{i,U})$ for $i = 1, \ldots, U$ where U denotes the number of sample points of the trace. Thus, the traces can be represented as a $D \times U$ matrix.

$$\mathrm{T}=\begin{bmatrix} t_{1,1} & t_{1,2} & \ldots & t_{1,U} \\ \vdots & \ddots & & \vdots \\ t_{D,1} & t_{D,2} & \ldots & t_{D,U} \end{bmatrix}$$

It is noteworthy that the measured traces should be correctly aligned so that every value of each column of T is caused by the same operation.

Calculate Hypothetical Intermediate Values In the third instance, the attacker has to calculate a hypothetical intermediate value for every potential key, written as $k = (k_1, ..., k_K)$ where K is the total number of possible keys. Each value of k is called a key hypotheses. For every p_i and k_j, the attacker computes $v_{i,j} = f(p_i, k_j)$ for $i = 1, \ldots, D$ and $j = 1, \ldots, K$. The result is represented as $D \times K$ matrix.

$$\mathrm{V}=\begin{bmatrix} v_{1,1} & v_{1,2} & \ldots & v_{1,K} \\ \vdots & \ddots & & \vdots \\ v_{D,1} & v_{D,2} & \ldots & v_{D,K} \end{bmatrix}$$

Each column of V contains the intermediate value based on the key hypothesis k_j. Since k includes every possible key, one column of V contains the intermediate value that is computed based on the correct key k_{ck}.

Map the Intermediate Values to Power Consumption Values The adversary maps each hypothetical intermediate value $v_{i,j}$ to a hypothetical power consumption value $h_{i,j}$ ($v_{i,j} \rightarrow h_{i,j}$ for $i = 1, \ldots, D$ and $j = 1, \ldots, K$) using an appropriate power model. The values $h_{i,j}$ are represented as a $D \times K$ matrix.

$$\mathrm{H}=\begin{bmatrix} h_{1,1} & h_{1,2} & \ldots & h_{1,K} \\ \vdots & \ddots & & \vdots \\ h_{D,1} & h_{D,2} & \ldots & h_{D,K} \end{bmatrix}$$

Thereby, the correct choice of the power model is crucial for the efficiency of the attack.

Compare the Hypothetical Power Consumption Values to the Power Traces In the last step, the adversary compares each column h_i of H to every column t_j of T. In most cases, the Pearson correlation coefficient is used. Finally, the attacker obtains a $K \times U$ matrix.

$$\mathrm{R}=\begin{bmatrix} r_{1,1} & r_{1,2} & \dots & r_{1,U} \\ \vdots & \ddots & & \vdots \\ r_{K,1} & r_{K,2} & \dots & r_{K,U} \end{bmatrix}$$

Each element $r_{i,j}$ represents the comparison between the columns h_i and t_j. The higher the value of $r_{i,j}$, the more those columns match. Therefore, the largest value $r_{i,j}$ most likely reveals the correct key k_{ck}.

2.2.4 Masking

Masking is a common used technique to counteract CPA attacks. It aims to make the power consumption independent of the intermediate values, even if the device has a data-dependent power consumption. This is achieved by randomizing the intermediate values that are processed by the cryptographic device [MOP08].
Each intermediate value v is concealed by a mask m as $v_m = v * m$. The mask is a randomly chosen value that varies from execution to execution and is therefore not known to the attacker. The operation used to bind the mask to the intermediate value depends on the operations of the cryptographic algorithm. In most cases, $*$ is either a multiplication, addition, or the XOR operator.[3] If the cryptographic algorithm uses both kinds of operations, a combined boolean and arithmetic masking has to be applied. It is vital that all intermediate values are concealed by a mask. If two masked intermediate values are combined by an operator, the result has to be masked as well. [MOP08]
By concealing an intermediate value v by n masks as $v_m = v * m_1 * m_2 * ... * m_n$, we are able prevent up to an nth order attack at the cost of additional execution time and memory use. [CJRR99]

2.2.5 Alignment

In the following, we discuss trace misalignment as one kind of hiding countermeasure to harden cryptographic algorithms against side-channel attacks. The information are based on [MOP08].
Trace misalignment is achieved through the random insertion of dummy operations or operation shuffling to randomly change the algorithmic execution. Therefore, the attacked intermediate value is processed at a different moment of time in each power trace.
To launch an attack on misaligned traces, the attacker can either (i) perform the attack on the traces as they are, (ii) try to align the traces before performing the attack, or (iii)

[3] In the case of multiplication or addition, we refer to as arithmetic masking, else we refer to as boolean masking. Arithmetic masking in the context of asymmetric cryptography is called blinding.

preprocess the traces to reduce the effect of misalignment. Obviously, the best way to deal with misaligned traces is to successfully align them in order to completely negate the countermeasure.
The alignment process is divided into two steps. First, a pattern that occurs in the first trace is selected. In order to obtain optimal results, the pattern should have to following properties.

1. The pattern should contain unique properties like characteristic maximal / minimal peaks.
2. The pattern should not depend on intermediate values of the cryptographic algorithm as this degrades the matching result.
3. The pattern should preferably be short to avoid intermediate results in the pattern.
4. The pattern should be located closely to the processing of the attacked intermediate value.

Subsequently, the attacker tries to find the pattern in all other power traces by determining the position of the trace that matches the pattern best. This can be accomplished by evaluating the cross correlation between the pattern and the trace. The cross correlation estimates the degree on which two series are correlated to each other. After the probable location of the pattern within a trace is estimated, the attacker shifts the trace in such a way that the pattern occurs at the same position as in the first trace.

3 Implementation Platform

Table 3.1 illustrates the hardware components of the server on which we have done the implementations. Furthermore, we provide an overview of the hardware capabilities of the GPUs inside the server in Table 3.2. Finally, Table 3.3 shows the most important CUDA compute capabilities of the Tesla K80.

Component	**Description**	**Characteristics**	**Number**
CPU	INTEL Xeon E5-2650 V3	10 cores @ 2300 MHz, 25 MB cache	2
RAM	16GB DDR4	ECC registered DDR4-2133	16
GPU	Tesla K80	see Table 3.2	2
SSD	Samsung SSD PM871	512 GB, SATA 6 GB/s	2
HDD	WD Red	1 TB, SATA 6 GB/s, 16 MB cache	1

Table 3.1: Overview of the server hardware components.

Specification	**Tesla GK210**
Number of GPUs	2x Tesla GK210B
Number of processor cores	2496
Core clocks	560 MHz (base)
	562-875 MHz (boost)
Memory clock	2,5 GHz
Memory size	24 GB per board
	12 GB per GPU
Memory I/O	384 Bit GDDR5
Memory bandwidth	480 GB/s per board
	240 GB/s per GPU
ECC available	Yes
PCI-Express	Gen3

Table 3.2: Specification of the Tesla K80 board. [nvi15]

Specification	**Tesla K80 (CUDA cc 3.7)**
Maximum dimensionality of a grid of thread blocks	3
Maximum x-dimension of a grid of thread blocks	$2^{31} - 1$
Maximum y- or z-dimension of a grid of thread blocks	65535
Maximum dimensionality of a thread block	3
Maximum x- or y-dimension of a thread block	1024
Maximum z-dimension of a thread block	64
Maximum number of threads per block	1024
Warp size	32
Maximum number of resident blocks per multiprocessor	16
Maximum number of resident warps per multiprocessor	64
Maximum number of resident threads per multiprocessor	2048
Number of 32-Bit registers per multiprocessor	128k
Maximum number of 32-Bit registers per thread block	64k
Maximum number of 32-Bit registers per thread	255
Maximum amount of shared memory per multiprocessor	112k
Maximum amount of shared memory per thread block	48k
Number of shared memory banks	32
Amount of local memory per thread	512k
Constant memory size	64k

Table 3.3: The CUDA compute capabilities of the Tesla K80. [nvi]

4 Implementation

This chapter discusses various implementations of GPU assisted correlation power analysis on AES. In the first instance, we present our implementations for first order CPA using an optimized version of the algorithm provided by Chang et al. [CDOR09]. In this context, we depict how to deal with the memory limitations of GPUs to be able to handle arbitrary large numbers of traces respectively sample points per trace. Next, we provide a comprehensive overview of higher order CPA implementations using robust one-pass formulas introduced in Section 2.2.1.
Moreover, we illustrate how to enhance the performance to obtain an optimal runtime. However, benchmark results as well as comparisons to CPU based implementations of CPA are provided in Chapter 5.

4.1 First Order CPA

This section reviews two implementations. The native implementation processes the whole trace pull at once. While this is reasonable for small datasets, the approach is not applicable for large datasets due to the limited global memory capacity.
In contrast to that, the iterative version processes the dataset iteratively by splitting the trace pull into smaller subsets. In each iteration, one subset is processed while the final result is obtained by merging the intermediate results.

4.1.1 Native Implementation

The structure of a basic CUDA application includes the following parts.

1. Allocate GPU memory
2. Copy data to work on from CPU to GPU
3. Invoke kernels
4. Copy results from GPU to CPU
5. Deallocate GPU memory

Section 2.2.3 highlighted the different steps of a standard CPA attack. A mapping of those steps as well as the five parts of a basic CUDA application to our native first order CPA implementation is examined in the following.

Besides the necessary declarations of variables and arrays, we allocate memory on the device for (i) the trace matrix, (ii) the plaintext vector, (iii) the leakage-model matrix, (iv) the correlation matrix, (v) the transposed trace matrix, and (vi) the transposed leakage-model matrix. This is accomplished by the CUDA API

```
cudaError_t cudaMalloc(void **devPtr, size_t size)
```

`cudaMalloc` allocates `size` bytes of linear memory on the device and returns it in `*devPtr`. In the case of a failure, `cudaMalloc` returns `cudaErrorMemoryAllocation`. [api]

In the next step, we read the plaintexts and traces from the corresponding files on the hard drive. We omit the listing of those two functions since the specific implementation depends on the file format. Subsequently, we copy the trace matrix and the plaintext vector from the CPU memory into the global memory of the GPU by the CUDA API

```
cudaMemcpy(void *dst, void *src, size_t count, cudaMemcpyKind kind)
```

`cudaMemcpy` copies `count` bytes from the memory area pointed to by `src` to the memory area pointed to by `dst`, where `kind` is one of `cudaMemcpyHostToHost`, `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, or `cudaMemcpyDeviceToDevice`, and specifies the direction of the copy. [api]

After all required data is transfered to the GPU's global memory, we invoke multiple kernels to launch the actual attack. As the intermediate value, we select the output of the AES S-Box since it is predestined for that purpose due to its non-linear nature. The computation of the hypothetical intermediate values as well as the mapping of those values to the power consumption values is computed by a single kernel shown in Listing 4.1.

```
1  __global__ void create_model(uint8_t *d_model, uint8_t *d_plaintexts)
2  {
3      uint8_t Sbox[] = {0x63, 0x7c, 0x77, ... , 0x54, 0xbb, 0x16};
4
5      int ix = blockIdx.x * blockDim.x + threadIdx.x;
6      int iy = blockIdx.y * blockDim.y + threadIdx.y;
7      int idx = iy * key_hypothesis + ix;
8
9      if(ix < key_hypothesis && iy < numTraces)
10     {
11         d_model[idx] = HW(Sbox[(d_plaintexts[iy] ^ ix)]);
12     }
13 }
```

Listing 4.1: Computation of hypothetical intermediate values and mapping of those values to the power consumption values.

As mentioned in Section 2.1.1, every thread has an identifier within its corresponding block, while each block has a unique identifier within the grid in the appropriate dimension. For the algorithm, we create a two-dimensional grid that consists of multiple two-dimensional

blocks. Each thread handles one element of the matrix. For this, the thread is assigned to the proper matrix index by computing `ix` and `iy` as shown in Figure 4.1.

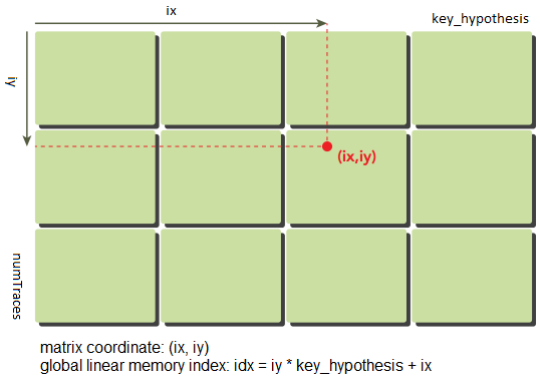

Figure 4.1: Computation of ix and iy to assign each thread to the proper matrix index based on [CGM14].

As the matrix is stored linearly in the memory, we further have to map the matrix coordinates `ix` and `iy` to the global memory index `idx`.
Next, the algorithm computes the Hamming-weight of the output of the S-Box for every possible plaintext byte and every possible key hypothesis. Finally, the result is stored in the correct location of the array `d_model` by using the global memory index `idx`. Thereby, each plaintext byte is addressed by `iy`, while every key hypothesis is addressed by `ix`.
To prevent out-of-bound memory addressing, we ensure that `ix` respectively `iy` does not exceed the number of key hypothesis respectively the number of plaintexts. Out-of-bound addressing occurs, if the size of the matrix is not divisible by the overall number of threads in the appropriate dimension.
Assuming, the algorithm works on a matrix with $ny = 1000$ plaintexts. We set $y = 32$ threads per block in the y-dimension as the execution configuration for the kernel. Therefore, the number of blocks is $b = \left\lceil \frac{ny}{y} \right\rceil = 32$. However, the total number of threads is $t = b \cdot y = 1024$. In a final conclusion, only those threads with an index $iy < 1000$ can be used to compute the matrix. The same holds for threads in the x-dimension.
Arguably, we can set the execution configuration to $y = 50$ and $b = \left\lceil \frac{ny}{y} \right\rceil = 20$, which leads to $t = 1000$ threads. This configuration matches the size of the matrix perfectly. However, to guarantee a high kernel performance, the number of threads per block must be a power of 2 [CGM14].

The computation of the Hamming-weight is shown in Listing 4.2.

```
1  __device__ uint8_t HW(uint8_t x)
2  {
3      x = x - ((x >> 1) & 0x55555555);
4      x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
5
6      return ((x + (x >> 4) & 0xF0F0F0F) * 0x1010101) >> 24;
7  }
```

Listing 4.2: Computation of the Hamming-weight.

The Hamming-weight is determined by a device function, which is an auxiliary function indicated by the prefix *__device__*. The algorithm is highly efficient since it only consists of shift operations, binary AND operations, and additions. For more information, we refer to [War13].

As a precomputation for the evaluation of the Pearson correlation, we transpose both, the leakage-model matrix and the trace matrix. By this, we optimize the memory accesses inside the correlation kernel because of the fact that row based memory accesses are more efficient than column based ones [CGM14]. The algorithm is based on [CGM14] and shown in Listing 4.3.

```
1  __global__ void transpose_model(uint8_t *out, uint8_t *in, const int nx, const int ny)
2  {
3      unsigned int ix = blockDim.x * blockIdx.x + threadIdx.x;
4      unsigned int iy = blockDim.y * blockIdx.y + threadIdx.y;
5
6      if (ix < nx && iy < ny)
7      {
8          out[ix * ny + iy] = in[iy * nx + ix];
9      }
10 }
```

Listing 4.3: Transposition of the leakage-model matrix based on [CGM14].

As the execution configuration, we choose to use a two-dimensional grid, which consists of multiple two-dimensional blocks. First, the algorithm computes the global index of each thread for the x-dimension respectively the y-dimension to store them in the variables `ix` respectively `iy`, similar to Listing 4.1. Each thread of the grid processes one element of the matrix. For this, the algorithm checks if `ix` is smaller than `nx` (number of key hypothesis) and if `iy` is smaller than `ny` (number of plaintexts) to avoid out-of-bound addressing. The transposition itself is a reordering of the matrix elements conducted in line `8`. We omit to list the code for the transposition of the trace matrix as it is the same except for the passed parameters.

The final step of the attack is the computation of the correlation matrix, which is shown in Listing 4.4.

```
__global__ void correlation(float *d_corr, int8_t *d_traces_t, uint8_t *d_model_t, int T, int D)
{
    __shared__ float Xs[Blocksize][Blocksize];
    __shared__ float Ys[Blocksize][Blocksize];

    int bx = blockIdx.x;
    int by = blockIdx.y;
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    int xBegin = bx * Blocksize * D;
    int yBegin = by * Blocksize * D;
    int yEnd = yBegin + D - 1;

    int x, y, k, o;
    float a1, a2, a3, a4, a5;
    float avgX, avgY, varX, varY, cov, rho;

    a1 = a2 = a3 = a4 = a5 = 0.0;

    for(y = yBegin, x = xBegin; y <= yEnd; y += Blocksize, x += Blocksize)
    {
        Xs[tx][ty] = d_model_t[x + ty * D + tx];
        Ys[ty][tx] = d_traces_t[y + ty * D + tx];

        __syncthreads();

        for(k = 0; k < Blocksize; k++)
        {
            a1 += Xs[k][tx];
            a2 += Ys[ty][k];
            a3 += Xs[k][tx] * Xs[k][tx];
            a4 += Ys[ty][k] * Ys[ty][k];
            a5 += Xs[k][tx] * Ys[ty][k];
        }

        __syncthreads();
    }

    avgX = a1 / D;
    avgY = a2 / D;

    varX = (a3 - avgX * avgX * D) / (D - 1);
    varY = (a4 - avgY * avgY * D) / (D - 1);
    cov = (a5 - avgX * avgY * D) / (D - 1);

    rho = cov / sqrtf(varX * varY);
    o = bx * Blocksize * T + tx * T + by * Blocksize + ty;

    d_corr[o] = rho;
}
```

Listing 4.4: Computation of the correlation matrix for a first order CPA attack based on [CDOR09].

The basic idea of the algorithm is illustrated in Figure 4.2.

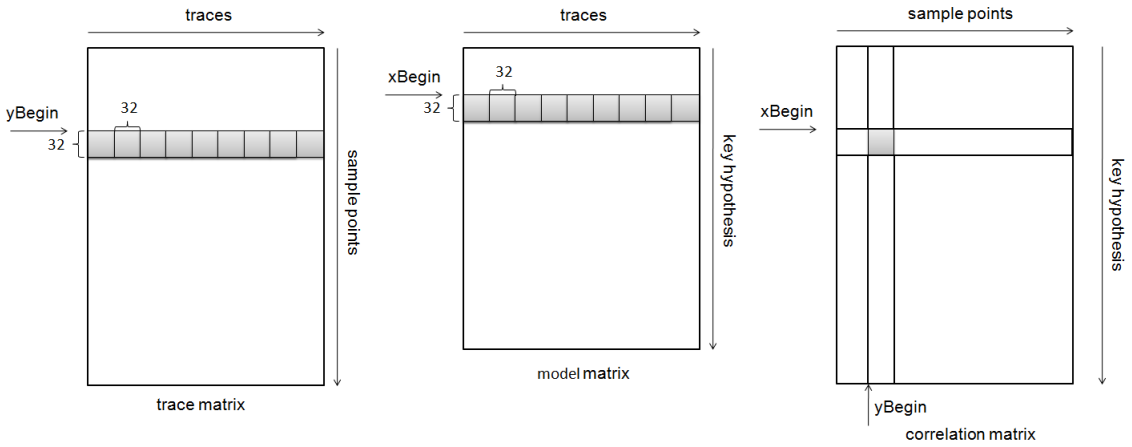

Figure 4.2: The basic idea of Listing 4.4 with a block size of 32×32 threads.

The grid consists of $\left\lceil \frac{K}{Blocksize} \right\rceil$ many blocks in the x-dimension and $\left\lceil \frac{T}{Blocksize} \right\rceil$ many blocks in the y-dimension, where K denotes the number of key hypothesis and T the number of sample points per trace. Each block of the grid computes one block of the correlation matrix. More precisely, each thread computes one entry of the correlation matrix. For this, each block in the x-dimension respectively the y-dimension processes one `Blocksize` wide column of the leakage-model matrix respectively the trace matrix as illustrated by Figure 4.2.
First of all, the offsets `xBegin` and `yBegin` are calculated based on the value of `Blocksize` and the block identifier. The outer "for" loop iterates over all blocks within each `Blocksize` wide row of the leakage-model matrix respectively the trace matrix addressed by `x` respectively `y`. In each iteration, the currently processed block is copied into shared memory. The array `Xs`, which resides in shared memory, is the transpose of its image in the leakage-model matrix. This transposition reduces the number of shared-memory bank conflicts, see Section 2.1.3. The $\_\_synchthreads()$ calls in line `23` and line `33` are vital in order to avoid race conditions, see Section 2.1.3.
Next, the elements of each block in shared memory are summed up in the inner "for" loop in order to compute the values `a1` to `a5`. Those values are required to compute the means, the variances, and the covariance of each row of the leakage-model matrix respectively the trace matrix. Subsequently, the Pearson correlation coefficient between each row of both matrices is computed. Finally, the result is stored in the correlation matrix as shown in Figure 4.2.

After the correlation matrix is computed, we copy it back to the CPU by using the CUDA API `cudaMemcpy` in order to be able to do further analysis. The analysis can, for instance, contain a search for the maximum correlation coefficient to extract the correct key hypothesis k_{ck}.

Additionally, we deallocate all allocated memory by the CUDA API

```
cudaError_t cudaFree(void* devPtr)
```

`cudaFree` frees the memory space pointed to by `devPtr`, which must have been returned by a previous call to `cudaMalloc` or `cudaMallocPitch`. Otherwise, or if `cudaFree(devPtr)` has already been called before, an error is returned. [api]

In a last step, we reset the device by the CUDA API

```
cudaError_t cudaDeviceReset(void)
```

`cudaDeviceReset` destroys all allocations and resets all state on the current device in the current process. [api]

4.1.2 Iterative Implementation

For the native implementation, we assume that all processed traces fit into the global memory of the device. This assumption is only reasonable up to a certain size of the trace pull. There are three parameters that influence the amount of required global memory, (i) the number of traces, (ii) the number of sample points per trace, and (iii) the number of key hypothesis.
The number of key hypothesis is constant and equal to 256 for the Hamming-weight model as we attack one key byte at once. Therefore, it is not adequate to split the attack in this dimension. However, for other power models this may become relevant.
Splitting the attack in the dimension of the sample points is trivial since they are independent from each other. Let the trace matrix be a $D \times T$ matrix, where D is the number of traces and T denotes the number of sample points per trace. We split this matrix in n submatrices of size $D \times \frac{T}{n}$ and compute the correlation for each of those submatrices. As a result, we obtain n correlation matrices of size $K \times \frac{T}{n}$, where K denotes the number of key hypothesis. To obtain the original $K \times T$ correlation matrix, we merge the n submatrices by appending them to each other in the appropriate order.
Dividing the attack in the dimension of the traces is not as straightforward. Let the trace matrix be a $D \times T$ matrix. We split this matrix in m submatrices of size $\frac{D}{m} \times T$ and compute the values $CS_{2,T}$, $CS_{2,L}$, μ_T, μ_L, and ACS_1 for each submatrix. To be able to compute the correlation, we merge the values of $CS_{2,T}$, $CS_{2,L}$, μ_T, μ_L, and ACS_1 for each of the m submatrices by applying the iterative formulas introduced in Section 2.2.1.
Obviously, the best way to handle the memory limitations of the GPU is to use both approaches. Thereby, we are capable of processing arbitrary large trace pulls, which consists of either a large number of traces, a large number of sample points per trace, or both.

From a programming point of view, we adapt the function that reads the traces to read $\frac{T}{n}$ many sample points per trace respectively to read $\frac{D}{m}$ many traces. Furthermore, we modify the function that reads the plaintexts to read $\frac{D}{m}$ many plaintexts. Apart from that, we add an outer "for" loop inside the main function that iterates from 0 to $n-1$. For the second approach, we additionally add an inner "for" loop, which iterates from 0 to $m-1$. Thereby, we process one subset of the trace pull in the dimension of the sample points in each iteration of the outer "for" loop and one subset in the dimension of the traces in each iteration of the inner "for" loop. Finally, we implement functions to handle the iterative formulas and adjust the correlation kernel.
The basic idea of the iterative implementation is as follows. We allocate two arrays of size T, which denotes the number of sample points per trace for $CS_{2,T}$, $CS_{2,L}$, μ_T, μ_L, and ACS_1. For the first iteration of the inner "for" loop, we compute the values of those parameters and store them in the first array, while the result of the computation is stored in the second array for every continuing iteration. At the end of each iteration — except for the first one — of the inner "for" loop, we compute the values of the parameters for the unified set by the iterative formulas and store the results in the first array.
To compute the sums and means, we use a slightly modified version of the algorithm shown in Listing 4.4. The only noteworthy modification is the omission of the division by $D-1$ in the lines 38 to 40. The computation of the unified set is accomplished as illustrated in Listing 4.5 and Listing 4.6.

```
1  __global__ void merge_sums(float *d_centr_sum_t_q1, float *d_centr_sum_l_q1, float *d_mean_t_q1,
2                             float *d_mean_l_q1, float *d_centr_sum_t_q2, float *d_centr_sum_l_q2,
3                             float *d_mean_t_q2, float *d_mean_l_q2, int D, int T, int K,
4                             int iteration)
5  {
6      int tidx = blockIdx.x * blockDim.x + threadIdx.x;
7
8      float mean_t_q1, mean_l_q1, centr_sum_t_q1, centr_sum_l_q1;
9      float mean_t_q2, mean_l_q2, centr_sum_t_q2, centr_sum_l_q2;
10     float mean_t_q, mean_l_q, centr_sum_t_q, centr_sum_l_q;
11     float delta_l, delta_t, delta_n_l, delta_n_t;
12
13     int n1, n2, n;
14
15     n1 = D * iteration;
16     n2 = D;
17     n = n1 + n2;
18
19     if(tidx < K)
20     {
21         centr_sum_l_q1 = d_centr_sum_l_q1[tidx];
22         centr_sum_l_q2 = d_centr_sum_l_q2[tidx];
23
24         mean_l_q1 = d_mean_l_q1[tidx];
25         mean_l_q2 = d_mean_l_q2[tidx];
26
27         delta_l = mean_l_q2 - mean_l_q1;
28         delta_n_l = delta_l / n;
29
30         centr_sum_l_q = centr_sum_l_q1 + centr_sum_l_q2 + n1 * n2 * delta_l * delta_n_l;
```

```
31          d_centr_sum_l_q1[tidx] = centr_sum_l_q;
32
33          mean_l_q = mean_l_q1 + n2 * delta_n_l;
34          d_mean_l_q1[tidx] = mean_l_q;
35      }
36
37      if(tidx < T)
38      {
39          centr_sum_t_q1 = d_centr_sum_t_q1[tidx];
40          centr_sum_t_q2 = d_centr_sum_t_q2[tidx];
41
42          mean_t_q1 = d_mean_t_q1[tidx];
43          mean_t_q2 = d_mean_t_q2[tidx];
44
45          delta_t = mean_t_q2 - mean_t_q1;
46          delta_n_t = delta_t / n;
47
48          centr_sum_t_q = centr_sum_t_q1 + centr_sum_t_q2 + n1 * n2 * delta_t * delta_n_t;
49          d_centr_sum_t_q1[tidx] = centr_sum_t_q;
50
51          mean_t_q = mean_t_q1 + n2 * delta_n_t;
52          d_mean_t_q1[tidx] = mean_t_q;
53      }
54  }
```

Listing 4.5: Computation of the unified set for $CS_{2,L}$, $CS_{2,T}$, μ_L, and μ_T.

In contrast to the previous algorithms, we choose to use a one-dimensional grid with one-dimensional blocks as the execution configuration. This is because we work on vectors of size T for the parameters $CS_{2,T}$ and μ_T respectively of size K for $CS_{2,L}$ and μ_L, where T is the number of sample points and K denotes the number of key hypothesis. Therefore, we invoke the kernel with t threads per block and $\left\lceil \frac{T}{t} \right\rceil$ blocks in the x-dimension.
First, we compute the global thread identifier of each thread inside the grid in line `5` like in the previous algorithms. The values `n1` and `n2` represent the size of each subset, where `D` denotes the number of processed traces per iteration of the inner "for" loop of the main function. Thus, the first set, which holds the result of the merging, gets larger by `D` for each kernel call.
Subsequently, we ensure that only those threads with an index `tidx` smaller than K compute $CS_{2,L}$ and μ_L. Moreover, we check whether `tidx` is smaller than T to prevent out-of-bound addressing. The remaining part of the algorithm handles the computation of the parameters for the unified set by the iterative formulas introduced in Section 2.2.1. The precomputation of `delta_n_t` and `delta_n_l` is performed for optimization purposes. Thereby, we save two divisions by `n` as those values occur twice in the computation of the sums and means.

```
__global__ void merge_adj_sum(float *d_adj_centr_sum_q1, float *d_mean_l_q1, float *d_mean_t_q1,
                              float *d_adj_centr_sum_q2, float *d_mean_l_q2, float *d_mean_t_q2,
                              int D, int T, int K, int iteration)
{
    int tidx = blockIdx.x * blockDim.x + threadIdx.x;

    float mean_t_q1, mean_l_q1, adj_centr_sum_q1;
    float mean_t_q2, mean_l_q2, adj_centr_sum_q2;
    float adj_centr_sum_q;
    float delta_t, delta_l;

    int n1, n2, n, index;

    if(tidx < T)
    {
        n1 = D * iteration;
        n2 = D;
        n = n1 + n2;

        for(int k = 0; k < K; k++)
        {
            index = k * T + tidx;

            mean_t_q1 = d_mean_t_q1[tidx];
            mean_t_q2 = d_mean_t_q2[tidx];

            mean_l_q1 = d_mean_l_q1[k];
            mean_l_q2 = d_mean_l_q2[k];

            adj_centr_sum_q1 = d_adj_centr_sum_q1[index];
            adj_centr_sum_q2 = d_adj_centr_sum_q2[index];

            delta_l = mean_l_q2 - mean_l_q1;
            delta_t = mean_t_q2 - mean_t_q1;

            adj_centr_sum_q = adj_centr_sum_q1 + adj_centr_sum_q2 + ((n1 * n2) / n) * delta_t *
                              delta_l;
            d_adj_centr_sum_q1[index] = adj_centr_sum_q;
        }
    }
}
```

Listing 4.6: Computation of the unified set for ACS_1.

We invoke the kernel with the same number of threads per block and blocks per grid as in Listing 4.5. Because the parameter ACS_1 is a $K \times T$ matrix rather than a vector of size T respectively K, we iterate over each row of the matrix in the "for" loop and process each column in parallel. The precomputation of `index` is performed for optimization purposes as it occurs three times inside the "for" loop.

Finally, Listing 4.7 shows the computation of the correlation matrix.

```
__global__ void correlation(float *d_corr, float *d_centr_sum_l, float *d_centr_sum_t,
                            float *d_adj_centr_sum, int T, int K)
{
    int tidx = blockIdx.x * blockDim.x + threadIdx.x;
    int tidy = blockIdx.y * blockDim.y + threadIdx.y;

    float adj_centr_sum, centr_sum_l, centr_sum_t, rho;

    int index;

    if(tidx < T && tidy < K)
    {
        index = tidy * T + tidx;

        centr_sum_t = d_centr_sum_t[tidx];
        centr_sum_l = d_centr_sum_l[tidy];
        adj_centr_sum = d_adj_centr_sum[index];

        rho = adj_centr_sum / sqrt(centr_sum_l * centr_sum_t);
        d_corr[index] = rho;
  }
}
```

Listing 4.7: Computation of the correlation matrix for the iterative implementation.

The kernel is launched with $t \times t$ threads per block and $\left\lceil \frac{T}{t} \right\rceil$ blocks in the x-dimension and $\left\lceil \frac{K}{t} \right\rceil$ blocks in the y-dimension. The choice of t has a crucial influence on the runtime of the kernel, see Section 4.1.3.

4.1.3 Optimizations

In this section, we provide hints on how we optimized the previously discussed implementations with regards to runtime. Moreover, the discussed procedure can be viewed as a general approach to enhance the performance of CUDA kernels. We use the following metrics[1] [CGM14] to illustrate the effect of our optimizations on the arithmetic effectiveness and the access to memory, and moreover, their effect on the runtime.

- gld_efficiency (GLE): Ratio of requested memory load throughput to the required global memory load throughput
- gst_efficiency (GSE): Ratio of requested memory store throughput to the required global memory store throughput
- gld_throughput (GLT): Global memory load throughput
- gst_throughput (GST): Global memory store throughput

[1] The metrics are measured with *nvprof*, a profiling application included in the CUDA toolkit.

- shared_load_transactions_per_request (SLTPR): Number of transactions per shared memory load request
- shared_store_transactions_per_request (SSTPR): Number of transactions per shared memory store request
- achieved_occupancy (AO): Ratio of average active warps per cycle to the maximum number of warps on a SM

Most metrics are memory related since the performance of most CUDA kernels is limited by the memory rather than by the arithmetic performance of the GPU. The impact of the optimizations is shown based on the code of Listing 4.4. However, the improvements are also applicable to the other kernels introduced in Section 4.1. As a measurement setup, we use 1000 traces with 100000 sample points per trace.

First, we set an optimal execution configuration for the kernel. The execution configuration includes the choice of the number of threads per block as well as the number of blocks per grid. Moreover, the dimensionality of the grid and the blocks belongs to the execution configuration. Table 4.1 shows the effect of various block sizes on the above mentioned metrics and the runtime.

block size	**GLE(%)**	**GSE(%)**	**GLT(GB/s)**	**GST(GB/s)**	**AO(%)**	**runtime(s)**
4 x 4	12.5	50	37.211	0.077	24.9	2.541
8 x 8	25	50	45.913	0.379	49.9	0.515
16 x 16	50	25	18.463	1.191	99.7	0.320
32 x 32	100	12.5	5.084	2.624	99.5	0.291

Table 4.1: Effect of various block sizes on the runtime of Listing 4.4.

Since the block size has no influence on the shared memory, the metrics `SLTPR` and `SSTPR` do not occur in the table. The most tremendous drop of the runtime appears between the first and the second row. While `GSE` remains constant, `GLE` as well as `AO` doubles. The metric `GLT` increases by nearly 25% and `GST` by nearly 500%. Although `GSE` and `GLT` decrease between the second and the third row, `GLE` and `AO` double again. This, in conjunction with the increase of `GST` by over 300% further leads to a significant runtime improvement.
Based on the metrics behaviour, we conclude the following. `GLE` does not influence the runtime significantly as it doubles in every row, while the runtime between the third and the fourth row does not change significantly. The same holds for `GSE` since it remains constant respectively decreases while the runtime improves. Furthermore, `GLT` does not have a great impact on the runtime. Although it decreases by roughly 60% between the second and the third row, the runtime gets better.
`GST` and `AO` effect the runtime most. Between the first and the second row, `GST` is five times higher and `AO` doubles, which leads to a five times better runtime. While `AO` doubles again between the second and the third row, `GST` increases by three times. The runtime

improves again but not as much as the first time. Between the third and the fourth row, `AO` remains the same while `GST` increases by 220%, which further leads to a minor optimization of the runtime.

Next, we optimize the accesses to shared memory. In line 3 and 4 of Listing 4.4, we declare two arrays of size $Blocksize \times Blocksize$ in shared memory. As mentioned in Section 2.1.3, padding reduces shared-memory bank conflicts. By adapting the number of columns of `Xs` and `Ys` from `Blocksize` to `Blocksize+2`, we significantly reduce the metrics `SLTPR` and `SSTPR`, see Table 4.2. Further improvements are achieved by configuring the shared memory bank size from 4 byte to 8 byte.

	SLTPR	**SSTPR**	**GLT(GB/s)**	**GST(GB/s)**	**runtime(ms)**
no padding	1.2	8.5	5.084	2.624	291.02
with padding	1.0	1.5	6.261	3.232	236.48
padding + config	1.0	1.0	6.371	3.288	232.25

Table 4.2: Reduction of shared memory bank conflicts for Listing 4.4.

The padding reduces `SLTPR` from 1.2 to 1.0 and `SSTPR` from 8.5 to 1.5, which positively effects the runtime. Moreover, configuring the shared memory to operate with 8 byte wide banks finally leads to conflict-free store operations. Interestingly, the metrics `GLT` and `GST` increase if `SLTPR` and `SSTPR` decrease. Since the data is loaded from global memory into shared memory and stored vice-versa, conflict-free accesses to shared memory lead to better access patterns on the global memory.

Further improvements of the access patterns to global memory are accomplished by enabling the L1 cache through the compiler flag `-Xptxas -dlcm=ca`, see Table 4.3.

	GLE(%)	**GST(%)**	**GLT(GB/s)**	**GST(GB/s)**	**runtime(ms)**
L1 cache disabled	100	12.5	6.371	3.288	232.25
L1 cache enabled	25	12.5	25.791	3.328	229.56

Table 4.3: Effect of enabling the L1 cache on Listing 4.4.

Although `GLE` decreases from 100% to 25%, `GLT` increases by approximately four times. This, in conjunction with the minor improvement of `GST` result in a slightly better runtime.

By applying all discussed optimizations, we decrease the overall GPU runtime for Listing 4.4 from 2.541 seconds to 0.230 seconds - a speed-up of approximately 11.

4.2 Higher Order CPA

We introduced formulas for robust and one-pass parallel computation of correlation-based attacks at arbitrary order [SMG15] in Section 2.2.1. In this section, we discuss the

implementation of those formulas on GPUs by the example of a second order attack. The extension to higher orders is straightforward. However, the limiting factor is the global memory since the required memory increases linearly with the order.

4.2.1 Native Implementation

The basic idea of the native implementation is as follows. In each iteration of the outer "for" loop of the main function, which handles one subset of the trace matrix in the dimension of the sample points, we initialize the arrays for $CS_{d,T}$, $CS_{2,L}$, μ_T, μ_L, and ACS_d with zeroes by the CUDA API

```
cudaError_t cudaMemset(void* devPtr, int value, size_t count)
```

`cudaMemset` fills the first `count` bytes of the memory area pointed to by `devPtr` with the constant byte `value`. [api]

This initialization is necessary because the incremental formulas require an empty set at the start, see Section 2.2.1. For every iteration of the inner "for" loop of the main function, which handles one subset of the trace matrix in the dimension of the traces, we read the corresponding plaintexts and traces, copy them to the device, and create the leakage-model matrix. Next, we compute $CS_{2,L}$ and μ_L by the incremental formulas as shown in Listing 4.8.

```
__global__ void compute_cs_l(float *d_mean_l, float *d_centr_sum_l2, uint8_t *d_model, int D,
                             int K, int iteration)
{
    int tidx = blockIdx.x * blockDim.x + threadIdx.x;

    float value_l, delta_l, delta_n_l;
    float n, n11;

    int index1, index2;

    n = D * iteration;

    d_mean_l[tidx] = d_mean_l[D * K + tidx];

    for(int i = 0; i < D; i++)
    {
        n++;
        n11 = n - 1;

        index1 = i * K + tidx;
        index2 = (i + 1) * K + tidx;

        value_l = d_model[index1];

        delta_l = value_l - d_mean_l[index1];
        delta_n_l = delta_l / n;

        d_mean_l[index2] = d_mean_l[index1] + delta_n_l;
        d_centr_sum_l2[tidx] += delta_n_l * delta_l * n11;
```

```
30      }
31  }
```

Listing 4.8: Computation of $CS_{2,L}$ and μ_L for the native higher order CPA implementation.

The kernel is invoked with one block of K threads in the x-dimension since the algorithm computes $CS_{2,L}$ and μ_L in parallel in the dimension of the key hypothesis. The array `d_mean_l` is a $(D+1) \times K$ matrix, where D is the number of traces processed in each iteration of the inner "for" loop of the main function and K denotes the number of key hypothesis. The variable n represents the number of so far processed traces.
In every iteration of the "for" loop in line `15`, the algorithm computes the current value of μ_L indexed by `index2` based on its previous value indexed by `index1` in line `28`. Therefore, `d_mean_l` acts as a lookup table for Listing 4.9. Additionally, $CS_{2,L}$ is computed in line `29`. Those computations are performed in parallel in the dimension of the key hypothesis by each thread with the index `tidx`.
The assignment in line `13` of the algorithm has the following purpose. At the end of the "for" loop, `index2` points to the $(D+1)$th row of `d_mean_l`. This row holds the value of μ_L for the first iteration of the "for" loop for the next call of the kernel. Thus, if we call the kernel again to process the next set of traces, we copy the $(D+1)$th row of the matrix into the first row and continue to compute the further mean values.

Subsequently, we compute $CS_{d,T}$, μ_T, and ACS_d as illustrated in Listing 4.9.

```
1  __global__ void sums2(float *d_adj_centr_sum1, float *d_adj_centr_sum2, float *d_centr_sum_t2,
2                        float *d_centr_sum_t3, float *d_centr_sum_t4, float *d_centr_sum_l2,
3                        float *d_mean_t, float *d_mean_l, int8_t *d_traces, uint8_t *d_model,
4                        int D, int T, int K, int iteration)
5  {
6      int tidx = blockIdx.x * blockDim.x + threadIdx.x;
7
8      float value_l, value_t;
9      float delta_l, delta_t, delta_n_l, delta_n_t, delta_n_t2, delta_n_t3, delta_n_t4;
10     float cs_t2, cs_t3, cs_t4, acs1, acs2, mean_t;
11     float n, n11, n12, n13;
12
13     int index1, index2;
14
15     if(tidx < T)
16     {
17         n = D * iteration;
18
19         cs_t2 = d_centr_sum_t2[tidx];
20         cs_t3 = d_centr_sum_t3[tidx];
21         cs_t4 = d_centr_sum_t4[tidx];
22
23         mean_t = d_mean_t[tidx];
24
25         for(int i = 0; i < D; i++)
26         {
27             n++;
28             n11 = n - 1;
29             n12 = n11 * n11;
```

```
30            n13 = n12 * n11;
31
32            value_t = d_traces[i * T + tidx];
33
34            delta_t = value_t - mean_t;
35            delta_n_t = delta_t / n;
36            delta_n_t2 = delta_n_t * delta_n_t;
37            delta_n_t3 = delta_n_t2 * delta_n_t;
38            delta_n_t4 = delta_n_t2 * delta_n_t2;
39
40            for(int k = 0; k < K; k++)
41            {
42                index1 = i * K + k;
43                index2 = k * T + tidx;
44
45                value_l = d_model[index1];
46                delta_l = value_l - d_mean_l[index1];
47                delta_n_l = delta_l / n;
48
49                acs2 = d_adj_centr_sum2[index2];
50                acs1 = d_adj_centr_sum1[index2];
51
52                acs2 += (-delta_n_l * cs_t2) - (2 * delta_n_t * acs1)
53                        + ((n13 - n + 1) * delta_n_t2 * delta_n_l);
54                acs1 += n11 * delta_n_t * delta_l;
55
56                d_adj_centr_sum2[index2] = acs2;
57                d_adj_centr_sum1[index2] = acs1;
58            }
59
60            cs_t4 += (-4 * delta_n_t * cs_t3) + ( 6 * delta_n_t2 * cs_t2)
61                     + (delta_n_t4 * n11 * (n13 + 1));
62            cs_t3 += (-3 * delta_n_t * cs_t2) + (delta_n_t3 * n11 * (n12 - 1));
63            cs_t2 += delta_n_t * delta_t * n11;
64
65            mean_t += delta_n_t;
66        }
67
68        d_centr_sum_t4[tidx] = cs_t4;
69        d_centr_sum_t3[tidx] = cs_t3;
70        d_centr_sum_t2[tidx] = cs_t2;
71
72        d_mean_t[tidx] = mean_t;
73    }
74 }
```

Listing 4.9: Computation of $CS_{d,T}$, μ_T, and ACS_d for a second order attack.

We invoke the kernel with $\left\lceil \frac{T}{t} \right\rceil$ blocks in the x-dimension, where T is the number of sample points and each block consists of t threads.
In the first instance, we check if the thread index `tidx` does not exceed T to avoid out-of-bound addressing. The variable n holds the number of traces processed so far. The values of $CS_{2,T}$, $CS_{3,T}$, $CS_{4,T}$, and μ_T from the previous kernel call are loaded from the global memory into registers in the lines `19` to `23`. For the first call of the kernel, those values are zero due the initialization.
All computations are performed in parallel in the dimension of the sample points by each thread with the index `tidx`. In the outer "for" loop in line `25`, the algorithm iterates

over each trace, loads the current value of the trace matrix and compute Δ_t. The precomputation of `n11` to `n13` as well as `delta_n_t` to `delta_n_t4` are for optimization purposes as they occur multiple times in the formulas.
In the inner "for" loop in line `40`, the algorithm iterates over all key hypothesis since ACS_d is a $K \times T$ matrix, see Listing 4.6. In each iteration, the algorithm loads the current value of the leakage-model matrix and computes Δ_l. For this, the previously computed array `d_mean_l` by Listing 4.8 is used as a lookup table. Furthermore, the values of ACS_1 and ACS_2 are updated in the lines `52` to `54` based on their previous values, which are loaded into registers in the lines `49` to `50`. After the inner "for" loop has finished its work, the values of $CS_{2,T}$, $CS_{3,T}$, $CS_{4,T}$, and μ_L are updated.

In the final step, we compute the correlation matrix. We omit to list the algorithm since its structure is very similar to that of Listing 4.7.

4.2.2 Optimized Implementation

The native implementation is parallelized only in the dimension of the sample points, while the traces are processed in an iterative manner. However, it is possible to additionally parallelize in the dimension of the traces. If there are D traces processed per iteration of the inner "for" loop of the main function and we spawn t threads in the y-dimension of the grid, every thread processes $\frac{D}{t}$ traces and there are t traces worked off in parallel.
This approach allows a fine grained distribution of the parallelism, which increases the performance significantly. Furthermore, the implementation becomes more flexible with regards to the layout of the trace pull. If the trace pull consists of many traces, where each trace has a small number of sample points, the native implementation is not efficient. This is due to the lack of a large number of sample points inhibits the parallelism.
The logical structure of the optimized implementation is similar to that of the native one. However, we have to merge the subsets at the end by using the iterative robust one-pass formulas, similar to the implementation discussed in Section 4.1.2.
We omit to list the merging algorithms as their logical structure is very similar to that of Listing 4.5 and Listing 4.6. If we configure our implementation to work on t traces in parallel, we obtain t subsets that have to be merged. This is accomplished by an additional "for" loop that iterates from $k = 1$ to $t - 1$, where the first subset is merged with the $(k + 1)$th subset in each iteration. Handling the parallelism in the dimension of the traces is shown by the example of Listing 4.10.

```
1  __global__ void compute_cs_l(float *d_mean_l, float *d_centr_sum_l2, uint8_t *d_model,
2                               int D, int K, int iteration)
3  {
4      int tidx = blockIdx.x * blockDim.x + threadIdx.x;
5      int bidy = blockIdx.y;
6
7      float value_l, delta_l, delta_n_l;
8      float n, n11;
9
10     int index1, index2, offset, subset;
11
12     n = (D / parallelTraces) * iteration;
13
14     subset = D / parallelTraces;
15     offset = bidy * subset;
16
17     d_mean_l[offset * K + tidx] = d_mean_l[(D + bidy) * K + tidx];
18
19     for(int i = 0; i < subset; i++)
20     {
21         n++;
22         n11 = n - 1;
23
24         index1 = (offset + i) * K + tidx;
25         index2 = (offset + i + 1) * K + tidx;
26
27         if(i == subset - 1)
28         {
29             index2 = (D + bidy) * K + tidx;
30         }
31
32         value_l = d_model[index1];
33
34         delta_l = value_l - d_mean_l[index1];
35         delta_n_l = delta_l / n;
36
37         d_mean_l[index2] = d_mean_l[index1] + delta_n_l;
38         d_centr_sum_l2[(bidy * K) + tidx] += delta_n_l * delta_l * n11;
39     }
40 }
```

Listing 4.10: Computation of $CS_{2,L}$ and μ_L for the optimized implementation.

In contrast to Listing 4.8, we additionally set b blocks in the y-dimension of the grid, where b is the number of traces to work on in parallel. The choice of b has a large influence on the performance, see Section 4.2.3. If we process b traces in parallel, each subset has the size $\frac{D}{b}$ for each iteration of the inner "for" loop of the main function. This size is assigned to the variable `subset`, which is used as a bound for the "for" loop in line `19`. Additionally, the variable `n` holds the number of processed traces so far for each subset, which is $\frac{D}{b} \cdot k$ for the kth iteration of the inner "for" loop of the main function. The variable `offset` denotes the number of the first row of the corresponding subset.
The rest of the algorithm is very similar to Listing 4.8. The array `d_mean_l` is a $(D+b) \times K$ matrix. The rows $D+1$ to $D+b$ function as buffers to store the value of μ_L for each subset for the first iteration of the "for" loop for the next call of the kernel, similar to Listing 4.8. Since we work on multiple subsets, `index2` has to point to the corresponding

buffer row of `d_mean_l` in the last iteration of the "for" loop in line `29`.

4.2.3 Optimizations

The structure of this section is similar to that of Section 4.1.3. For the measurements, which are conducted by the example of Listing 4.9, we use a trace pull of 1000 traces with 100000 sample points per trace.
First, we set an optimal execution configuration for the kernel. Table 4.4 shows the effect of different block sizes on the metrics and the runtime.

Blocksize	**GLE(%)**	**GSE(%)**	**GLT(GB/s)**	**GST(GB/s)**	**AO(%)**	**runtime(s)**
32	81.57	100	36.638	29.299	-	6.512
64	81.57	100	61.779	49.404	49.67	3.852
128	81.57	100	61.457	49.147	99.38	3.855
256	81.57	100	65.732	52.565	98.46	3.552
512	81.57	100	66.510	53.188	99.29	3.518
1024	81.57	100	66.042	52.813	98.37	3.623

Table 4.4: The Effect of different block sizes on Listing 4.9.

The metrics `GLE` and `GSE` have no influence on the runtime since their values are the same for every choice of the block size. Moreover, `AO` does not effect the runtime as it doubles between the second and the third row, while the runtime does not alter. The metrics `GLT` and `GST` significantly increase between the first and the second row, which leads to a considerably runtime improvement. Furthermore, they increase slightly between the third and fourth row, which has a minor effect on the runtime.
In a final conclusion, a block size larger than 128 should be used, where a block size of $t = 512$ threads is the optimal execution configuration for the kernel. The metric `AO` could not be measured for $t = 32$ threads because for an unknown reason, nvprof reports that an overflow occurs.

As mentioned in Section 4.1.3, the performance can be enhanced further by enabling the L1 cache through the compiler flag `-Xptxas -dlcm=ca`. However, in the case of Listing 4.9, the runtime becomes worse by enabling the L1 cache as shown in Table 4.5.

	GLE(%)	**GSE(%)**	**GLT(GB/s)**	**GST(GB/s)**	**AO(%)**	**runtime(s)**
L1 cache off	81.57	100	66.510	53.188	99.29	3.518
L1 cache on	50.95	100	98.673	49.289	94.83	3.861

Table 4.5: The Effect of enabling the L1 cache on Listing 4.9.

Although the metric `GLT` increases significantly, `GLE`, `GST`, and `AO` drop, which overall leads to a worse runtime. From this, we conclude that the three metrics that decrease have a higher impact on the runtime than the metric `GLT`.

Optimizations with regards to shared memory are not an issue since none of the kernels of the higher order implementations use any shared memory. However, we can further decrease the runtime by the choice of an optimal number of sample points processed per iteration of the outer "for" loop of the main function, see Table 4.6.

sample points per iteration	**iterations**	**number of sample points**	**runtime(s)**
100000	1	100000	3.518
50000	2	100000	3.247
33333	3	99999	4.902
25000	4	100000	3.879
20000	5	100000	2.849
16666	6	99996	3.995
12500	8	100000	4.280
10000	10	100000	4.874

Table 4.6: Effect of different number of sample points per iteration on Listing 4.9.

The reference configuration is 100000 sample points, which are processed in one iteration. The best configuration, which is 20000 processed sample points in five iterations, yield a speed-up of approximately 19%. For a number of processed sample points different to 100000, the speed-up can be even higher. There is no recognizable pattern on which we can conclude how to choose an optimal configuration. The differences of overall processed sample points to 100000 in row three and six are negligible and should have no noticeable impact on the measurement.

By applying all discussed optimizations, we decrease the overall GPU runtime of Listing 4.9 from 6.512 seconds to 2.849 seconds, a speed-up of approximately 2.3. It is noteworthy that all statements are only valid for a second order CPA attack. For a third order attack, the performance factor is 1.62. Interestingly, the best configuration is achieved by using 64 threads per block, enabling the L1 cache, and processing 100000 sample points in one iteration. As we have to load more values from the global memory in a third order attack, enabling the L1 cache is advantageous.

As a conclusion of this section, we provide an overview of various distributions of the parallelism in the dimension of parallel processed traces and sample points for the optimized implementation of Listing 4.9. The different configurations are shown in Figure 4.3.

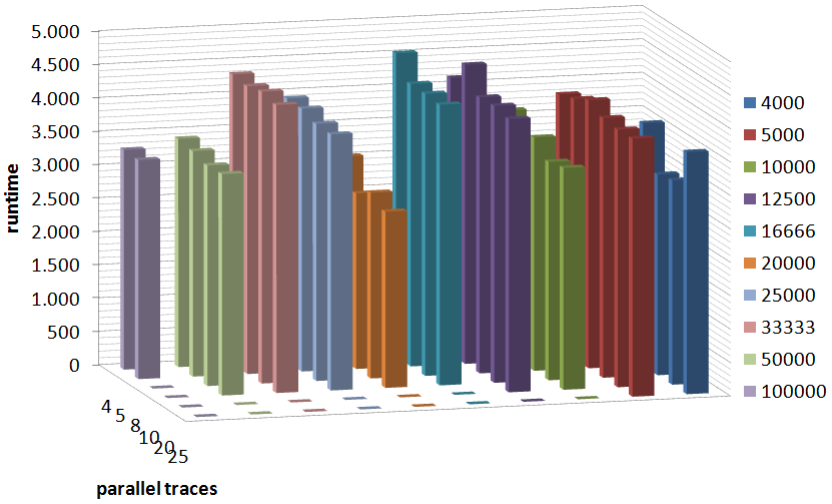

Figure 4.3: Illustration of the distribution of parallelism for the optimized implemenation of Listing 4.9.

For each number of processed sample points per iteration, we list the possible number of parallel processed traces. It is, for instance, not possible to process more than five traces in parallel, if we process 100000 traces per iteration because the GPU runs out of memory. As a reference value, we take the best configuration of Table 4.6, which results in a runtime of 2.849 seconds. The best configuration for the optimized implementation of Listing 4.9 is 20000 sample points per iteration with five parallel processed traces, which results in runtime of 2.636 seconds - a speed-up of approximately 7.5%. Empirical measurements showed that applying the same approach for a third order attack yields to a speed-up of approximately 26.5%. Thus, the optimized implementation is capable of enhancing the performance of the attack significantly.
By evaluating Figure 4.3, we conclude the following. First, the choice of the processed sample points per iteration has a larger influence on the runtime than the number of parallel processed traces. The runtimes for 12500 processed sample points are worse than for 50000 sample points, regardless of the number of parallel processed traces. Secondly, within each choice of processed sample points, the runtime tend to get better for a higher the number of parallel processed traces, at least up to a certain point. Beyond this point, the additional computational costs for merging the subsets is higher than the performance gain achieved through the additional parallelism.

4.3 Multiple GPU Support

In addition to the discussed optimizations, we can further enhance the performance of the attack through multiple GPU support, which we implemented for the in Section 4.1.2 and Section 4.2.2 discussed implementations. It is dispensable to implement that feature for the native versions as there is no argument to use them over the mentioned ones.

There are two approaches to distribute the attack among multiple GPUs. For the first one, each GPU processes a subset of the trace matrix. Let the trace matrix be a $D \times T$ matrix, where D is the number of traces and T the number of sample points per trace. Each GPU processes a submatrix of size $D \times \frac{T}{n}$, where n denotes the number of GPUs. By applying this approach, we have to modify every kernel since the submatrices differ in their size from the original $D \times T$ matrix. In general, it is possible to distribute the work among the number of traces so that each GPU processes a $\frac{D}{n} \times T$ submatrix. However, this approach is inappropriate since we have to merge the submatrices by the iterative robust one-pass formulas. Therefore, the additional computational effort effects the runtime negatively.
For the second approach, we distribute the number of iterations of the outer "for" loop of the main function among the GPUs. Let T be the number of sample points per trace and we process S sample points in each iteration. Thus, the number of iterations is $i = \frac{T}{S}$. Therefore, each GPU handles the computational effort of at most $m = \left\lceil \frac{i}{n} \right\rceil$ iterations, where n denotes the number of GPUs. We prefer this approach as there is no need to change any kernel.
Listing 4.11 demonstrates the second approach for the implementation discussed in Section 4.1.2 by a minimalistic main function in pseudo code.

```
1  int main(void)
2  {
3      int8_t *d_traces[numGPU], *d_traces_t[numGPU];
4      uint8_t *d_plaintexts[numGPU], *d_model[numGPU], *d_model_t[numGPU];
5      int iteration, i;
6
7      //more declarations
8
9      cudaStream_t stream[numGPU];
10     omp_set_num_threads(numGPU);
11
12     #pragma omp parallel
13     {
14         int thread = omp_get_thread_num();
15         cudaSetDevice(thread);
16
17         CHECK(cudaMallocHost(&d_traces[thread], ...));
18         CHECK(cudaMallocHost(&d_plaintexts[thread], ...));
19         CHECK(cudaMalloc(&d_model[thread], ...));
20         CHECK(cudaMalloc(&d_corr[thread], ...)));
21
22         //more allocations
23
24         cudaStreamCreate(&stream[thread]);
25     }
26
27     #pragma omp parallel
28     {
29         #pragma omp for private(iteration, block, grid, i)
30         for(int iterationDP = 0; iterationDP < numIterationsDP; iterationDP++)
31         {
```

```
32            int thread = omp_get_thread_num();
33            cudaSetDevice(thread);
34
35            for(iteration = 0; iteration < numIterations; iteration++)
36            {
37                read_plaintexts(Byte, iteration, thread);
38                read_traces(iteration, iterationDP, thread);
39
40                CHECK(cudaMemcpyAsync(d_plaintexts[thread], plaintexts[thread], ...));
41                CHECK(cudaMemcpyAsync(d_traces[thread], traces[thread], ...));
42
43                block.x = ...;
44                block.y = ...;
45                grid.x = ...;
46                grid.y = ...;
47
48                create_model<<<grid, block>>>(d_model[thread], d_plaintexts[thread]);
49
50                //more kernel calls
51            }
52
53            block.x = ...;
54            block.y = ...;
55            grid.x = ...;
56            grid.y = ...;
57
58            correlation<<<grid, block>>>(d_corr[thread], d_centr_sum_l_q1[thread], ...);
59            CHECK(cudaMemcpyAsync(corr[thread], d_corr[thread], ...));
60        }
61    }
62
63    //search for maximum correlation coefficient in correlation matrix
64
65    #pragma omp parallel
66    {
67        int thread = omp_get_thread_num();
68        cudaSetDevice(thread);
69
70        CHECK(cudaFreeHost(d_traces[thread]));
71        CHECK(cudaFreeHost(d_plaintexts[thread]));
72        CHECK(cudaFree(d_model[thread]));
73        CHECK(cudaFree(d_corr[thread]));
74
75        //more deallocations
76
77        CHECK(cudaDeviceReset());
78    }
79
80    return 0
81 }
```

Listing 4.11: Multiple GPU support for the implementation of Section 4.1.2.

In contrast to the single GPU version, we declare an array of `numGPU` many pointers and allocate `numGPU` many arrays for each parameter, where `numGPU` denotes the number of GPUs. Moreover, we create `numGPU` many `openmp` threads with `omp_set_num_threads(numGPU)`. Therefore, at least `numGPU` many logical CPU cores are required in order to distribute the attack among that many GPUs. Each thread is responsible to handle the necessary operations (allocation, data copy, kernel calls, deallocation) for one specific GPU. For

this, we create one stream for each GPU with the CUDA API `cudaStreamCreate`.
A CUDA stream refers to a sequence of asynchronous CUDA operations that execute on a device in the order issued by the host code. Because all operations queued in a CUDA stream are asynchronous, it is possible to overlap their execution with other operations in the host-device system. [CGM14]
In order to be able to overlap kernel calls and data transfers, we have to use the asynchronous version of `cudaMemcpy`, namely `cudaMemcpyAsync`. Furthermore,
`cudaMemcpyAsync` requires the data to be allocated by `cudaMallocHost` instead of `cudaMalloc` and deallocated by `cudaFreeHost` instead of `cudaFree`.
Memory allocations and deallocations are performed in parallel by each openmp thread. For this, we encapsulate the operations in a block labeled by `#pragma omp parallel`. Inside this block, every thread examines its thread number by calling `omp_get_thread_num()` and sets its assigned GPU through `cudaSetDevice`. After the device is set, every subsequent operation is executed with respect to this device. We parallelize the outer "for" loop in line `30` by the label `#pragma omp for private(iteration, block, grid, i)`. Thus, each thread and therefore each GPU processes $\frac{numIterationsDP}{numGPU}$ many iterations. The part `private(iteration, block, grid, i)` ensures that each thread works on its private variables `iteration`, `block`, `grid`, and `i`. Otherwise, the inner "for" loop in line `35` would not be run for `iteration` many iterations by each thread. The variable `i` is used to search for the maximum correlation coefficient, `grid` and `block` are used to set the execution configuration for each kernel.

4.4 Peak Extraction

The complexity of a CPA attack increases linearly with the number of power traces processed for the attack and with the number of sample points per trace. One goal of the attack should be to succeed with as few power traces and sample points as possible. An important fact is that peaks in power traces carry more information than flanks. Therefore, the attacker is able to reduce the computational effort of an attack by compressing the power traces, which can be accomplished by either peak extraction or integration. Integration means to sum up all data points of a clock cycle to a new substituting data point. Peak extraction is the extraction of the highest respectively the lowest point of each clock cycle to represent the information of this cycle [MOP08]. Peak extraction can be realized by using windows in the following way.

1. Start at some fixed sample point.
2. Find the first peak as the `maximum` point within the next `windowsize` many points.
3. Skip `stepsize` many sample points from the current position `pos`.
4. Find the next peak as the `minimum` point within the window $\left[pos - \frac{windowsize}{2}, pos + \frac{windowsize}{2}\right]$ around the current position `pos`.

5. Skip `stepsize` many sample points from the current position `pos`.

6. Find the next peak as the `maximum` point within the window $\left[pos - \frac{windowsize}{2}, pos + \frac{windowsize}{2}\right]$ around the current position `pos`.

7. Repeat Step 3 to 6 until the desired amount of peaks has been extracted.

An implementation of this algorithm is shown in Listing 4.12.

```
__global__ void peak_extraction(int8_t *d_traces_p, int8_t *d_traces, int T)
{
    int bidy = blockIdx.y;

    int8_t point;

    float alt = 0.0;
    int offset = 0;
    int ctr = 1;
    int peak = -128;

    for(int i = 0; i < windowsize; i++)
    {
        point = d_traces[bidy * T + i];

        if(point > peak)
        {
            peak = point;
        }
    }

    d_traces_p[bidy * (T / stepsize)] = peak;

    while((offset + windowsize / 2) < numPoints)
    {
        peak = 127 * powf(-1.0, alt);

        offset += stepsize;

        for(int j = -(windowsize / 2); j < (windowsize / 2); j++)
        {
            point = d_traces[bidy * T + offset + j];

            if(-(point * powf(-1.0, alt)) > -(peak * powf(-1.0, alt)))
            {
                peak = point;
            }
        }

        d_traces_p[(bidy * (T / stepsize)) + ctr] = peak;

        ctr++;
        alt++;
    }
}
```

Listing 4.12: A peak-extraction algorithm using windows.

We invoke the kernel with `numTraces` many blocks in the y-dimension, where each block consists of one thread. Therefore, one block performs the peak extraction for one trace. We start at the fixed sample point zero and the current position is stored in the variable `offset`. The second step of the algorithm is performed in the lines `12` to `20`. The third and fifth step is realized by incrementing `offset` by `stepsize` in line `28`. The repetition of the steps 3 to 6 is accomplished through a while loop in which we check whether or not `offset + windowsize/2` is smaller than the number of sample points per trace. If the condition fails, the whole trace has been processed. In line `26`, we assign the maximum respectively the minimum value of a byte to the variable `peak` to ensure that the first processed datapoint is smaller respectively larger than `peak`.
The steps 4 and 6 are realized inside the "for" loop in line `30`, which iterates from `-windowsize/2` to `windowsize/2` around the current position indicated by `offset`. We have to search for the highest respectively the lowest sample point indicated by `peak` in an alternating fashion. If the variable `alt` is odd, we search for the next maximum else we search for the next minimum. Initially, we set `peak` to the highest respectively the lowest possible value to ensure that the first sample point within the current window is lower respectively higher than `peak`.
If `alt` is odd, the term `-(point * powf(-1.0, alt)) > -(peak * powf(-1.0, alt))` is equal to `point > peak` since -1.0^{alt} is always -1.0 for odd values of `alt`. Similarly, `-(point * powf(-1.0, alt)) > -(peak * powf(-1.0, alt))` is equal to `-point > -peak = point < peak` for even values of `alt`.

4.5 Execution Configuration

In this section, we explain the different parameters used to configure the attacks on a given dataset. Thereby, we refer to the implementation discussed in Section 4.2.2 with multiple GPU support as well as support for peak extraction since it has the most complex execution configuration and the set of parameters for the other implementations is just a subset. Listing 4.13 represents an example configuration.

```
1  #define numTraces          1000
2  #define numIterations        10
3  #define parallelTraces       25
4  #define numPoints          4000
5  #define numIterationsDP      25
6  #define startPoint            0
7
8  #define TracesPerFile      1000
9  #define numFiles             10
10 #define step_t           100000
11 #define step_p            10000
12
13 #define key_hypothesis      256
14 #define Byte                  5
15 #define order                 3
16
17 #define Blocksize            32
```

```
18 #define Blocksize2          256
19 #define Blocksize3          512
20
21 #define numGPU                4
22
23 #define windowsize          100
24 #define stepsize             10
25 #define initialPoints     40000
```

Listing 4.13: Parameters of the execution configuration for the implementation of Section 4.2.2.

The parameter `numTraces` defines the number of processed traces per iteration, where the number of iterations is specified by the parameter `numIterations`. The overall number of processed traces is $D = numTraces \cdot numIterations$. The same holds for `numPoints` and `numIterationsDP` with respect to the sample points per trace T.
The number of parallel processed traces is specified by `parallelTraces`. The parameter `startPoint` denotes the first sample point of the trace that is considered in the attack. We use `startPoint` in conjunction with `numGPU`, which specifies the number of used GPUs, for scalability, see Section 4.6. If `numGPU` is set to a value larger than one, `numIterationsDP` should be set to a multiple of `numGPU` to evenly distribute the work among the GPUs.
The next four parameters are used to read the plaintexts and traces from the files on the hard drive and can be altered depending on the file format. The parameters `TracesPerFile` and `numFiles` specify the number of traces in each trace file and the overall number of trace files on the hard drive. The parameter `step_t` defines the number of sample points per trace in the trace file, while `step_p` specifies the number of plaintexts.
The number of key hypothesis is defined by `key_hypothesis` and is, in the case of the Hamming-weight model, equal to 256. `Byte` denotes the key byte that is extracted for one run of the attack. The order of the attack is defined by `order`. If we set `order` to d, the attack is performed for every order from 1 to d.
Furthermore, the parameters `Blocksize`, `Blocksize1`, and `Blocksize2` define the block sizes for the execution configuration of the various kernels.
The parameters `windowsize`, `stepsize`, and `initialPoints` are used for peak extraction, see Section 4.4. Since `numPoints` has to be set to $\frac{initialPoints}{stepsize}$, `stepsize` must be equal to 1, if no peak extraction shall be applied. In the example configuration, we initially use $initialPoints = 40000$ sample points per iteration and compress them by a factor of $stepsize = 10$. Therefore, $numPoints = \frac{initialPoints}{stepsize} = 4000$. If no peak extraction is applied, $initialPoints = numPoints$ since $stepsize = 1$.
For the implementations discussed in Section 4.1.1 and Section 4.1.2, `numPoints` must be a multiple of `blocksize`. Moreover, for the implementation of Section 4.1.1, `numTraces` must be a multiple of `blocksize`. For the implementation of Section 4.2.2, the parameter `parallelTraces` must be a divisor of `numTraces` and `stepsize` must be a divisor of `initialPoints`. The define `order` must be 1, 2, or 3 since the attack currently works

only for an order up to 3. The parameter `Blocksize` must be smaller than 32 and `Blocksize1` respectively `Blocksize2` must be smaller than 1024 since the maximum number of threads per block is restricted to 1024, at least for devices of compute capability up to 3.7, see Chapter 3.

4.6 Scalability

Scalability is a desirable attribute of a network, system, or process. The concept connotes the ability of a system to accommodate an increasing number of elements or objects, to process growing volumes of work gracefully, and/or to be susceptible to enlargement [Bon00].
We facilitate scalability by two aspects. First, it is possible to distribute the attack across an arbitrary large cluster of servers, where each server performs the attack on one subset of the trace pull. This is accomplished by the define `startPoint`, see Section 4.5.
Assuming, we have a trace pull of D traces and T sample points per trace and intend to evenly distribute a CPA attack among multiple servers with different computation capabilities. Initially, we perform the attack on each server on a subset of R traces and S sample points, where $R \ll D$ and $S \ll T$. By this, we measure the time consumption t_x to evaluate the performance of each server x in relation to the others. Once the performance is evaluated, we evenly distribute the work to the servers in a way that each server consumes the same time with respect to its subset of sample points by setting the defines `numPoints` and `startPoint` accordingly, see Section 4.5. Table 4.7 illustrates an example configuration of three servers performing a CPA attack with 10000 traces and 1 million sample points per trace.

Server	**Runtime t_x (s)**	**startPoint**	**numPoints**	**Runtime T_x (s)**
1	1.44	0	576000	82.944
2	2.88	576000	288000	82.944
3	6.09	864000	136000	82.824

Table 4.7: An example configuration to evenly distribute a CPA attack among multiple servers.

We assume that the sample set consists of $R = 1000$ traces with $S = 10000$ sample points per trace. It holds that $T_x = t_x \cdot \frac{numPoints}{S}$, where T_x denotes the overall runtime for the attack and t_x the runtime for the attack on the sample set for server x. The whole range of T sample points is covered, while each server has a runtime of approximately 83 seconds.
Secondly, a fine grained scalability is achieved through multiple GPU support, see Section 4.3. By this, we can further distribute the amount of work for each server among an arbitrary number of GPUs. However, it is not possible to evenly distribute the work among GPUs with different computation capabilities. For this, we have to implement the first approach, discussed in Section 4.3.

5 Results

This chapter discusses the performance of our implementations with respect to first and higher order CPA including a comparison to the performance of respective CPU implementations.

5.1 First Order CPA

To measure the performance of the implementation discussed in Section 4.1.2 with one respectively four GPUs, we use a trace pull that consists of 100000 traces with 1 million sample points per trace. The measurements are conducted with nvvp, a visual profiling tool included in the CUDA toolkit, on the Tesla K80, see Chapter 3. We denote the number of traces by D, the number of sample points per trace by T, and the number of overall processed sample points by $V = D \cdot T$. Figure 5.1 and Figure 5.2 show the runtime of the single GPU version for different numbers of sample points V.

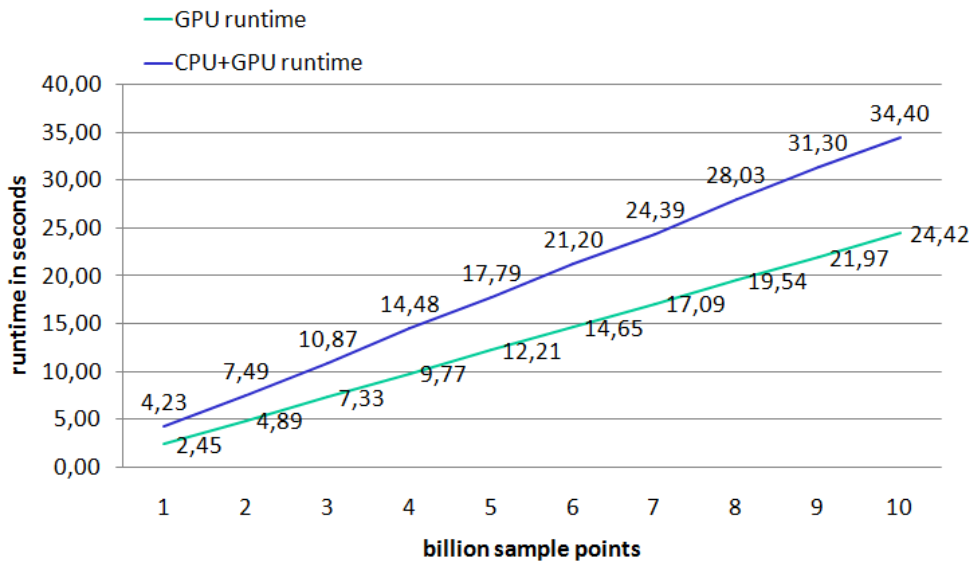

Figure 5.1: Runtime of the implementation from Section 4.1.2 using a single GPU for a constant number of sample points T and a variable number of traces D.

For the plot illustrated in Figure 5.1, we fixed T to 1 million and increased the number of traces D from 1000 to 10000 in steps of 1000. Therefore, the number of processed sample points V ranges from 1 to 10 billion.

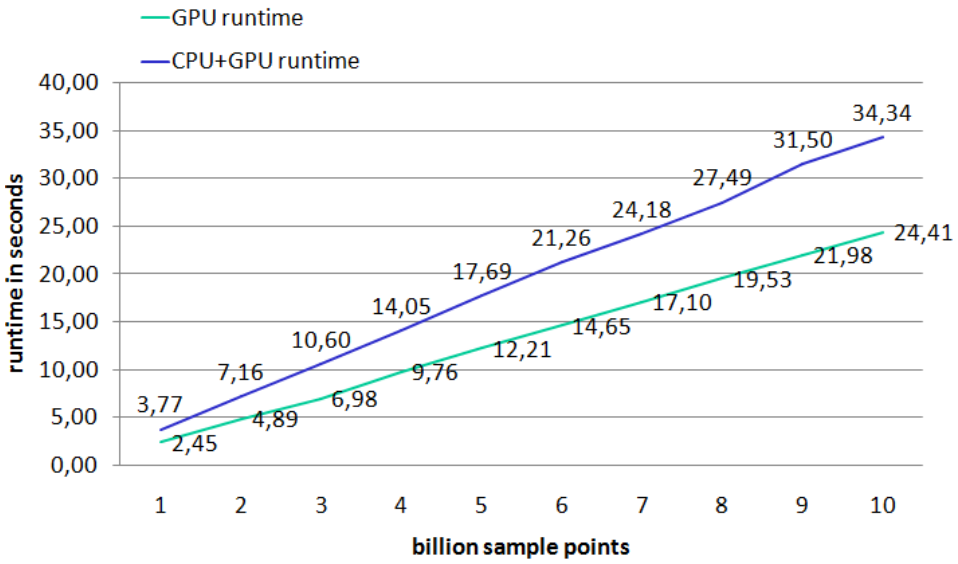

Figure 5.2: Runtime of the implementation from Section 4.1.2 using a single GPU for a constant number of traces D and a variable number of sample points T.

For the plot illustrated in Figure 5.2, we fixed the number of traces D to 10000 and increased the number of sample points T from 100000 to 1 million in steps of 100000. Therefore, the number of sample points V is identical to Figure 5.1.

The combined CPU and GPU runtime includes all overhead caused by the CPU (memory allocation / deallocation, reading traces and plaintexts from the hard drive, data transfers, searching for the maximum correlation coefficient), while the GPU runtime is the summation of the runtime of all invoked kernels.
The runtimes behave the same for a fixed number of traces D and a variable number of sample points T respectively for a fixed number of sample points T and a variable number of traces D with minor deviations. Both runtimes are linear in V.

Figure 5.3 and Figure 5.4 illustrate the same approach for the implementation using four GPUs.

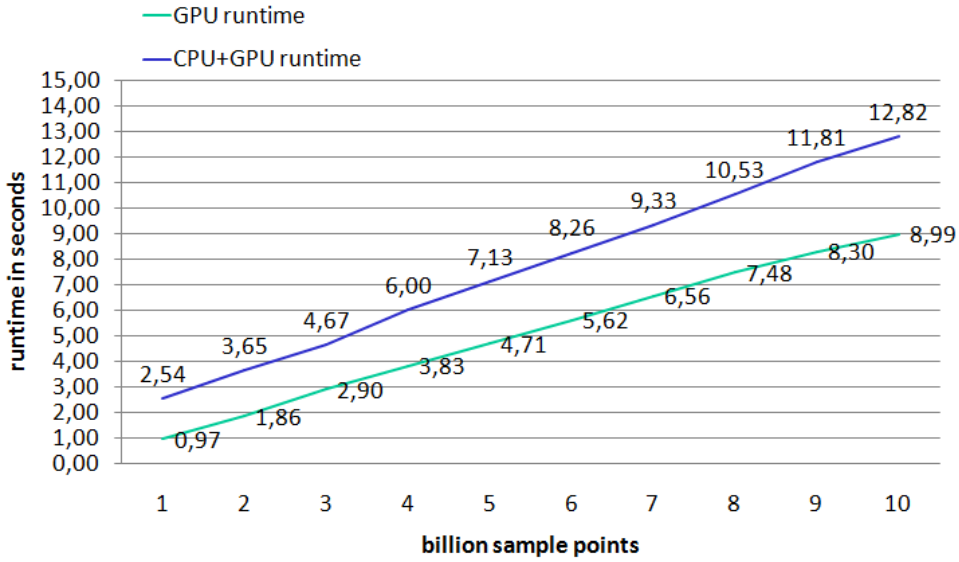

Figure 5.3: Runtime of the implementation from Section 4.1.2 using four GPUs for a constant number of sample points T and a variable number of traces D.

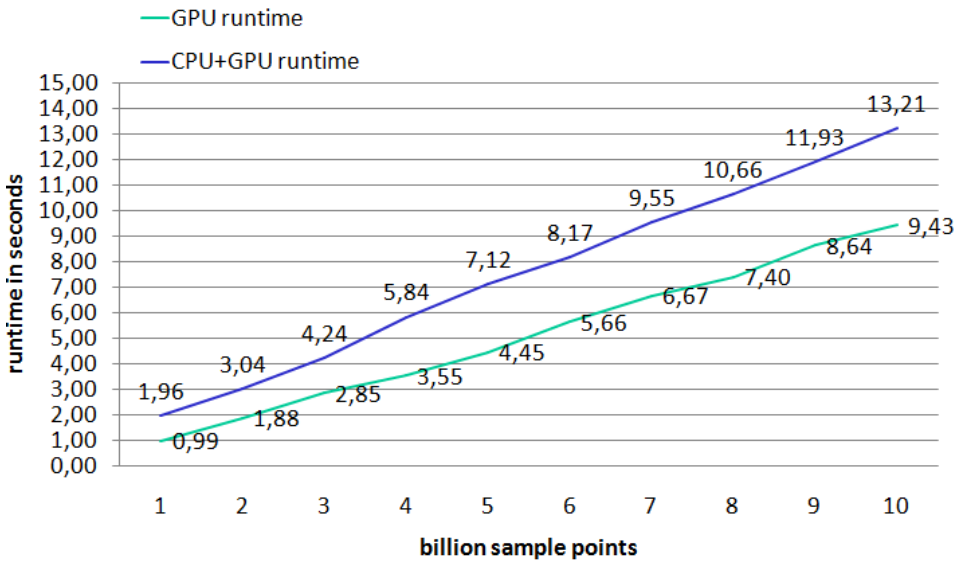

Figure 5.4: Runtime of the implementation from Section 4.1.2 using four GPUs for a constant number of traces D and a variable number of sample points T.

The GPU as well as the combined CPU and GPU runtime is slightly worse for a fixed number of traces D and variable number of sample points T than for a fixed number of sample points T and a variable number of traces D. Nevertheless, the runtimes behave the same in both scenarios and are linear in V.

We implemented a CPU based first order CPA attack using openmp for the parallelization. For a fair comparison, we implemented the CPU code as similar as possible to the GPU code. The used CPUs are two INTEL Xeon E5-2650 V3, see Chapter 3. Each CPU contains 10 cores and supports hyperthreading, thus we launch the attack with 40 openmp threads. All implementations are compiled with the compiler flag `-O3` in order to optimize the code with respect to the runtime. First, the CPU runtime is illustrated in Figure 5.5.

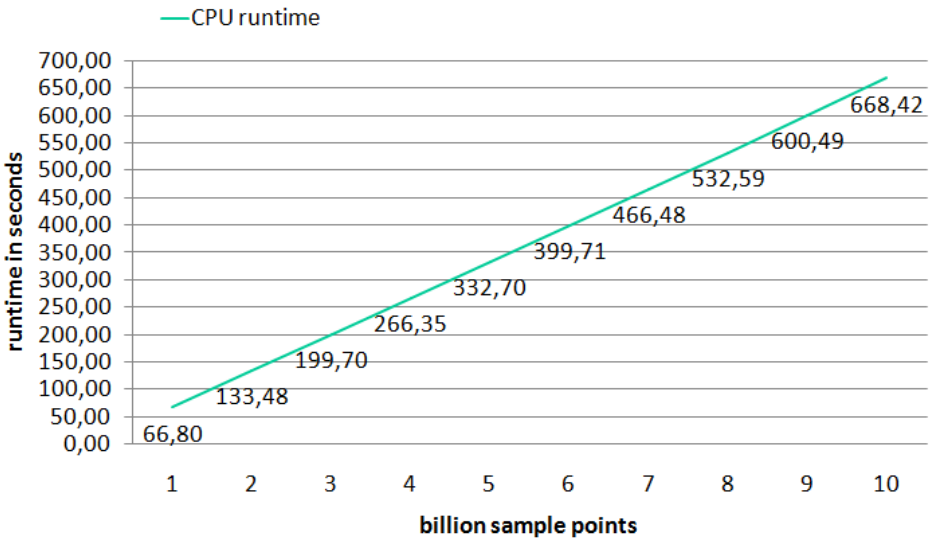

Figure 5.5: Runtime of the first order CPU implementation for a constant number of sample points T and a variable number of traces D.

For the plot, we fixed the number of sample points T to 1 million and increased the number of traces D from 1000 to 10000 in steps of 1000. Therefore, the number of processed sample points V ranges from 1 to 10 billion. The runtime is linear in V. Empirical measurements revealed that, if we fix the number of traces D to 10000 and alter the number of sample points T from 100000 to 1 million in steps of 100000, the runtime is approximately the same and therefore also linear in V. The comparison between the GPU and CPU is shown in Figure 5.6 and Figure 5.7.

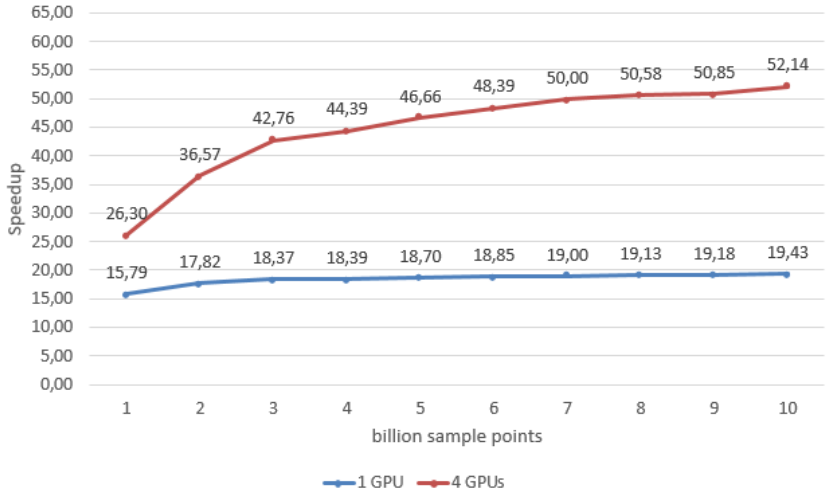

Figure 5.6: First part of the comparison between GPU and CPU for a first order attack.

For the comparison, we fixed the number of sample points T to 1 million and increased the number of traces D from 1000 to 10000 in steps of 1000. Therefore, the number of processed sample points V ranges from 1 to 10 billion. For the GPU implementations, we consider the combined GPU and CPU runtime.

Since the speed-up increases constantly from 1 to 10 billion processed sample points V, we conclude that it is not bounded by 19.43 for the single GPU version and by 52.14 for the multiple GPU version. Therefore, we conducted a second measurement, where we increased the number of traces from 20000 to 100000 in steps of 20000 while fixing the number of sample points T to 1 million. The result is shown in Figure 5.7.

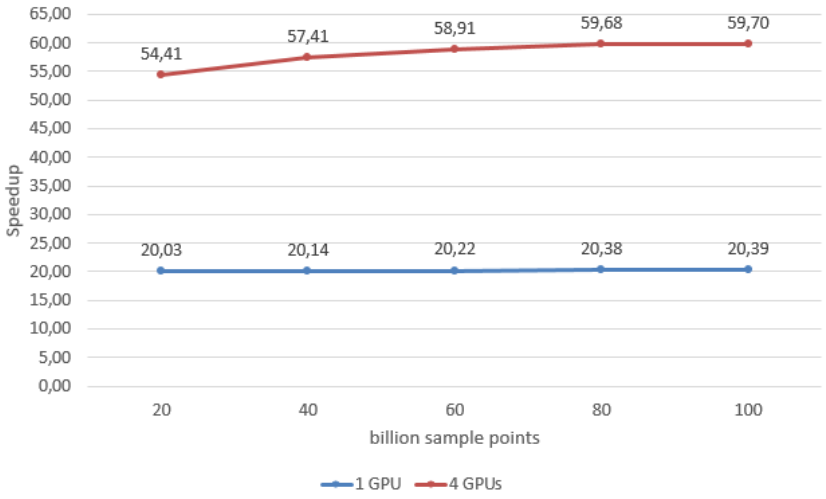

Figure 5.7: Second part of the comparison between GPU and CPU for a first order attack.

The speed-up of the single GPU implementation compared to the CPU based one reaches its maximum of 20.39 at the point of processing 100 billion sample points. Because the speed-up does not significantly increase from 80 to 100 billion sample points, we imply that it is limited to approximately this value. The speed-up of the multiple GPU implementation increases significantly until it reaches its maximum of 59.70 at the point of processing 100 billion sample points. The speed-up does not significantly increase from 80 to 100 billion sample points. From this, we conclude that the speed-up is limited to approximately this value. At the point of processing 80 billion sample points, the multiple GPU implementation is capable of processing 1 billion sample points in 1.17 seconds. Although we use four GPUs, the speed-up of the multiple GPU implementation is bounded by approximately 3 compared to the single GPU implementation.

5.2 Higher Order CPA

To measure the performance of the implementation discussed in Section 4.2.2 with one respectively four GPUs, we use a trace pull that consists of 20000 traces with 200000 sample points per trace. The measurements are conducted for a third order attack, which includes a second and a first order attack, see Section 4.5. Figure 5.8 and Figure 5.9 show the runtime of the single GPU version for different numbers of processed sample points V.

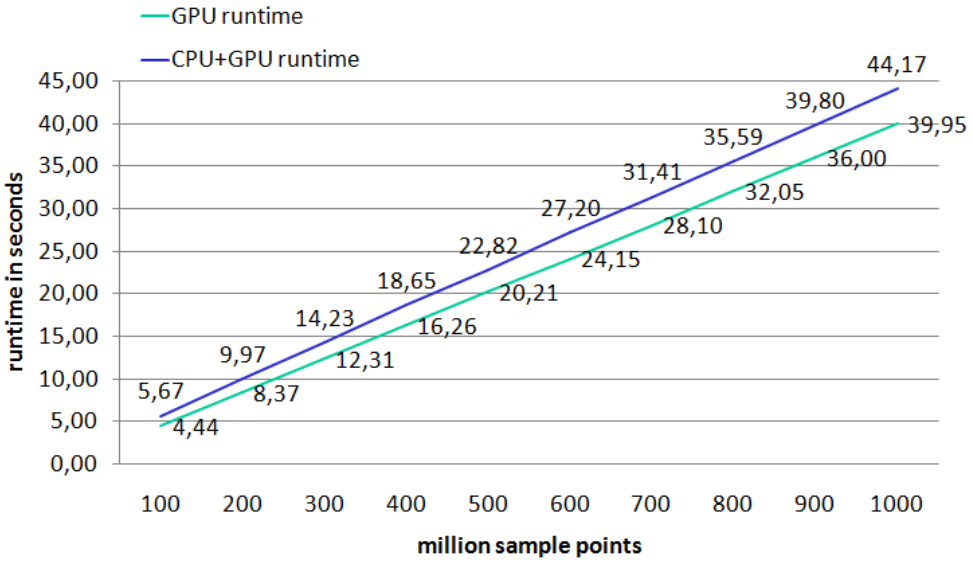

Figure 5.8: Runtime of the implementation from Section 4.2.2 using one GPU for a constant number of sample points T and a variable number of traces D.

For the plot illustrated in Figure 5.8, we fixed the sample points T to 100000 and increased the number of traces D from 1000 to 10000 in steps of 1000. Therefore, the number of processed sample points V ranges from 100 million to 1 billion.

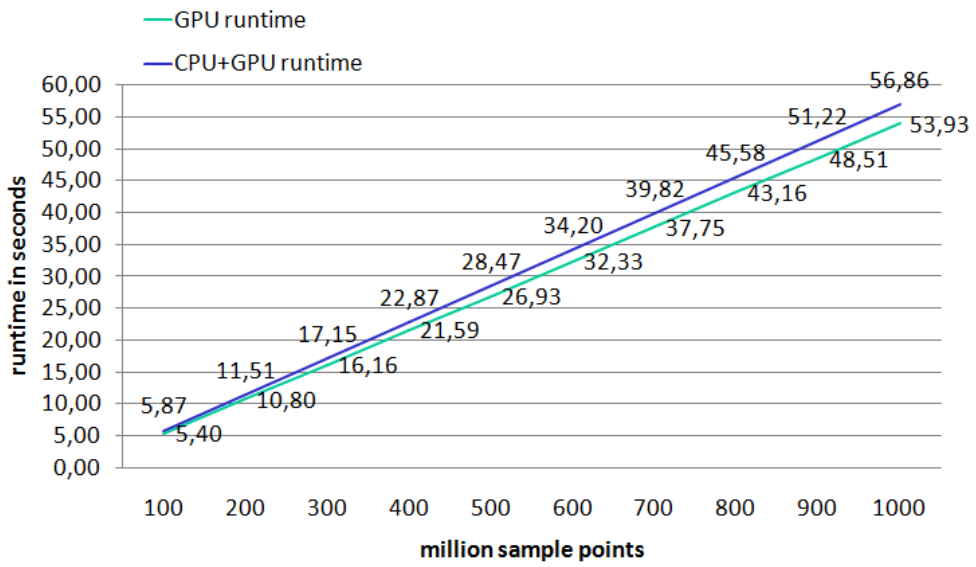

Figure 5.9: Runtime of the implementation from Section 4.2.2 using one GPU for a constant number of traces D and a variable number of sample points T.

For the plot illustrated in Figure 5.9, we fixed the number of traces D to 10000 and increased the number of sample points T from 10000 to 100000 in steps of 10000. Therefore, the number of processed sample points V is identical to the plot illustrated in Figure 5.8.

The runtimes behave the same for a fixed number of traces D and a variable number of sample points T respectively for a fixed number of sample points T and a variable number of traces D and are linear in V. However, the runtimes for a fixed number of traces D and a variable number of sample points T are worse since we enlarge T in steps of 10000. Therefore, we are not able to use the optimal number of sample points per iteration of 4000, see Section 4.2.3. Instead, we have to use 5000 sample points per iteration.

Figure 5.10 and Figure 5.11 illustrate the same approach for the implementation using four GPUs.
For the plot illustrated in Figure 5.10, we fixed the sample points T to 100000 and increased the number of traces D from 2000 to 20000 in steps of 2000. Therefore, the number of processed sample points V ranges from 200 million to 2 billion.

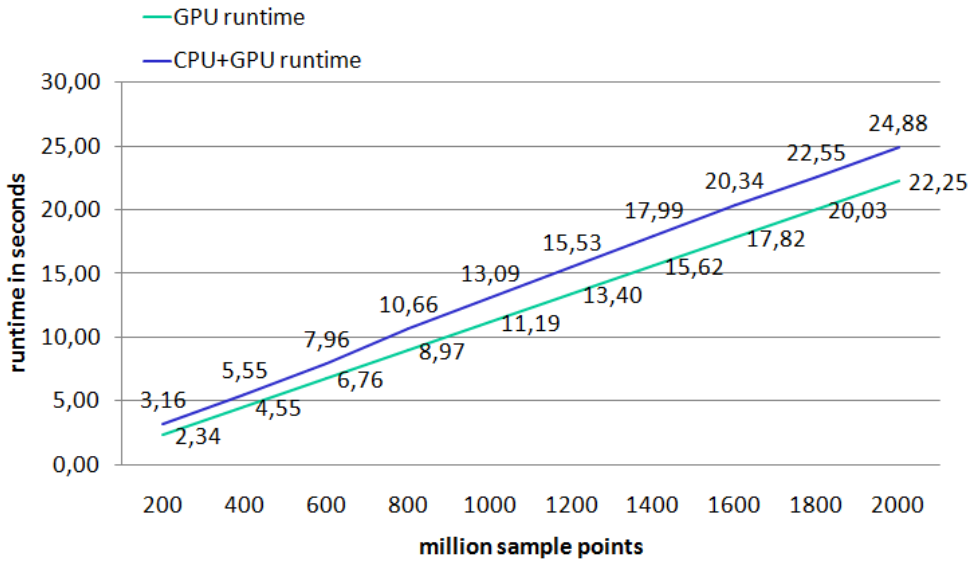

Figure 5.10: Runtime of the implementation from Section 4.2.2 using four GPUs for a constant number of sample points T and a variable number of traces D.

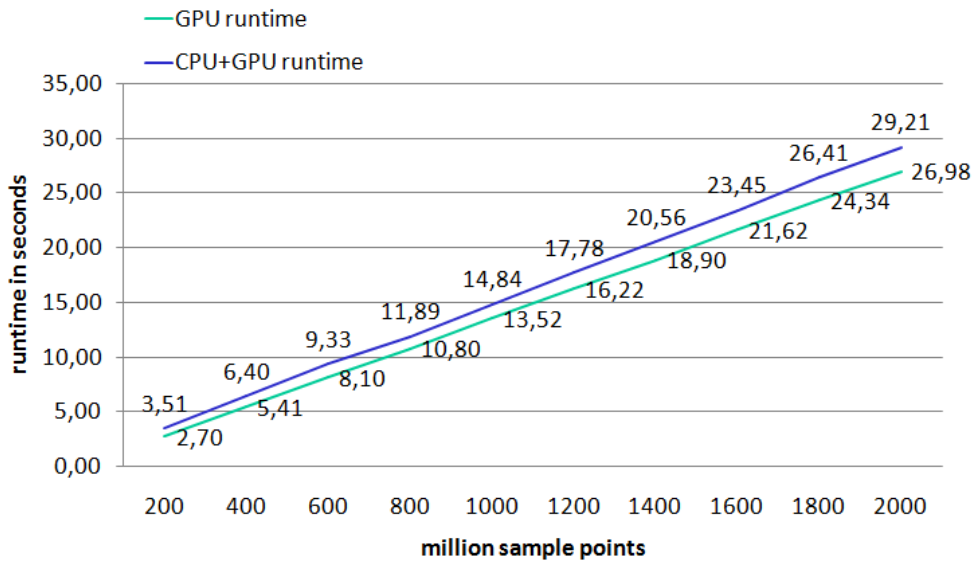

Figure 5.11: Runtime of the implementation from Section 4.2.2 using four GPUs for a constant number of traces D and a variable number of sample points T.

For the plot illustrated in Figure 5.11, we fixed the number of traces D to 10000 and increased the number of sample points T from 20000 to 200000 in steps of 20000. Therefore, the number of processed sample points V is identical to the plot illustrated in Figure 5.10. We increased the number of traces D respectively the number of sample points T in steps

of 2000 respectively 20000 since we use 5000 sample points per iteration. Thus, to evenly distribute the work to four GPUs, we have to increase the number of iterations by four with respect to the sample points.

The GPU as well as the combined CPU and GPU runtimes are linear in the number of processed sample points V. The runtimes for a constant number of traces D and a variable number of sample points T are worse than for a constant number of sample points T and variable number of traces D with the same reason as for the runtimes of Figure 5.8 and Figure 5.9.

We implemented a CPU based third order CPA attack using 40 openmp threads for the parallelization. For a fair comparison, we implemented the CPU code as similar as possible to the GPU code. All implementations are compiled with the compiler flag `-O3` in order to optimize the code with respect to the runtime. First, the CPU runtime is illustrated in Figure 5.12.

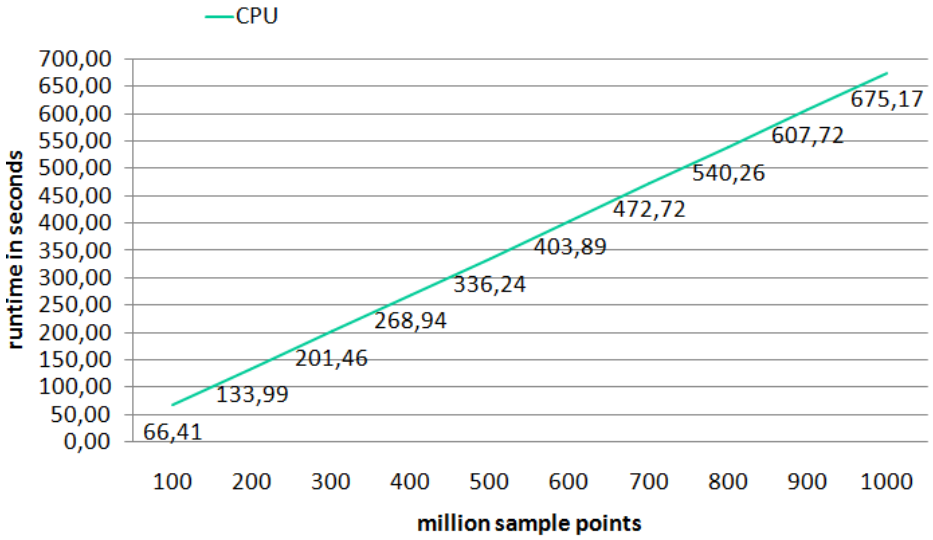

Figure 5.12: Runtime of the third order CPU implementation for a constant number of sample points T and a variable number of traces D.

For the plot, we fixed the number of sample points T to 100000 and increased the number of traces D from 1000 to 10000 in steps of 1000. Therefore, the number of processed sample points V ranges from 100 million to 1 billion. The runtime is linear in V. Empirical measurements revealed that, if we fix the number of traces D and alter the number of sample points T, the runtime is approximately the same and therefore also linear in V. The runtime of the third order CPU implementation, which includes a first and a second order attack, is approximately 10 times slower than the raw first order implementation. The comparison between the GPU and CPU is shown in Figure 5.13.

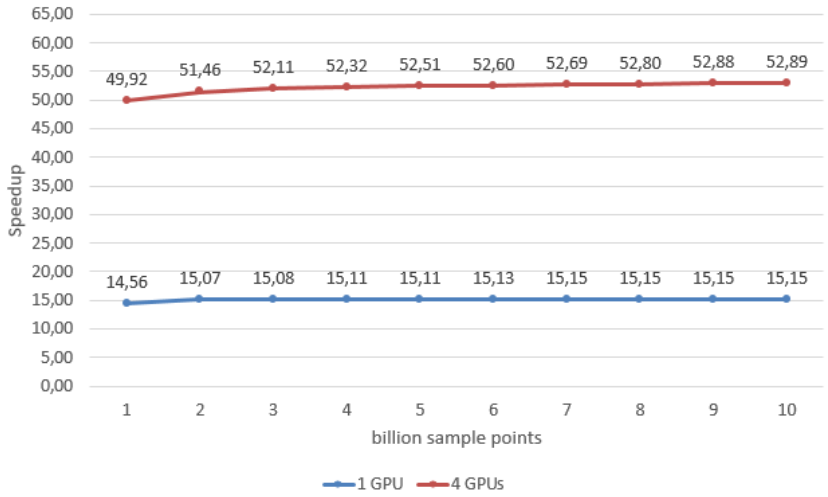

Figure 5.13: Comparison between GPU and CPU for a third order attack.

For the comparison, we fixed the number of sample points T to 100000 and increased the number of traces D from 10000 to 100000 in steps of 10000. Therefore, the number of processed sample points V ranges from 1 to 10 billion. For the GPU implementations, we consider the combined GPU and CPU runtime and use the optimal number of sample points per iteration.

At the point of processing 7 billion sample points, the speed-up of the single GPU implementation reaches its maximum of 15.15 and remains constant for larger numbers of sample points. This implies that the speed-up is limited to this value. For the multiple GPU version, the speed-up hardly increases and reaches its maximum of 52.89 at the point of processing 10 billion sample points. Since the speed-up does not significantly increase from 9 to 10 billion sample points, we conclude that it is bounded by approximately this value. Although we use four GPUs, the speed-up of the multiple GPU implementation compared to the single GPU implementation is limited to 3.49.

We define the cost overhead as the ratio of the costs for the GPUs to the costs of the CPUs

$$c_{overhead} = \frac{c_{gpu}}{c_{cpu}}$$

Furthermore, we define the cost effectiveness as

$$c_{eff} = \frac{c_{overhead}}{s}$$

where s denotes the speed-up of the GPU implementation compared to the CPU implementation.

At the point of writing this thesis, the price for two Tesla K80 [1] is approximately 8000$, while two INTEL Xeon E5-2650 V3 cost approximately 2000$. Therefore, with $c_{overhead} = 4$, the cost effectiveness is equal to $c_{eff} = 14.93$ for a first order attack and $c_{eff} = 13.22$ for a third order attack.

In a final conclusion, we are able to conduct a first order CPA attack respectively a third order CPA attack 14.93 respectively 13.22 times faster - at the same costs - by evaluating them on GPUs, at least for these particular GPUs and CPUs.

[1] Note that each Tesla K80 consists of two GPUs. Therefore, we consider two Tesla K80 and the GPU implementations using four GPUs for the analyis.

6 Conclusion

We provided information on GPU assisted side-channel evaluation by implementing a framework, which contains various correlation-based power-analysis attacks for first and higher orders. Thereby, we used the concept of incremental and iterative robust one-pass formulas.
Currently, the framework only supports univariate attacks. We consider the integration of multivariate attacks, which are also covered by [SMG15], as further work. Moreover, the attacks are limited to AES and use the Hamming-weight model to estimate the power consumption of the attacked device. The extension of the framework to support other cryptographic algorithms and the support for a wider range of power models can be seen as further work as well. It is conceivable to perform the attack with multiple power models in parallel since at the moment, we parallelize the attack in up to two dimensions. As mentioned in Section 2.1.1, CUDA allows parallelizations in up to three dimensions. Furthermore, the framework can be extended to higher orders than three, which is in the current state the highest order on which an attack can be performed.
We highlighted that trace preprocessing can efficiently be implemented on GPUs by the example of peak extraction. The framework can further be extended to support other trace preprocessing-techniques like integration and trace alignment.
In Section 4.1.3 and Section 4.2.3, we discussed various possibilities to optimize the attacks with regards to runtime and pointed out that multiple factors influence the runtime. Moreover, those parameters partially depend on the layout of the trace pull, especially the number of sample points per trace. For practical applications of the framework, it is vital to optimally set the parameters in order to obtain a minimal runtime. Thereby, the examination of the best execution configuration can be challenging, if it has to be conducted manually. Thus, we see the implementation of an automated execution finder as future work.
The framework's property to be scalable enables us to evenly distribute the work of the attack among an arbitrary large cluster of servers and moreover to distribute the work for each server among an arbitrary large number of GPUs. However, currently it is not possible to evenly distribute the work across multiple GPUs with different computation capabilities. Thus, the GPU with the lowest computation power determines the runtime of the attack. This can be counteracted by implementing the second approach discussed in Section 4.3.
The performance measurements in Chapter 5 revealed that the parallel computation capabilities of GPUs are superior to that of CPUs with respect to correlation power analysis. By evaluating the performance of a specific GPU and CPU, we showcased that we obtain a speed-up of up to approximately 60 for a first order CPA attack and up to approximately 53 for a third order attack. This results in a cost effectiveness of 14.93

respectively 13.22 in favour of the GPUs.
In a final conclusion, the thesis showcased that the performance of correlation-based power-analysis attacks can be enhanced tremendously by evaluating them on GPUs, which implies that this is a very promising architecture to examine compute-intensive side-channel attacks.

7 Acronyms

AES Advanced Encryption Standard

ALU Arithmetic Logic Unit

API Application Programming Interface

APU Accelerated Processing Unit

CMOS Complementary Metal Oxide Semiconductor

CPA Correlation Power Analysis

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

DES Data Encryption Standard

DPA Differential Power Analysis

FPGA Field Programmable Gate Array

GPGPU General Purpose Computation on Graphics Processing Unit

GPU Graphics Processing Unit

IEEE Institute of Electrical and Electronics Engineers

MPI Message Passing Interface

OpenCL Open Computing Language

OpenGL Open Graphics Library

PCI-E Peripheral Component Interconnect Express

SFU Special Function Unit

SIMD Single Instruction Multiple Data

SIMT Single Instruction Multiple Thread

SM Streaming Multiprocessor

SPA Simple Power Analysis

List of Figures

List of Tables

List of Listings

Bibliography

[api] NVIDIA CUDA Runtime API. `http://docs.nvidia.com/cuda/cuda-runtime-api`. Accessed: 06.07.2016.

[BDL01] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the Importance of Eliminating Errors in Cryptographic Computations. *J. Cryptology*, 14(2):101–119, 2001.

[BLR11] Timo Bartkewitz and Kerstin Lemke-Rust. A high-performance implementation of differential power analysis on graphics cards. In *Smart Card Research and Advanced Applications*, pages 252–265. Springer, 2011.

[Bon00] André B Bondi. Characteristics of scalability and their impact on performance. In *Proceedings of the 2nd international workshop on Software and performance*, pages 195–203. ACM, 2000.

[CDOR09] Dar-Jen Chang, Ahmed H Desoky, Ming Ouyang, and Eric C Rouchka. Compute pairwise manhattan distance and pearson correlation coefficient of data points with gpu. In *Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing, 2009. SNPD'09. 10th ACIS International Conference on*, pages 501–506. IEEE, 2009.

[CGM14] John Cheng, Max Grossman, and Ty McKercher. *Professional Cuda C Programming*. John Wiley & Sons, 2014.

[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards Sound Approaches to Counteract Power-Analysis Attacks. In Michael J. Wiener, editor, *CRYPTO*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 1999.

[Coo12] Shane Cook. *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes, 2012.

[Far11] Rob Farber. *CUDA application design and development*. Elsevier, 2011.

[Fly72] Michael J Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, 100(9):948–960, 1972.

[GNG+11] Daniel Gembris, Markus Neeb, Markus Gipp, Andreas Kugel, and Reinhard Männer. Correlation analysis on GPU systems using NVIDIA's CUDA. *Journal of real-time image processing*, 6(4):275–280, 2011.

[GRJ14] Hasindu Gamaarachchi, Roshan Ragel, and Darshana Jayasinghe. Accelerating correlation power analysis using graphics processing units (GPUs). In *Information and Automation for Sustainability (ICIAfS), 2014 7th International Conference on*, pages 1–6. IEEE, 2014.

[Hna10] William Hnath. *Differential power analysis side-channel attacks in cryptography.* PhD thesis, Worcester Polytechnic Institute, 2010.

[i7] Intel Core i7-3900 Desktop Processor Series. `http://www.intel.com/content/dam/support/us/en/documents/processors/corei7/sb/core_i7-3900_d.pdf`. Accessed: 06.07.2016.

[K80] Tesla K80 GPU accelerator. `http://www.nvidia.de/object/tesla-k80-de.html`. Accessed: 06.07.2016.

[KNT+11] Ekasit Kijsipongse, Chumpol Ngamphiw, Sissades Tongsima, et al. Efficient large Pearson correlation matrix computing using hybrid MPI/CUDA. In *Computer Science and Software Engineering (JCSSE), 2011 Eighth International Joint Conference on*, pages 237–241. IEEE, 2011.

[KWm12] David B Kirk and W Hwu Wen-mei. *Programming massively parallel processors: a hands-on approach.* Newnes, 2012.

[Len96] Arjen K. Lenstra. Memo on RSA signature generation in the presence of faults. Technical report, 1996. manuscript.

[MBMT13] Hassen Mestiri, Noura Benhadjyoussef, Mohsen Machhout, and Rached Tourki. A Comparative Study of Power Consumption Models for CPA Attack. *International Journal of Computer Network and Information Security*, 5(3):25, 2013.

[MOP08] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks: Revealing the secrets of smart cards*, volume 31. Springer Science & Business Media, 2008.

[nvi] Technical Specifications per Compute Capability. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities`. Accessed: 06.07.2016.

[nvi10] Optimizing Matrix Transpose in CUDA. `http://docs.nvidia.com/cuda/samples/6_Advanced/transpose/doc/MatrixTranspose.pdf`, 2010. Accessed: 06.07.2016.

[nvi12] NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. `https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`, 2012. Accessed: 06.07.2016.

[nvi15] TESLA K80 GPU ACCELERATOR - Board Specification. `http://images.nvidia.com/content/pdf/kepler/Tesla-K80-BoardSpec-07317-001-v05.pdf`, 2015. Accessed: 06.07.2016.

[SMG15] Tobias Schneider, Amir Moradi, and Tim Güneysu. Robust and one-pass parallel computation of correlation-based attacks at arbitrary order. *IACR Cryptol. ePrint Arch*, 571:2015, 2015.

[SSL+14] Tushar Swamy, Neel Shah, Pei Luo, Yunsi Fei, and David Kaeli. Scalable and efficient implementation of correlation power analysis using graphics processing units (GPUs). In *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy*, page 10. ACM, 2014.

[War13] Henry S Warren. *Hacker's delight*. Pearson Education, 2013.