A High-Performance Implementation of Differential Power Analysis on Graphics Cards

Timo Bartkewitz and Kerstin Lemke-Rust

Department of Computer Science,
Bonn-Rhine-Sieg University of Applied Sciences,
Grantham-Allee 20, 53757 Sankt Augustin, Germany
{timo.bartkewitz,kerstin.lemke-rust}@h-brs.de

Abstract. We present an implementation for Differential Power Analysis (DPA) that is entirely based on Graphics Processing Units (GPUs). In this paper we make use of advanced techniques offered by the CUDA Framework in order to minimize the runtime. In security testing DPA still plays a major role for the smart card industry and these evaluations require, apart from educationally prepared measurement setups, the analysis of measurements with large amounts of traces and samples, and here time does matter. Most often DPA implementations are tailor-made and adapted to fit certain platforms and hence efficient reference implementations are sparsely seeded. In this work we show that the powerful architecture of graphics cards is well suited to facilitate a DPA implementation, based on the Pearson correlation coefficient, that could serve as a high performant reference, e.g., by analyzing one million traces of $20k$ samples in less than two minutes.

Keywords: DPA, CPA, Graphics Cards, CUDA

1 Introduction

The resistance of a cryptographic device against side channel attacks is defined by the amount of traces that is at least required to recover a secret information that is embedded in the device under a specific adversarial scenario. In commercial applications, many crypto devices which can either consist of a microcontroller or an FPGA (Field Programmable Gate Array), respectively an ASIC (Application-Specific Integrated Circuit) are hardened to resist side channel attacks. In order to fulfill the security requirements for highly resistant devices side channel testing with one million or even more traces are nowadays common for security labs, cf. [7] for the state of the art in testing AES hardware implementations in 2005. Currently, CPA attacks in research are carried out with up to one hundred million traces [8]. Both processes that are implied by a side channel attack, namely trace recording and computational analysis, can be highly time consuming for a smart card evaluation. Contrary to the effort involved in the trace recording that is mostly dictated by the device and done once, the computational analysis is repeatedly carried out to meet different attack scenarios. Hence, an acceleration

concerning the analysis part is definitely desirable. Differential Power Analysis (DPA) [4] using the Pearson correlation coefficient [2] is still the most common statistical tool to evaluate the side channel resistance.

Motivation: Graphics cards provide a powerful parallel architecture which became widely accepted in scientific computations. Also cryptography is well established on GPUs with several implementations that were made during the last few years. For instance, cryptosystems like ECC, RSA, and AES were efficiently implemented [13, 1, 3]. Until now, only little efforts were spent to speed up DPA with the help of graphics cards. The only other proposal we are aware of is [5]. Their approach includes Difference of Means (DOM) as the statistical test and makes use of both, the general purpose CPU and a CUDA related GPU. All in all they achieve a reasonable speed-up factor of about two by parallelizing the summation of samples on the GPU side.

Our contribution: In this paper we make use of advanced techniques offered by CUDA to achieve key benefits for the runtime. Further, we shift any computation to the GPU. We adopt algorithms for DPA to achieve optimal results on graphics cards involving the Pearson correlation coefficient as the statistical test. A major part for the speed-up is the covariance whose implementation is done through a matrix multiplication which performs very well on graphics cards.

Organization of the paper: This paper is organized as follows: Section 2 briefly introduces Differential Power Analysis, especially the formula of the Pearson correlation coefficient that will be adapted in the remainder. Section 3 provides a short overview on modern graphics cards architecture considering their programming and memory model. In Section 4, we describe the chosen implementation approach concerning algorithms and requirements when being applied on a graphics card. Section 5 reports our experimental results before we conclude in Section 6.

2 Differential Power Analysis

Differential Power Analysis (DPA) is a passive implementation attack aiming at key recovery of a cryptographic implementation. The physical leakage of the device that is exploited is usually the power consumption or the electromagnetic emanation of the device while it processes the cryptographic algorithm. DPA is a divide-and-conquer attack, i.e., a cryptographic key is successively compromised by its subkeys.

We assume that the device processes a sensitive variable z which is the conjunction of known input v to the cryptographic computation, i.e., a plaintext or ciphertext and unknown secret information embedded in the device, i.e., a subkey k, such that $z = f_k(v)$. We further assume that the physical leakage of the device under test leaks is modeled by

$$L_t = \delta_t + \mathcal{L}(z) + B_t.$$

Herein, L_t is the leakage at time t that depends on a constant portion δ_t, a certain deterministic leakage function $\mathcal{L}(\bullet)$ that describes how the leaked information depends on the sensitive variable z, and a noise term B_t, a randomly

and normally distributed variable centered in zero with standard deviation σ. Note that in practice the leakage function $\mathcal{L}(\bullet)$ is usually not known by the adversary, however, it is well-known that often the Hamming weight of z is a good approximation [6]. Alternatively, the adversary may evaluate single-bit leakage of z.

Measurement As the first step of DPA the power consumption of the device under attack is measured with a digital storage oscilloscope (DSO). Such a measurement is given by the matrix

$$\mathbf{X} = \begin{pmatrix} X_1 \; X_2 \; X_3 \ldots X_s \end{pmatrix} = \begin{pmatrix} x_{1,1} & x_{1,2} & x_{1,3} & \ldots & x_{1,s} \\ x_{2,1} & x_{2,2} & x_{2,3} & \ldots & x_{2,s} \\ x_{3,1} & x_{3,2} & x_{3,3} & \ldots & x_{3,s} \\ \vdots & \vdots & \vdots & \ldots & \vdots \\ x_{m,1} & x_{m,2} & x_{m,3} & \ldots & x_{m,s} \end{pmatrix}$$

involving m independent measurements (traces) containing s samples per trace, where each $x_{i,j}$ is a sample from trace i at time j (row-major order).

There are usually different (randomly chosen and uniformly distributed) inputs for each trace i, such that

$$\mathbf{V} = \begin{pmatrix} v_{1,1} & v_{1,2} & v_{1,3} & \ldots & v_{1,b} \\ v_{2,1} & v_{2,2} & v_{2,3} & \ldots & v_{2,b} \\ v_{3,1} & v_{3,2} & v_{3,3} & \ldots & v_{3,b} \\ \vdots & \vdots & \vdots & \ldots & \vdots \\ v_{m,1} & v_{m,2} & v_{m,3} & \ldots & v_{m,b} \end{pmatrix},$$

where $v_{i,l}$ is the l_{th} input of trace i, for $l \in \{1, 2, \ldots, b\}$ and b is the length of the plaintext.

DPA works with hypotheses on subkeys. Let v_i be the partial entry of row i of the matrix $\mathbf{V}$ that enters the computation of $z = f_k(v)$. That is, we get a hypothetical leakage matrix

$$\mathbf{Y} = \begin{pmatrix} Y_1 \; Y_2 \; Y_3 \ldots Y_p \end{pmatrix} = \begin{pmatrix} y_{1,1} & y_{1,2} & y_{1,3} & \ldots & y_{1,p} \\ y_{2,1} & y_{2,2} & y_{2,3} & \ldots & y_{2,p} \\ y_{3,1} & y_{3,2} & y_{3,3} & \ldots & y_{3,p} \\ \vdots & \vdots & \vdots & \ldots & \vdots \\ y_{m,1} & y_{m,2} & y_{m,3} & \ldots & y_{m,p} \end{pmatrix}$$
$$= \begin{pmatrix} \mathcal{L}(f_1(v_1)) & \mathcal{L}(f_2(v_1)) & \mathcal{L}(f_3(v_1)) & \ldots & \mathcal{L}(f_p(v_1)) \\ \mathcal{L}(f_1(v_2)) & \mathcal{L}(f_2(v_2)) & \mathcal{L}(f_3(v_2)) & \ldots & \mathcal{L}(f_p(v_2)) \\ \mathcal{L}(f_1(v_3)) & \mathcal{L}(f_2(v_3)) & \mathcal{L}(f_3(v_3)) & \ldots & \mathcal{L}(f_p(v_3)) \\ \vdots & \vdots & \vdots & \ldots & \vdots \\ \mathcal{L}(f_1(v_m)) & \mathcal{L}(f_2(v_m)) & \mathcal{L}(f_3(v_m)) & \ldots & \mathcal{L}(f_p(v_m)) \end{pmatrix},$$

covering all p subkey candidates $i \in \{0, 1, \ldots, p\}$.

Pearson Correlation Coefficient In this paper, we use the Pearson correlation coefficient as the statistical test for DPA. It computes the correlation coefficient of each column of the leakage matrix with each column of the measurement matrix. DPA finally outputs the key hypothesis reaching the absolute maximum of correlation.

For completeness, we provide the explicit formula for the estimated Pearson correlation coefficient:

$$r_{X,Y} = \frac{\sum_{i=1}^{m} x_i y_i - \frac{1}{m}\sum_{i=1}^{m} x_i \cdot \sum_{i=1}^{m} y_i}{\sqrt{\sum_{i=1}^{m} x_i^2 - \frac{1}{m}(\sum_{i=1}^{m} x_i)^2} \cdot \sqrt{\sum_{i=1}^{m} y_i^2 - \frac{1}{m}(\sum_{i=1}^{m} y_i)^2}}. \quad (1)$$

3 Computations on Graphics Cards

General-purpose computing on graphics processing units (GPGPU) is the shift of computations that are traditionally handled by the *central processing unit* (CPU) or *host* processor, to the *graphics processing unit* (GPU), also known as *device*. In this paper, we focus on nVidia GPUs and CUDA [10] that can be programmed with *C for CUDA*, a C language derivative with special extensions.

The main unit of the device is the multiprocessor which is a set of a number of stream processors (depends on the generation) which share memory, caches, and an instruction unit. The multiprocessor creates, manages, and executes *threads* in hardware. As Figure 1 shows, a thread in CUDA is the smallest unit of

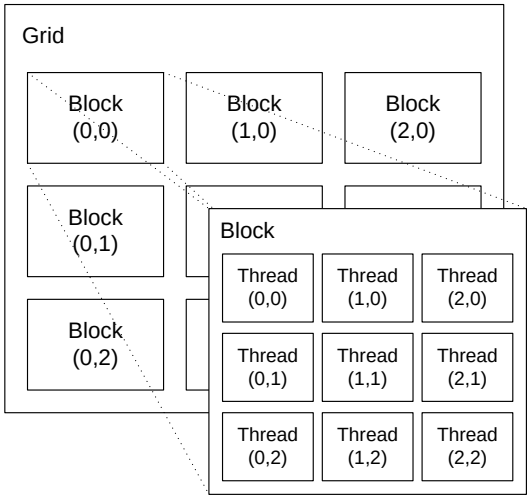

Fig. 1. CUDA thread hierarchy

parallelism that is executed concurrently with other threads (*warps*) on the hardware. Threads are organized in a *thread block*, a group of threads in which

the threads can communicate with each other and synchronize their state. A group of thread blocks is called a *thread grid*. A thread grid forms the execution unit in the CUDA model since it is not possible to execute a thread or thread block solely.

Threads in the CUDA programming model can access data from various memory spaces that differ in size and access time. The CUDA memory model (Fig. 2) describes the accessible memory spaces from the view point of the thread. At lowest level, a thread has read and write access to its own *registers* and additionally its own copy of *local memory*. Threads within the same block have read and write access to a *shared memory* on the next higher level. Beyond the block, all threads can have read and write access to the largest memory space, the *global memory*. Beside the global memory, there are two further spaces that are read-only, the *constant memory* and the *texture memory*. Usually, mem-

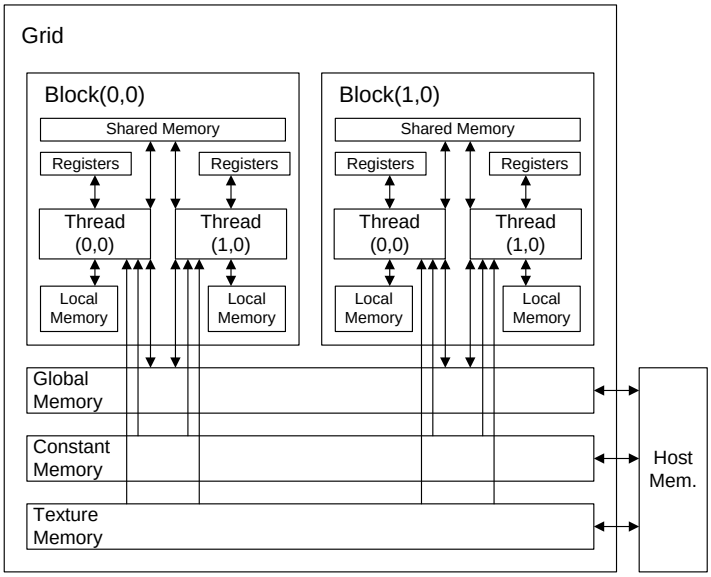

Fig. 2. CUDA memory model

ory spaces that are shared by threads contain potential hazards of conflicts such as *read-after-write*, *write-after-read*, or *write-after-write*. Thus, the programming model implements a barrier by defining a synchronization instruction. As a consequence, a large number of divergent threads (i.e., threads which follow a different execution flow of an algorithm) require frequent synchronization, reducing the overall computation time of the entire systems due to wait cycles. Accesses to the global memory are crucial for the overall performance due to its latency. But most of that latency can be hidden if there are enough independent arithmetic instructions that are executed while waiting for access to complete.

4 Differential Power Analysis on Graphics Cards

The implementation of the Pearson correlation coefficient according to its representation (1) requires us to compute five sums:

$$\sum_{\forall i} x_i y_i, \quad \sum_{\forall i} x_i, \quad \sum_{\forall i} y_i, \quad \sum_{\forall i} x_i^2, \quad \text{and} \sum_{\forall i} y_i^2$$

Taking both matrices $\mathbf{X}$ and $\mathbf{Y}$ into account the first sum embodies the matrix multiplication

$$\mathbf{Y}^T * \mathbf{X} = \begin{pmatrix} Y_1 * X_1 & Y_1 * X_2 & Y_1 * X_3 & \dots & Y_1 * X_s \\ Y_2 * X_1 & Y_2 * X_2 & Y_2 * X_3 & \dots & Y_2 * X_s \\ Y_3 * X_1 & Y_3 * X_2 & Y_3 * X_3 & \dots & Y_3 * X_s \\ \vdots & \vdots & \vdots & \dots & \vdots \\ Y_p * X_1 & Y_p * X_2 & Y_p * X_3 & \dots & Y_p * X_s \end{pmatrix},$$

where X_i and Y_j are column, respectively row vectors of length m.

Matrix multiplications perform very well on graphics cards, and hence the idea is to build an implementation of the correlation coefficient based upon the matrix multiplication. The other sums could then be computed simultaneously. However, in this case prerequisite is the computation of the hypothetical leakage matrix $\mathbf{Y}$ beforehand. Additionally, we have to face some other issues that arise when we aim for an implementation that can handle arbitrary large measurement matrices. First, we have to keep in mind that the global memory of a graphics card is a constrained resource. Second, we probably run into numerical problems considering the single dot products, respectively the sums of large vectors (arithmetic overflow). Finally, we aim to distribute the computation of the correlation coefficient over an arbitrary number of graphics cards, respectively the computation has to be iteratively issuable if only one graphics card is available in the case of very large measurements. Therefore, our implementation approach consists of three major steps, i.e., CUDA kernels, that are carried out iteratively.

Initially, a kernel that computes the leakage model, next a kernel that performs the computations of the sums, and finally a kernel that computes the correlation coefficients. Apart from this approach it is, of course, possible to have one kernel that computes everything but that depends on the fact how large the matrices are. Here, we merely assume measurements being too large to be processed by one kernel at once but we will also briefly include this point into our considerations.

4.1 Leakage Model Creation

The leakage model is created by a kernel that is given an input vector $V \in \mathbf{V}$. Therefore, the row vectors of $\mathbf{Y^T}$, of which each is based on a copy of V, are distributed over different thread blocks, that is each row is computed among several

$$\begin{pmatrix} \boxed{\mathcal{L}(f_1(v_1))\ \mathcal{L}(f_1(v_2))\ \mathcal{L}(f_1(v_3))\ \ldots\ \mathcal{L}(f_1(v_m))} \\ \boxed{\mathcal{L}(f_2(v_1))\ \mathcal{L}(f_2(v_2))\ \mathcal{L}(f_2(v_3))\ \ldots\ \mathcal{L}(f_2(v_m))} \\ \boxed{\mathcal{L}(f_3(v_1))\ \mathcal{L}(f_3(v_2))\ \mathcal{L}(f_3(v_3))\ \ldots\ \mathcal{L}(f_3(v_m))} \\ \vdots \qquad \vdots \qquad \vdots \qquad \ldots \qquad \vdots \\ \boxed{\mathcal{L}(f_p(v_1))\ \mathcal{L}(f_p(v_2))\ \mathcal{L}(f_p(v_3))\ \ldots\ \mathcal{L}(f_p(v_m))} \end{pmatrix}$$

Fig. 3. Computation of $\mathbf{Y^T}$ among different thread blocks (outlined by solid lines)

threads within a thread block as depicted by Figure 3. As usual, the computation of the leakage prediction function $\mathcal{L}(f_k(v_i))$ is realized using a table, e.g. a S-box with precomputed Hamming weights, which is copied into the constant memory of the graphics card prior to the execution of the kernel and referenced later on. Eventually, this kernel can be omitted if the inputs are directly fed into the matrix multiplication kernel. This saves global memory because the leakage model matrix does not need to be stored. However, in some cases it might be more convenient to have a separated kernel when the leakage model is more complex for instance. Our straightforward approach is represented by Algorithm 1. As stated in the algorithm the integer values *tIdx.x* and *bIdx.x* represent the

Algorithm 1 Leakage Model Creation

Input: Input vector $V \in \mathbf{V}$
Output: Leakage model matrix $\mathbf{Y^T}$

1: **for** each block **parallel do**
2: **for** each thread **parallel do**
3: $\mathbf{Y}[bIdx.x, tIdx.x] = \mathcal{L}(f_{bIdx.x}(V[tIdx.x]))$
4: **end for parallel**
5: **end for parallel**

index of a single thread and thread block in the first dimension complying with the CUDA model.

4.2 Computation of the Sums

The computation of the correlation coefficient sums is, as already mentioned above, based upon the matrix multiplication $\mathbf{Y}^T * \mathbf{X}$. In order to achieve the best performance in the sense of DPA some effort has to be spent. Regarding arbitrary large matrices we have to keep in mind that at some point the matrices exceed the global memory of the graphics card. From this point of view we can follow two approaches, first a kernel that is given the matrices and computes the correlation coefficient directly, because the matrices fit the global memory. Contrary, if the matrices do not fit the memory, a kernel is needed that outputs the sums for a later processing. In the remainder we deal with the latter opportunity but the similarity is high. However, these deliberations also lead us to the

next implementation decision. Each sum has to be stored in a single variable and a 32-bit variable may not be sufficient in general because of a potential overflow. This is aggravated by the fact that the CUDA performance involving 64-bit variables is quite low [11]. Nevertheless, the framework is highly optimized to employ 32-bit floating arithmetic. So to address these issues the correlation coefficients, respectively their implied sums, are computed as follows.

$$r_{X,Y} = \frac{\sum_{i=1}^{m} \frac{1}{m} x_i y_i - \sum_{i=1}^{m} \frac{1}{m} x_i \cdot \sum_{i=1}^{m} \frac{1}{m} y_i}{\sqrt{\sum_{i=1}^{m} \frac{1}{m} x_i^2 - (\sum_{i=1}^{m} \frac{1}{m} x_i)^2} \cdot \sqrt{\sum_{i=1}^{m} \frac{1}{m} y_i^2 - (\sum_{i=1}^{m} \frac{1}{m} y_i)^2}} \tag{2}$$

Shifting the fractal into the sums does not necessarily cause a performance penalty due to potential latency hiding since it also works vice versa by which means additional instructions are covered by the waiting time.

Referring to [12] the resultant matrix of the multiplication is computed by two-dimensional thread blocks, here called *tiles*, that are distributed as shown in Figure 4. Each tile consists of n^2 threads where a single thread is responsible

$$\begin{pmatrix} x_{1,1} & x_{1,2} & x_{1,3} & \dots & x_{1,s} \\ x_{2,1} & x_{2,2} & x_{2,3} & \dots & x_{2,s} \\ x_{3,1} & x_{3,2} & x_{3,3} & \dots & x_{3,s} \\ \vdots & \vdots & \vdots & \dots & \vdots \\ x_{m,1} & x_{m,2} & x_{m,3} & \dots & x_{m,s} \end{pmatrix}$$

$$\begin{pmatrix} y_{1,1} & y_{2,1} & y_{3,1} & \dots & y_{m,1} \\ y_{1,2} & y_{2,2} & y_{3,2} & \dots & y_{m,2} \\ y_{1,3} & y_{2,3} & y_{3,3} & \dots & y_{m,3} \\ \vdots & \vdots & \vdots & \dots & \vdots \\ y_{1,p} & y_{2,p} & y_{3,p} & \dots & y_{m,p} \end{pmatrix} \begin{pmatrix} Y_1 * X_1 & Y_1 * X_2 & Y_1 * X_3 & \dots & Y_1 * X_s \\ Y_2 * X_1 & Y_2 * X_2 & Y_2 * X_3 & \dots & Y_2 * X_s \\ Y_3 * X_1 & Y_3 * X_2 & Y_3 * X_3 & \dots & Y_3 * X_s \\ \vdots & \vdots & \vdots & \dots & \vdots \\ Y_p * X_1 & Y_p * X_2 & Y_p * X_3 & \dots & Y_p * X_s \end{pmatrix}$$

Fig. 4. Computation of $\mathbf{Y^T} * \mathbf{X}$ among different two-dimensional thread blocks, here called tiles. Exemplarily, only one tile is emphasized to show which portions of the matrices $\mathbf{X}$ and $\mathbf{Y}$ are involved to compute the resultant sub-matrix covered by that tile. Actually, the whole resultant matrix is covered by several tiles.

to compute a single dot product of the entire resultant matrix. At the beginning a tile loads portions such that n^2 elements of $\mathbf{X}$ and n^2 of $\mathbf{Y^T}$ are deposited into the shared memory of the thread block. This strategy avoids loading every vector each time it is needed. With these portions a thread can now compute the first n products of a dot product, since the first n elements of each row vector of $\mathbf{Y^T}$ and each column vector of $\mathbf{X}$ are loaded. Afterwards, a tile fetches the next portions to compute the next n products, a procedure which is repeated until all elements are passed through. Obviously, we can exploit synergies and compute all other correlation coefficient sums concurrently since all necessary

elements are already loaded. Figuratively, the tile move rightwards regarding $\mathbf{Y^T}$ and downwards regarding $\mathbf{X}$ (Fig. 4). The kernel, constituted by its optimized version, is depicted in Algorithm 2. Additionally, the algorithm reveals

Algorithm 2 Computation of correlation coefficient sums

Input: Leakage model matrix $\mathbf{Y^T}$ and measurement matrix $\mathbf{X}$
Output: Sums of corr. coef.: $\sum_{\forall i} \frac{1}{m} x_i y_i, \sum_{\forall i} \frac{1}{m} x_i, \sum_{\forall i} \frac{1}{m} y_i, \sum_{\forall i} \frac{1}{m} x_i^2$, and $\sum_{\forall i} \frac{1}{m} y_i^2$

1: **for** each block **parallel do**
2: **for** each thread **parallel do**
3: prefetch first tile of $\mathbf{Y^T}$ and first horizontal tiles of $\mathbf{X}$ into registers: y_i, x_i.
4: **end for parallel**
5: **end for parallel**
6:
7: **for** k = 1 to $\frac{m}{n}$ **do**
8: **for** each block **parallel do**
9: **for** each thread **parallel do**
10: $i \leftarrow$ thread position within column vectors of $\mathbf{X}$
11: $j \leftarrow$ thread position within row vectors of $\mathbf{Y^T}$
12: deposit prefetched tiles into shared memory: $X^{shared}[tIdx.x, tIdx.y] = \frac{x_i}{\sqrt{m}}$, $Y^{shared}[tIdx.x, tIdx.y] = \frac{y_j}{\sqrt{m}}$
13: prefetch next tiles into registers: x_{i+k}, y_{j+k}
14: **for** l = 1 to n **do**
15: $\sum_{thread} x_i y_i = \sum_{thread} x_i y_i + X^{shared}[tIdx.x, l] \cdot Y^{shared}[l, tIdx.y]$
16: $\sum_{thread} x_i = \sum_{thread} x_i + X^{shared}[tIdx.x, l] \cdot \frac{1}{\sqrt{m}}$
17: $\sum_{thread} y_i = \sum_{thread} y_i + Y^{shared}[l, tIdx.y] \cdot \frac{1}{\sqrt{m}}$
18: $\sum_{thread} x_i^2 = \sum_{thread} x_i^2 + X^{shared}[tIdx.x, l]^2$
19: $\sum_{thread} y_i^2 = \sum_{thread} y_i^2 + Y^{shared}[l, tIdx.y]^2$
20: **end for**
21: **end for parallel**
22: **end for parallel**
23: **end for**

two mandatory optimizations that were not mentioned so far. The portions are prefetched by the tile threads into their registers first and then deposited into shared memory with the effect that the sum calculations only consume already fetched tile elements while the next elements are being loaded. This enables latency hiding. The second optimization considers the workload balance within kernel. Therefore, a number of tiles of matrix $\mathbf{X}$, instead of one, are loaded horizontally to compute multiple dot products involving the loaded single tile of $\mathbf{Y^T}$.

4.3 Computation of the Correlation Coefficient Matrix

This kernel implementation is similar to that of the leakage model creation. The matrix of the correlation coefficients is segmented in the same way (Fig. 3). Every

thread block is responsible for samples that are related to one key hypothesis, hence a block is given the corresponding correlation coefficient sums which result from the measurement matrix and any sum from the leakage model matrix. The implementation is shown in Algorithm 3.

Algorithm 3 Computation of correlation coefficient matrix

Input: Sums of corr. coef.: $\sum_{\forall i} \frac{1}{m} x_i y_i, \sum_{\forall i} \frac{1}{m} x_i, \sum_{\forall i} \frac{1}{m} y_i, \sum_{\forall i} \frac{1}{m} x_i^2$, and $\sum_{\forall i} \frac{1}{m} y_i^2$
Output: Correlation coefficient matrix $\mathbf{R}$

```
for each block parallel do
    for each thread parallel do
        R[bIdx, tIdx] = r_{X_tIdx, Y_bIdx}, complying with (2) and (3)
    end for parallel
end for parallel
```

4.4 Special Case: Hamming Weight Model

In the case of the Hamming Weight model, a further approach to achieve an even better performance outcome is to estimate the mean and standard deviation of the leakage model. This would save computational effort, above all divisions and square root calculations which should be avoided. These operations only offer one fourth, respectively one eighth of the performance of a multiplication [11]. In addition to that, estimation also saves global memory due the avoided sums that do not need to be stored anymore.

Presuming that the inputs, contained in $\mathbf{V}$, are randomly chosen and uniformly distributed the mean $\sum_{\forall i} \frac{1}{m} y_i$ can be estimated with

$$\sum_{\forall i} \frac{1}{m} y_i = E[HW(Z)] = E[\sum_{i=1}^{b} Z(i)] = b \cdot E[Z(i)] = \frac{b}{2},$$

where $Z(i)$ is the i_{th} bit of random variable Z, i.e., Z is a b-bit variable. Whereas the mean of the squares $\sum_{\forall i} \frac{1}{m} y_i^2$ can be estimated with

$$\sum_{\forall i} \frac{1}{m} y_i^2 = E[HW(Z)^2] = E[\sum_{i,j=1}^{b} Z(i) \cdot Z(j)] = \sum_{i \neq j}^{b} E[Z(i) \cdot Z(j)] + \sum_{i}^{b} E[Z(i)]$$
$$= b \cdot (b-1) \cdot E[Z(i)Z(j)] + b \cdot E[Z(i)] = \frac{b \cdot (b-1)}{4} + \frac{b}{2} = \frac{b^2 + b}{4}.$$

Eventually, we obtain the correlation coefficient with leakage estimation being expressed as

$$r_{X,Y} = \frac{\sum_{i=1}^{m} \frac{1}{m} x_i y_i - \frac{b}{2} \cdot \sum_{i=1}^{m} \frac{1}{m} x_i}{\sqrt{\frac{b}{4}} \cdot \sqrt{\sum_{i=1}^{m} \frac{1}{m} x_i^2 - (\sum_{i=1}^{m} \frac{1}{m} x_i)^2}}. \tag{3}$$

5 Experimental Results

For our experiments, we used nVidia Tesla C2070 graphics cards with 6 GiB video RAM and an Intel Xeon X5660 at 2.8 GHz running Windows 7 64-bit. The results were obtained using the CUDA toolkit and SDK of version *3.2*, the CUDA driver *270.61*, and the Microsoft Visual C++ compiler.

We presume attacking a sequential 8-bit implementation of AES [9] to recover one subkey byte in the Hamming Weight model. In order to gain meaningful results the inputs are composed of byte values that are randomly chosen and uniformly distributed. Since in most cases employing a DSO with a vertical resolution of eight bit suffices, we set $\mathbf{X} \in_R [-127, 127]^{m \times s}$.

First of all the best kernel configuration has to be figured out, more precisely the thread and thread block numbers. For the leakage creation and the correlation coefficient kernel it is quite obvious that they have to be launched with p thread blocks (one block per key hypothesis). Regarding the correlation coefficient sum kernel two constraints show up, the maximum number of threads a kernel can take and the shared memory in use that is dictated by this number. Actually, we have at most 1536 threads per kernel on our graphics card, thus the tile size could be $n_{max} = \lfloor \sqrt{1536} \rfloor = 39$, but due to the restricted shared memory and the horizontal tiles (one tile computes more than one portion of $\mathbf{X}$) we need to find out the optimal trade-off. Through empirical testing, it turned out that the kernel performs best with $n = 28$, that is $n^2 = 784$ threads per block and four horizontal tiles while barely not exceeding the available shared memory. The number of tiles (thread blocks) can be obtained by simply dividing each dimension, the number of key hypotheses p and the number of given samples s, by n. Another limitation is the total global memory M_{global} which accommodates the measurement matrix, the input vector, and the five correlation coefficient sums. Presuming single precision variables for the sums and 8-bit variables for the samples and inputs, the global memory usage consists of $m \cdot s$ bytes for the measurement matrix, m bytes for the inputs, $p \cdot s \cdot 4$ bytes for the covariance, and $2 \cdot p \cdot 4$ bytes, respectively $2 \cdot s \cdot 4$ bytes for the variances and means. Eventually, we obtain the inequality

$$ms + m + 4ps + 8p + 8s < M_{global}.$$

The runtime was then measured in steps of $10k$ traces and a number of samples fixed to $20k$. Figure 5 shows the results for the following variations of the sums kernel: the kernel is given the precomputed leakage model, the kernel is given the input vector directly (cf. Sec. 4.1), and further both variants with the leakage estimation (cf. Sec. 4.4). Additionally, we show the runtime for the data transfer between the host memory and the device memory.

As it can be seen the kernel applying the leakage estimation performs slightly better and further it can be seen that the effort increases linearly with the number of traces. It is not worthwhile to compute the leakage model beforehand which is most likely caused by the frequent global memory accesses. Furthermore, we get the same results if we fix the number of traces and iteratively increase

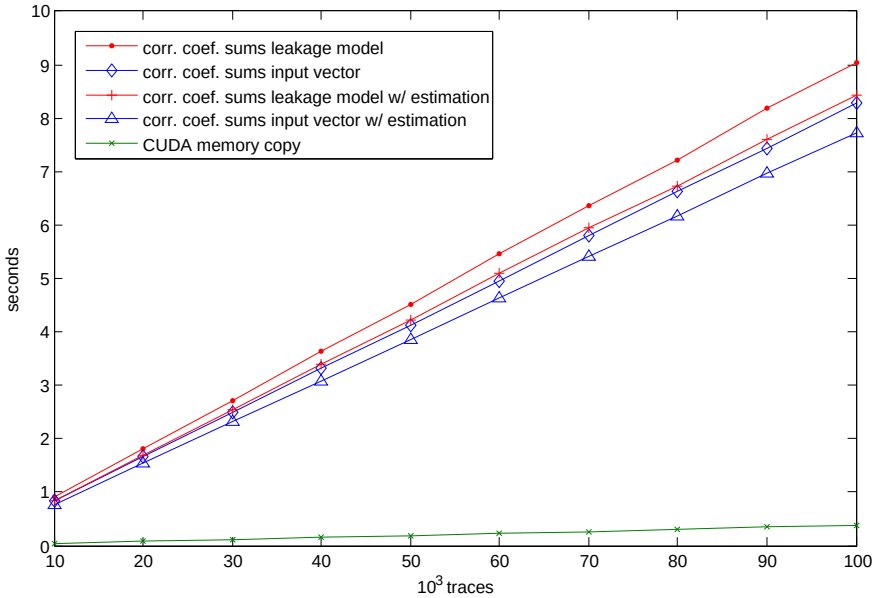

Fig. 5. Runtime of the correlation coefficient sums kernel. The kernel can either be given the leakage model or the input vector directly.

the number of samples by which means the measurement matrix can be cut at any point to make it fit the graphics cards global memory in the case of very large measurements. Since the effort increases linearly with both, the number of traces and the number of samples, the implementation is perfectly scaled with additional graphics cards. The runtimes of the remaining two kernels, namely leakage model creation and correlation coefficient computation, are absolutely negligible while being in the range of a few milliseconds. It is not surprising that their runtimes hardly contribute to the overall runtime since the elements of the resultant matrices are independent of each other, in marked contrast to the sums kernel where as well thread synchronization is vital due to the usage of shared memory. That is also true for the overhead, i.e., transferring data from the host memory to the device memory and vice versa, respectively the kernel launches. However, data transfers are dependent on the data and the time increases linearly with the amount of bytes. Figure 5 thus shows the respective transfers of the full measurement matrix. The influence of I/O, i.e., loading traces from a hard disk, is not considered because this also affects a CPU implementation in the same way. However, measurements containing over one million traces can be analyzed in less than two minutes employing just one graphics card.

Furthermore, we provide a comparison between the CPU and the GPU. Therefore, we implemented and optimized the sums kernel with leakage estimation on the CPU which does exactly the same as its GPU counterpart. Table 1

compares the results. As expected, the CPU implementation is, as well, linearly

Table 1. Runtime comparison between CPU and GPU where one thread runs the CPU implementation. The number of samples is fixed to $20k$.

	$10k$ traces	$20k$ traces	$50k$ traces	$100k$ traces
GPU	0.774 s	1.545 s	3.861 s	7.733 s
CPU	302.72 s	622.11 s	1531.69 s	3152.21 s

scaled with the number of traces. Hence, we can derive a speed-up factor of about almost 100 taking a common processor with four cores into account, a performance gain that obviously suggests CUDA as a very promising platform for DPA.

6 Conclusion and Future Work

In this paper we presented a highly performant implementation of Differential Power Analysis on graphics cards. The implementation can handle arbitrary large measurement matrices which can be split up at any point to make them fit into the graphics cards memory. Large measurements can be analyzed within a few minutes. Our ongoing work will deal with the implementation of DPA based higher order attacks.

Acknowledgements

The work described in this paper was supported by the German Federal Ministry of Education and Research (BMBF) through the EXSET project (promotional reference 01IS10026B). The information in this document reflects the views only of the authors.

References

1. Bernstein, D.J., Chen, T.R., Cheng, C.M., Lange, T., Yang, B.Y.: ECM on Graphics Cards. In: Proceedings of the 28th Annual International Conference on Advances in Cryptology: The Theory and Applications of Cryptographic Techniques. pp. 483–501. EUROCRYPT '09 (2009)
2. Brier, E., Clavier, C., Olivier, F.: Correlation Power Analysis with a Leakage Model. In: Joye, M., Quisquater, J.J. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2004. LNCS, vol. 3156, pp. 16–29. Springer-Verlag (2004)
3. Harrison, O., Waldron, J.: AES Encryption Implementation and Analysis on Commodity Graphics Processing Units. In: Proceedings of the 9th International Workshop on Cryptographic Hardware and Embedded Systems. pp. 209–226. CHES '07 (2007)

4. Kocher, P.C., Jaffe, J., Jun, B.: Differential Power Analysis. In: Wiener, M. (ed.) Advances in Cryptology – CRYPTO '99. LNCS, vol. 1666, pp. 388–397. Springer-Verlag (1999)
5. Lee, S.J., Seo, S.C., Han, D.G., Hong, S., Lee, S.: Acceleration of Differential Power Analysis through the Parallel Use of GPU and CPU. IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences E93.A(9), 1688–1692 (2010)
6. Mangard, S., Oswald, E., Popp, T.: Power Analysis Attacks. Springer (2007)
7. Mangard, S., Pramstaller, N., Oswald, E.: Successfully Attacking Masked AES Hardware Implementations. In: Rao, J.R., Sunar, B. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2005
8. Moradi, A., Poschmann, A., Ling, S., Paar, C., Wang, H.: Pushing the Limits: A Very Compact and a Threshold Implementation of AES. In: Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 69–88 (2011)
9. National Institute of Standards and Technology: Advanced Encryption Standard (AES). Federal Information Processing Standards Publications 197 (2001), `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`
10. nVidia: NVIDIA CUDA Development Tools (2010), `http://developer.download.nvidia.com/compute/cuda/3_2/docs/Getting_Started_Windows.pdf`
11. nVidia: NVIDIA CUDA Programming Guide (2010), `http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Programming_Guide.pdf`
12. Sanders, J., Kandrot, E.: CUDA by Example: An Introduction to General-Purpose GPU Programming. Addison-Wesley Longman, Amsterdam (2010)
13. Szerwinski, R., Güneysu, T.: Exploiting the Power of GPUs for Asymmetric Cryptography. In: Proceedings of the 10th International Workshop on Cryptographic Hardware and Embedded Systems. pp. 79–99. CHES '08 (2008)