High-Speed Parallel Implementations of the Rainbow Method in a Heterogeneous System

Jung Woo Kim[1,*], Jungjoo Seo[1,⋆], Jin Hong[2,**], Kunsoo Park[1,⋆,†], and Sung-Ryul Kim[3,⋆]

[1] Department of Computer Science and Engineering and Institute of Computer Technology, Seoul National University, Seoul, Korea
{jkim,jjseo,kpark}@theory.snu.ac.kr

[2] Department of Mathematical Sciences and ISaC, Seoul National University, Seoul, Korea
jinhong@snu.ac.kr

[3] Division of Internet and Media, Konkuk University, Seoul, Korea
kimsr@konkuk.ac.kr

Abstract. The computing power of graphics processing units (GPU) has increased rapidly, and there has been extensive research on general-purpose computing on GPU (GPGPU) for cryptographic algorithms such as RSA, ECC, NTRU, and AES. With the rise of GPGPU, commodity computers have become complex heterogeneous GPU+CPU systems. This new architecture poses new challenges and opportunities in high-performance computing. In this paper, we present high-speed parallel implementations of the rainbow method, which is known as the most efficient time-memory tradeoff, in the heterogeneous GPU+CPU system. We give a complete analysis of the effect of multiple checkpoints on reducing the cost of false alarms, and take advantage of it for load balancing between GPU and CPU. Our implementation with multiple checkpoints requires no more time on average for resolving false alarms and it actually finishes earlier than generating all online chains unlike other implementations on GPU.

Keywords: Cryptanalysis, Cryptanalytic Time-Memory Tradeoff, Rainbow Method, GPGPU, CUDA, Heterogeneous Computing

1 Introduction

With the GPU's rapid evolution from a graphics processor to a programmable parallel processor, GPU is a many-core multi-threaded multiprocessor that excels

[*] This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No. 20120006492).

[**] This work was supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2012003379)

[†] corresponding author

at not only graphics but also computing applications. Today's GPUs have hundreds of parallel processor cores executing tens of thousands of parallel threads. Using a large number of processors, GPUs are used for accelerating the performance of mathematical and scientific works. General-purpose computing on GPUs (GPGPU) was first introduced in 2006 by unveiling CUDA by NVIDIA [6]. CUDA enables programmers to easily control GPUs by writing programs similar to C.

Recently, researchers and developers have enthusiastically adopted CUDA and GPU computing for cryptographic algorithms. In 2007, Manavski et al. efficiently implemented the Advanced Encryption Standard (AES) algorithm using CUDA [22]. In 2008, Szerwinski and Güneysu made use of CUDA for GPGPU processing of asymmetric cryptosystems (RSA, DSA, ECC) [27]. In 2009, Bernstein et al. showed that GPU can be used for cryptanalysis as well as implementation of cryptographic algorithms [9]. They implemented the elliptic-curve method for integer factorization on GPUs. In 2010, NTRU cryptosystem was implemented on CUDA by Hermans et al. [16].

One-way functions are fundamental tools for cryptography, and it is a hard problem to invert them. There are three generic approaches to invert them. The simplest approach is an exhaustive search. An attacker tries all possible values until the pre-image is found; however, it needs a lot of time. Another simple approach is a table lookup, in which an attacker precomputes the images of a one-way function for all possible pre-images and stores them in a table. The attack can be carried out quickly, but a large amount of memory is needed to store all precomputed values. Cryptanalytic time-memory tradeoffs [21, 11, 14, 26, 20, 10, 23, 8, 28, 18] are compromise solutions between time and memory. Cryptanalytic time-memory tradeoff was introduced by Hellman in 1980 [15]. Rivest proposed to apply *distinguished points* technique [13] to Hellman's method which reduces the number of table lookup operations. In 2003, a new method, which is referred to as *rainbow method*, was suggested by Oechslin [25]. The rainbow method saves a factor of two in the worst case time complexity compared to Hellman's method. Up until now, the rainbow method is the most efficient time-memory tradeoff. Avoine et al. introduced a technique detecting false alarms, called *checkpoints* [7]. Using the technique, the cost of false alarms is reduced with a minute amount of memory.

With the rise of GPGPU, commodity computers are complex heterogeneous GPU+CPU systems that provide high computational power [24, 12]. The GPU and CPU can execute in parallel and have their own independent memory systems connected through the PCIe bus. The GPU+CPU co-processing and data transfers use the bidirectional PCIe bus. This new architecture poses new challenges and opportunities in high-performance computing.

In this paper, we propose high-speed parallel implementations of the rainbow method in the heterogeneous GPU+CPU system through the analysis of the behavior of time-memory tradeoffs. We give a complete analysis of the effect of multiple checkpoints on reducing the cost of false alarms for the non-perfect rainbow table, and take advantage of it for load balancing between GPU and CPU.

The proposed implementation requires no more time on average for resolving false alarms by fully parallelizing the rainbow method on GPU and CPU. Our implementation actually finishes earlier on average than generating all online chains unlike other implementations on GPU. To the best of our knowledge, this is the first work implementing the rainbow method in a heterogeneous system.

The rest of the paper is organized as follows. We begin with an overview of modern GPUs and CUDA in Section 2, followed by a brief review of the rainbow method in Section 3. In Section 4, we describe our fast implementations in a heterogeneous GPU+CPU system. In Section 5, we analyze the checkpoint technique. Finally, Section 6 presents the experimental results.

2 GPGPU and CUDA

While traditional GPUs were used for graphical applications, many modern GPUs can deal with general parallel programs which had been performed normally on CPUs. CUDA [6] is NVIDIA's software and hardware architecture that enables GPUs to be programmed with a variety of high-level programming languages, and it is a parallel computing architecture that is used to improve computing performance by exploiting the power of GPU. NVIDIA has released several improved versions of architectures since its first architecture, G80, and the newest one is called Fermi [4], which was introduced in 2009.

One of the most attractive features of GPUs is that it has a large number of processor cores. Basically, GPUs consist of a number of streaming multiprocessors (SM), and each SM contains multiple processor cores. The clock rate of each core is relatively lower than that of a CPU core. In our experiment, we used the GeForce GTX580 which belongs to the Fermi architecture. The GTX580 accommodates 16 SMs, each of which consists of 32 processor cores operating in the clock rate 1,544 MHz, as presented in Figure 1. Hence, the total number of processor cores is 512.

One can program the GPU with a high-level programming language. We write programs in CUDA C that supports the CUDA programming with a minimal set of extensions to the C language. In the rest of this section, we will describe the key features of the CUDA that we must take into account for programming.

Thread Hierarchy One of the key abstractions of the CUDA is a hierarchy of threads. By this abstraction, we can divide the whole problem into coarse-grained subproblems, *blocks*, which can be solved independently in parallel. A block can be further partitioned into fine-grained subproblems that can also be solved in parallel within the block. This fine-grained subproblem unit is called a *thread*. CUDA's hierarchy of threads maps to a hierarchy of processors on the GPU. An SM executes one or more blocks, and CUDA cores in the SM execute threads.

Scheduling & Branch The way threads are scheduled in GPUs is somewhat different from that in CPUs. The unit of thread scheduling in SMs is a *warp* which is a collection of 32 threads.

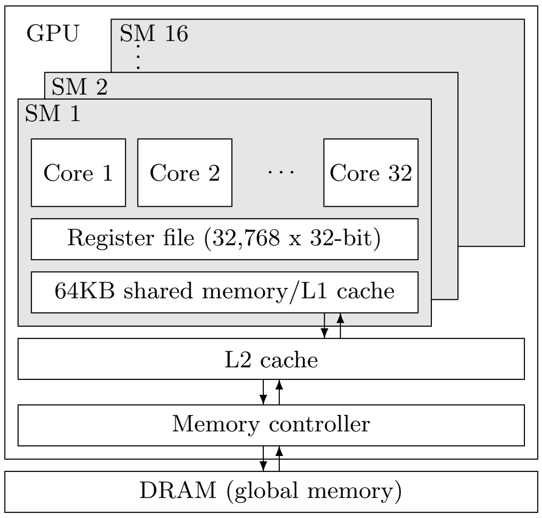

Fig. 1. Fermi architecture

Basically, all the threads within a single warp execute the same instruction at the same time. However, multiple threads of the same warp may execute serially. When they meet any flow control instruction such as `if A else B`, they could take different execution paths. Then, different execution paths within a warp are serialized. It is called *warp serialization* [5, 6], which will slow down the overall performance.

Memory The physically separated place where CUDA threads are executed is referred to as *device*, which includes the GPU. The *host* is where the C program runs, and this includes the CPU. The host and device have their own memory address space. The data to be processed are firstly loaded on the host memory and then copied to the device memory, so that threads running on the GPU can access the data. The processed data on the device needs to be copied back to the host memory after the execution.

The device memory has a hierarchy and it consists of registers, shared memory, caches and global memory. Registers are the fastest on-chip memory and the GTX580 contains about $32K$ registers for each stream multiprocessor. The global memory resides in the off-chip DRAM on the graphics board. It has the longest access latency but has the largest space.

3 Rainbow method

In this section, we summarize the rainbow method [25]. Let g be a one-way function from $\mathcal{N}$ to $\mathcal{H}$ and R_i be a reduction function from $\mathcal{H}$ to $\mathcal{N}$. The function f_i, defined by $f_i(x) = R_i(g(x))$, maps $\mathcal{N}$ into $\mathcal{N}$, where $|\mathcal{N}| = N$.

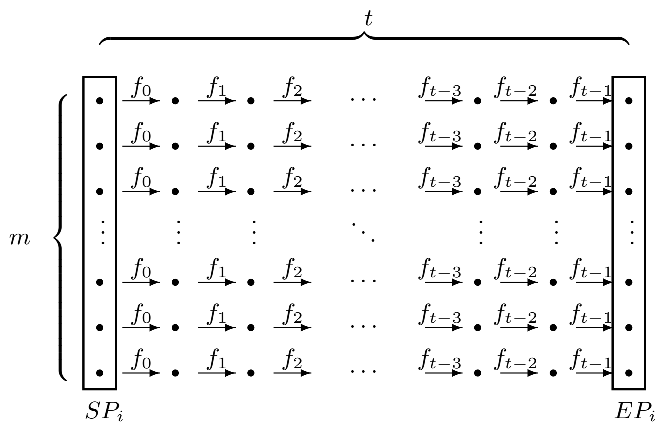

Fig. 2. A rainbow table

The rainbow method consists of two phases: precomputation and online phases. In the precomputation phase, we randomly choose m start points in $\mathcal{N}$, labeled SP_0, SP_1, ... , SP_{m-1}. For each $0 \leq i < m$, we set

$$x_{i,0} = SP_i,$$

and compute

$$x_{i,j} = f_{j-1}(x_{i,j-1}), 1 \leq j \leq t$$

recursively. In other words, m chains of length t are produced starting from SP_i ($0 \leq i < m$) as shown in Figure 2. The last element $x_{i,t}$ for each i-th chain is called an end point (EP_i). The pairs of the start and end points, (SP_i, EP_i), are stored in a table, and they are sorted with respect to the end points. Note that all intermediate points are discarded to reduce memory requirements.

In the online phase, given an image $y_0 = g(x_0)$, we try to invert the one-way function $g(\cdot)$ to find the pre-image x_0, by generating online chains that start from y_0.

At the first iteration, the online chain of length one is generated by computing $y_1 = R_{t-1}(y_0) = f_{t-1}(x_0)$, and we check whether it is an end point on the table by conducting a binary search. If $y_1 = EP_i$ for some i, which is referred to as an *alarm*, it means that x_0 is next to EP_i in Figure 2 or EP_i has more than one inverse images. The latter case is referred to as a *false alarm*. Therefore, we regenerate a chain starting from SP_i to compute $x_{i,t-1}$, and check whether it is a false alarm or not by computing $g(x_{i,t-1}) = y_0$. If $g(x_{i,t-1}) = y_0$, we find the pre-image x_0, which is equal to $x_{i,t-1}$, and the online phase stops. If $y_1 \neq EP_i$ or a false alarm occurred, then we compute $y_2 = f_{t-1}(R_{t-2}(y_0))$, the online chain of length two, and check whether it is an end point. The above process is repeated until x_0 is found or all t online chains fail to invert the given image y_0.

The online phase of the rainbow method can be divided into three parts: *online chain*, *lookup* and *regenerating chain*. The *online chain* procedure generates the online chain of length i at the i-th iteration. The *lookup* procedure checks whether each of these is an end point (alarm) through a binary search in the rainbow table. The *regenerating chain* procedure regenerates the chains of length $(t-i)$, starting from start points for resolving alarms. Because table lookup time through a binary search is negligible in comparison to the one-way function invocation time, the one-way function invocation is the dominant factor in the overall cost of the rainbow method.

Note that the rainbow method is a probabilistic algorithm. That is, success is not guaranteed and the success probability depends on the time and memory allocated for cryptanalysis. If the pre-image x_0 that we want to find exists in the rainbow table, the rainbow method will succeed in finding it; Otherwise, it will fail. The success probability can be computed by the equation presented in [19, 25]. In the case of failure, the online phase generates t online chains, and it carries out t lookups. Also, it regenerates some chains starting from start points whenever alarms occur in the lookup procedure. On the other hand, if the rainbow method succeeds in finding the pre-image x_0, it immediately stops in the middle of the online phase.

4 Implementation in a heterogeneous GPU+CPU system

In this section, we describe our implementations in a heterogeneous GPU+CPU system. Using both GPU and CPU, we implement the rainbow method in parallel. The key factors for achieving good performance are: (i) eliminating the warp serialization by splitting the online phase of the rainbow method, and (ii) load balancing between GPU and CPU using checkpoints.

Before explaining our implementations, we first present the table used in our experiment. Cryptographic hash algorithm SHA-1 was used as the one-way function. We assumed that our table is used for cracking passwords which consist of lowercase, uppercase alphabets (a-z, A-Z) and numbers (0-9), and their lengths are shorter than or equal to 7. That is, $N = 62 + 62^2 + \cdots + 62^7 \approx 3.58 \times 10^{12} \approx 2^{41.7}$. We created a single non-perfect[4] rainbow table with 70% success probability, in which $m = 80,530,636$, $t = 73,403$. For reasons of efficient memory access, a start point of $\lceil \log_2 m \rceil = 27$ bits is stored in a 32-bit data type, `uint32_t`, and an end point of $\lceil \log_2 N \rceil = 42$ bits[5] is stored in a 64-bit data type, `uint64_t`. Thus, the total size of the table is about 0.9 GB. We conducted our experiments on an Intel i7 2.8GHz quad-core CPU and a GTX580 1544MHz 512-core GPU. We used Microsoft Visual Studio 2008 environment on Window 7.

The naive implementation of the parallel rainbow method is that each thread generates the corresponding online chain in parallel. That is, the i-th thread $(1 \le i \le t)$ generates the online chain of length i (the online chain procedure), and it checks whether an alarm occurs (the lookup procedure). If an alarm occurs, the i-th thread regenerates the chain of length $(t-i)$ and it checks whether the element in the $(t-i)$-th column is x_0 or a false alarm (the regenerating chain

[4] None of the colliding chains in the rainbow table are removed.

[5] For the simple implementation, efficient storage techniques [19] such as the index file and the end point truncation were not considered.

procedure). We created 896 threads per SM, i.e., total $896 \times 16 = 14,336$ threads. Thus, at first, threads generate the online chains whose lengths are between 1 and 14,336, and some of them in which alarms occur regenerate the chains and check whether each of these is a success or a false alarm. If some SM finishes its workload, the next 896 online chains, whose lengths are between 14,337 and 15,232, are assigned to the SM. We call this implementation *the Naive GPU.*

Table 1 shows the execution time when it fails to find a pre-image. The second row represents the time for executing all three procedures, and the third row represents the time for executing the online chain and the lookup procedures excluding the regenerating chain procedure. The third column in the table represents the total length of the chains generated in the online chain and regenerating chain procedures.

Table 1. Time of online phase when it fails

procedures	time	chain length
online chain+lookup+regenerating chain	258 sec	4.2×10^9
online chain+lookup	13 sec	2.7×10^9

Generally, the sum of the chain lengths in the regenerating chain procedure is smaller than that of the lengths in the online chain procedure, because alarms occur only in some of the online chains. [17] As can be seen in Table 1, the sum of chain lengths in the online chain procedure (2.7×10^9) is larger than that in the regenerating chain (1.5×10^9). However, the regenerating chain procedure takes much more time than the online chain procedure in the Naive GPU. This is because of *warp serialization.* Since alarms occur in some of the 32 threads within a warp, only these threads regenerate chains for resolving alarms. Thus, the other threads within a warp should wait until the threads finish the regenerating chain procedure. We should eliminate the warp serialization to improve the performance.

To solve this problem (warp serialization), we split the online phase of the rainbow method into the online chain+lookup procedures (A) and the regenerating chain procedure (B). A is processed in the GPU, and B is processed in the CPU, as in Figure 3. Each thread in the GPU (i) generates the online chain assigned to itself and (ii) checks whether it is an end point (alarm). (iii) If an alarm occurs, the number and the length of the corresponding chain are copied to the alarm table in the host memory. At the same time, (i) the threads in the CPU check whether the values copied from the GPU exist in the alarm table. (ii) If so, they read the copied values and (iii) regenerate chains for resolving alarms. By doing this, we can eliminate the warp serialization that occured in the Naive GPU. We call this implementation *the GPU+CPU.*

The execution time of the GPU+CPU is shown in Table 2. The GPU processes A in 13 seconds, whereas on the CPU it takes 102 seconds to process B. While the workload on the GPU is heavier than that on the CPU, the com-

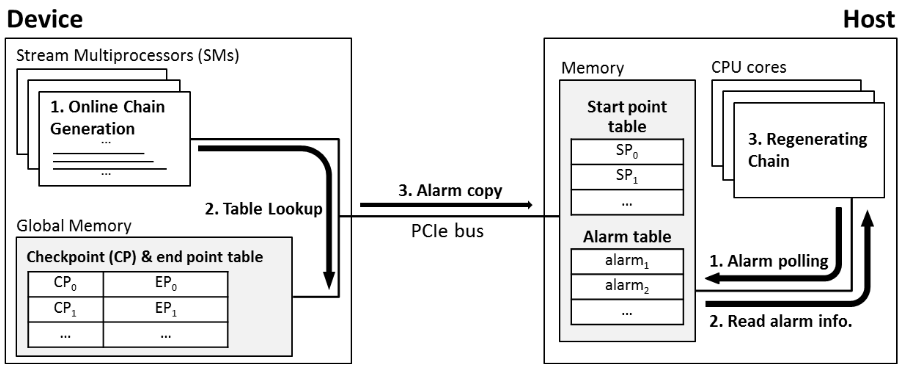

Fig. 3. Implementation in a heterogeneous GPU+CPU system

puting power of the GPU is much better than that of the CPU. Therefore, it is necessary to reduce the workload on the CPU for the efficient GPU+CPU implementation.

Table 2. Time of online phase when it fails

online chain+lookup (GPU)	regenerating chain (CPU)	total
13 sec	102 sec	102 sec

We take advantage of checkpoints [7] for load balancing between GPU and CPU. By decreasing the number of false alarms with checkpoints, we can reduce the workload on the CPU. The more checkpoints we use, the less workload the CPU have to process. In the following section, we analyze the performance improvement using checkpoints and their optimal positions. In Section 6, we present the experimental results when the checkpoints are applied to the GPU+CPU.

5 Checkpoints

By using checkpoints [7], we can reduce the time for the regenerating chain procedure. We store not only the start and end points of the chains in the table but also the information of some intermediate points, i.e., *checkpoints*. The least significant bits of the intermediate points are usually stored. Using the information, we can detect false alarms in advance without regenerating the chains starting from start points. If alarms occur, we compare the information stored in the table with those of the online chain for each checkpoint. If they differ at least for one checkpoint, we know for certain that this is a false alarm. In [7], Avoine *et al.* analyzed the effect of checkpoints for the perfect rainbow table. Analysis for the non-perfect rainbow table was done only for one checkpoint in

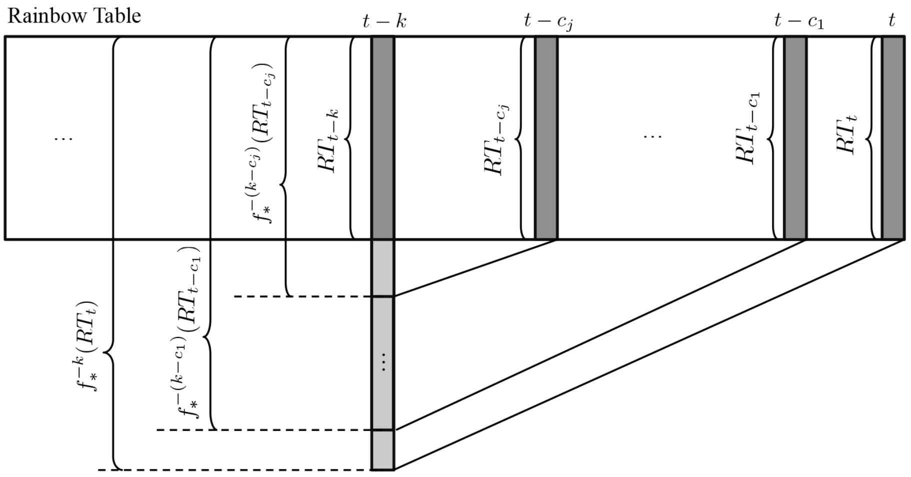

Fig. 4. Sizes of the pre-images at the $(t-k)$-th column

[17]. In this section, we analyze the performance improvement of checkpoints and their optimal positions when multiple checkpoints are used for the non-perfect rainbow table.

The set of elements in the k-th column of the rainbow table is denoted by RT_k. Let $c_1, c_2, \ldots, c_n$ $(c_1 < c_2 < \cdots < c_n)$ be the positions of n 1-bit checkpoints. That is, n checkpoints are located at $(t-c_j)$-th columns of the table for $j = 1, \ldots, n$.

First, at the k-th iteration (an online chain of length k is generated) for $k \leq c_1$, the checkpoints cannot filter out false alarms. Thus, we assume that an alarm is observed at the k-th iteration such that $c_j < k \leq c_{j+1}$ for $j = 1, \ldots, n$, where $c_{n+1} = t$. This means that the pre-image x_0 is in $f_\star^{-k}(RT_t)$, where $f_\star$ is function f_j whose index j is not explicitly specified and $f_\star^{-k}(RT_t)$ is the set of pre-images under $f_\star^k (= f_\star \circ \cdots \circ f_\star)$ of the end points RT_t. As can be seen in Figure 4, the following relations hold:

$$RT_{t-k} \subset f_\star^{-(k-c_j)}(RT_{t-c_j}) \subset \cdots \subset f_\star^{-(k-c_1)}(RT_{t-c_1}) \subset f_\star^{-k}(RT_t).$$

We compute the probability of false alarms when checkpoints are used. If $x_0 \in RT_{t-k}$, x_0 can be found. If $x_0 \in f_\star^{-(k-c_j)}(RT_{t-c_j}) \setminus RT_{t-k}$ (Figure 5), a false alarm always occurs. It is because the online chain starting from x_0 is merged with an precomputed chain in the rainbow table before the $(t-c_j)$-th column, and j checkpoints are thus useless in detecting false alarms. If $x_0 \in f_\star^{-(k-c_u)}(RT_{t-c_u}) \setminus f_\star^{-(k-c_{u+1})}(RT_{t-c_{u+1}})$ for $1 \leq u \leq j-1$ (Figure 6), this means that the online chain is merged with an chain in the table between c_u and c_{u+1}. Hence, a false alarm occurs with probability $1/2^{j-u}$ by $(j-u)$ 1-bit checkpoints, i.e., $c_{u+1}, \ldots, c_j$. Finally, if $x_0 \in f_\star^{-k}(RT_t) \setminus f_\star^{-(k-c_1)}(RT_{t-c_1})$ (Figure 7), a false alarm occurs with probability $1/2^j$.

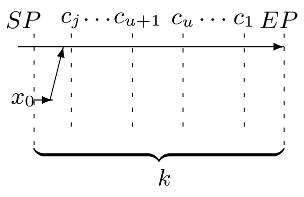

Fig. 5. Merge before c_j

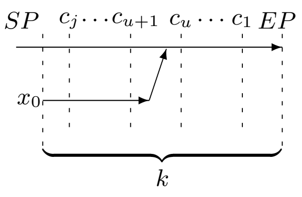

Fig. 6. Merge between c_u and c_{u+1}

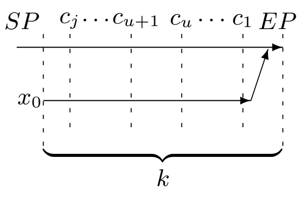

Fig. 7. Merge after c_1

We now compute the improvement in the number of $f_\star$ applications due to checkpoints. Let $z_* = |RT_{t-k}|$, $z_0 = |f_\star^{-k}(RT_t)|$, and $z_j = |f_\star^{-(k-c_j)}(RT_{t-c_j})|$ for $j = 1, \ldots, n$. The probability that $x_0 \in f_\star^{-(k-c_j)}(RT_{t-c_j}) \setminus RT_{t-k}$ is

$$\frac{1}{N}\Big|f_\star^{-(k-c_j)}(RT_{t-c_j}) \setminus RT_{t-k}\Big| = \frac{1}{N}(z_j - z_*),$$

where N is the size of $\mathcal{N}$. In this case, a false alarm always occurs. The probability that $x_0 \in f_\star^{-(k-c_u)}(RT_{t-c_u}) \setminus f_\star^{-(k-c_{u+1})}(RT_{t-c_{u+1}})$ is

$$\frac{1}{N}\Big|f_\star^{-(k-c_u)}(RT_{t-c_u}) \setminus f_\star^{-(k-c_{u+1})}(RT_{t-c_{u+1}})\Big| = \frac{1}{N}(z_u - z_{u+1}).$$

In this case, a false alarm occurs with probability $1/2^{j-u}$. The probability that $x_0 \in f_\star^{-k}(RT_t) \setminus f_\star^{-(k-c_1)}(RT_{t-c_1})$ is

$$\frac{1}{N}\Big|f_\star^{-k}(RT_t) \setminus f_\star^{-(k-c_1)}(RT_{t-c_1})\Big| = \frac{1}{N}(z_0 - z_1)$$

In this case, a false alarm occurs with probability $1/2^j$. Therefore, the expected number of false alarms at the k-th iteration such that $c_j < k \le c_{j+1}$ ($j = 1, \ldots, n$) can be written as

$$\frac{1}{N}\Big\{(z_j - z_*) + \sum_{u=0}^{j-1} \frac{1}{2^{j-u}}(z_u - z_{u+1})\Big\}. \tag{1}$$

Also, the expected number of false alarms at the k-th iteration without checkpoints is

$$\frac{1}{N}(z_0 - z_*). \tag{2}$$

Hence, the expected decreasing number of false alarms at the k-th iteration due to checkpoints is (2) − (1), which simplifies to

$$\frac{1}{N}\Big\{(1 - \frac{1}{2^j})z_0 - \sum_{u=0}^{j-1} \frac{1}{2^{j-u}} z_{u+1}\Big\}. \tag{3}$$

According to Propositions 4 and 5 in [17], $z_0 \approx m(1+k), z_u \approx m(1+k-c_u)$. Simplification of (3) using these approximations results in

$$\frac{m}{N}\sum_{u=0}^{j-1}\left(\frac{c_{u+1}}{2^{j-u}}\right).$$

We shall write $D(j)$ for this. At the k-th iteration such that $c_j < k \leq c_{j+1}$, the expected decreasing number of false alarms due to checkpoints is $D(j)$ and the number of $f_\star$ applications for checking false alarms is $t-k+1$[6]. The probability that the k-th iteration is processed is equal to the probability to fail until the $(k-1)$-th iteration. This probability is

$$\prod_{i=1}^{k-1}(1-\frac{m_{t-i}}{N}),$$

where m_j denotes the distinct number of elements in the j-th column of the rainbow table, i.e., $m_j \approx \frac{N}{N/m+j/2}$ [7]. Therefore, the expected number of $f_\star$ applications that can be removed through n 1-bit checkpoints is

$$\sum_{j=1}^{n}\Big\{\sum_{c_j<k\leq c_{j+1}}(t-k+1)\cdot D(j)\cdot\prod_{i=1}^{k-1}\big(1-\frac{m_{t-i}}{N}\big)\Big\},$$

where $c_{n+1} = t$.

Table 3 shows the performance improvement due to the checkpoints and the optimal positions of those, where $N = 3.58 \times 10^{12}$, $m = 80,530,636$, and $t = 73,403$. The optimal positions represent the ratio from the rightmost column of the table. We used Maple 12 [3] to obtain these positions. The number of $f_\star$ applications in the regenerating chain procedure without checkpoints can be calculated from Theorem 3 of [17].

We made use of 22 1-bit checkpoints. Because $N = 3.58 \times 10^{12} \approx 2^{41.7}$, we used uint64_t, which is the data type of 64 bits, to store an end point, as mentioned in Section 4. An end point was stored in the lower 42 bits, and 22 1-bit checkpoints were stored in the upper 22 bits which remained empty. Therefore, no additional memory is needed to store the checkpoints. The 22 checkpoints are expected to decrease the number of $f_\star$ applications due to false alarms by about 84%. The optimal positions of 22 checkpoints are 0.0416, 0.0633, 0.0855, 0.1083, 0.1316, 0.1556, 0.1802, 0.2056, 0.2318, 0.2589, 0.2870, 0.3162, 0.3465, 0.3783, 0.4117, 0.4470, 0.4845, 0.5247, 0.5684, 0.6168, 0.6718, and 0.7381.

6 Experimental Results

In this paper, we introduced three different kinds of implementations using the GPU: naive GPU, GPU+CPU and GPU+CPU with checkpoints. Figure 8 also

[6] Strictly speaking, one extra g application follows $(t-k)$ number of $f_\star$ applications in order to check false alarms.

Table 3. Expected numbers of $f_\star$ applications (unit: t^2) in the regenerating chain procedure and performance improvement due to checkpoints at the optimal positions.

# of checkpoints	1	2	3	4	5	6	7	8
# of $f_\star$ applications without checkpoints[†] (1)	0.1676							
Reduced # of $f_\star$ applications with checkpoints[†] (2)	0.0354	0.0577	0.0732	0.0847	0.0936	0.1008	0.1066	0.1115
Improvement ((2)/(1))	21.1%	34.4%	43.7%	50.5%	55.9%	60.1%	63.6%	66.5%
Optimal positions	0.2792	0.2123 0.3591	0.1732 0.2827 0.4179	0.1470 0.2356 0.3379 0.4637	0.1281 0.2028 0.2863 0.3826 0.5008	0.1136 0.1785 0.2495 0.3287 0.4199 0.5317	0.1022 0.1597 0.2216 0.2894 0.3649 0.4517 0.5579	0.0930 0.1446 0.1996 0.2590 0.3239 0.3962 0.4792 0.5806

[†] The $f_\star$ applications in the online chain procedure are not included.

shows the experimental results using the CPU, as well as those of the three implementations presented in this paper. In the case of the CPU, the i-th thread generates the online chain of length i and regenerates the chain of length $(t - i)$ from a start point if an alarm occurs, as in the naive GPU. We used the i7 quad-core CPU for our experiment. Every experiment was carried out 50 times, and numerical values in the figure represent the average times for searching a pre-image.

There are several GPU-accelerated implementations of the rainbow method: RainbowCrack [2] and Cryptohaze [1]. The overall performance of the rainbow method depends on the implementations of the one-way function and the reduction function. Because their source codes are not publicly available, however, direct comparisons are not possible. Also, our work does not focus on the optimized implementations of these functions. Thus, we show the advantage of ours through an indirect comparison with RainbowCrack and Cryptohaze. Assume that the implementations of the one-way function and the reduction function are the same. Then, the online chain procedures of all implementations will take more or less the same time, since the parallelization of the online chain procedure is straightforward. The time for the lookup procedure is negligible compared to the other two procedures. We regard the time for the online chain procedure as 100%, and measure the total time as its percentage. According to our experiments, RainbowCrack and Cryptohaze take about 56% and 158% time for the regenerating chain procedure, respectively. Therefore, RainbowCrack and Cryptohaze take about 156% and 258% for the total time, since they regenerate chains for resolving false alarms after the online chain and lookup procedures are finished. However, our method requires no more time on average for the regenerating chain procedure because the online chain+lookup procedures and

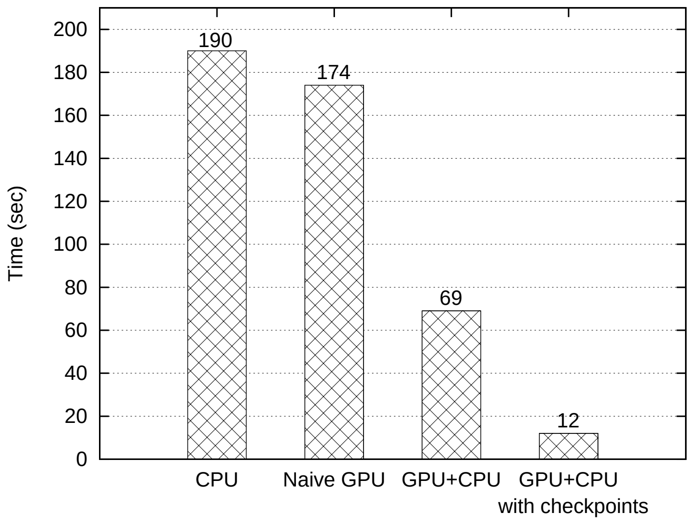

Fig. 8. Timings for searching a pre-image. Each bar represents the average time for the whole 50 experiments.

the regenerating chain procedure are simultaneously executed in GPU and CPU. Our GPU+CPU with checkpoints (12 seconds) actually finishes earlier on average than the worst case of the online chain+lookup procedures (13 seconds). That is, our implementation takes about 92% on average for the total time.

References

1. Cryptohaze gpu rainbow cracker, `https://www.cryptohaze.com`.
2. RainbowCrack Project, `http://project-rainbowcrack.com`.
3. Maplesoft, Maple 12 user manual, 2007.
4. Nvidia, Nvidia's next generation CUDA compute architecture: Fermi, 2009.
5. Nvidia, CUDA best practices guide, 2012.
6. Nvidia, CUDA C programming guide, 2012.
7. G. Avoine, P. Junod, and P. Oechslin. Characterization and improvement of time-memory trade-off based on perfect tables. *ACM Trans. Inf. Syst. Secur.*, 11(4), 2008.
8. E. Barkan, E. Biham, and A. Shamir. Rigorous bounds on cryptanalytic time/memory tradeoffs. In *CRYPTO*, pages 1–21, 2006.
9. D. J. Bernstein, T.-R. Chen, C.-M. Cheng, T. Lange, and B.-Y. Yang. ECM on graphics cards. In *EUROCRYPT*, pages 483–501, 2009.
10. A. Biryukov, S. Mukhopadhyay, and P. Sarkar. Improved time-memory trade-offs with multiple data. In *Selected Areas in Cryptography*, pages 110–127, 2005.
11. J. Borst, B. Preneel, and J. Vandewalle. On the time-memory tradeoff between exhaustive key search and table precomputation. In *Proc. of the 19th Symposium in Information Theory in the Benelux, WIC*, pages 111–118, 1998.

12. A. R. Brodtkorb, C. Dyken, T. R. Hagen, J. M. Hjelmervik, and O. O. Storaasli. State-of-the-art in heterogeneous computing. *Scientific Programming*, 18(1):1–33, 2010.
13. D. E. Denning. *Cryptography and Data Security.* Addison-Wesley, 1982. p.100.
14. A. Fiat and M. Naor. Rigorous time/space trade-offs for inverting functions. *SIAM J. Comput.*, 29(3):790–803, 1999.
15. M. Hellman. A cryptanalytic time-memory trade-off. *Information Theory, IEEE Transactions on*, 26(4):401 – 406, July 1980.
16. J. Hermans, F. Vercauteren, and B. Preneel. Speed records for NTRU. In *CT-RSA*, pages 73–88, 2010.
17. J. Hong. The cost of false alarms in Hellman and rainbow tradeoffs. *Des. Codes Cryptography*, 57(3):293–327, 2010.
18. J. Hong, G. W. Lee, and D. Ma. Analysis of the parallel distinguished point tradeoff. In *INDOCRYPT*, pages 161–180, 2011.
19. J. Hong and S. Moon. A comparison of cryptanalytic tradeoff algorithms. *Journal of Cryptology.* To appear.
20. J. Hong and P. Sarkar. New applications of time memory data tradeoffs. In *ASIACRYPT*, pages 353–372, 2005.
21. K. Kusuda and T. Matsumoto. Optimization of time-memory trade-off cryptanalysis and its application to DES, FEAL-32 and Skipjack. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E79-A(1):35–48, 1996.
22. S. A. Manavski. CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In *ICSPC*, 2007.
23. S. Mukhopadhyay and P. Sarkar. Application of LFSRs in time/memory trade-off cryptanalysis. In *WISA*, pages 25–37, 2006.
24. J. Nickolls and W. J. Dally. The GPU computing era. *IEEE Micro*, 30(2):56–69, 2010.
25. P. Oechslin. Making a faster cryptanalytic time-memory trade-off. In *CRYPTO*, pages 617–630, 2003.
26. F.-X. Standaert, G. Rouvroy, J.-J. Quisquater, and J.-D. Legat. A time-memory tradeoff using distinguished points: New analysis & FPGA results. In *CHES*, pages 593–609, 2002.
27. R. Szerwinski and T. Güneysu. Exploiting the power of GPUs for asymmetric cryptography. In *CHES*, pages 79–99, 2008.
28. W. Wang, D. Lin, Z. Li, and T. Wang. Improvement and analysis of VDP method in time/memory tradeoff applications. In *ICICS*, pages 282–296, 2011.