SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2013; **00**:1–18
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

High-Speed Parallel Implementations of the Rainbow Method Based on Perfect Tables 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 151-747, Korea*
[2]*Department of Mathematical Sciences and ISaC, Seoul National University, Seoul 151-747, Korea*
[3]*Division of Internet and Media, Konkuk University, Seoul 143-701, Korea*

SUMMARY

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 based on perfect tables, 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. For GTX460, our implementation is about 1.86 and 3.25 times faster than other GPU-accelerated implementations, RainbowCrack and Cryptohaze, respectively, and for GTX580, 1.53 and 2.40 times faster. Copyright © 2013 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: GPGPU; CUDA; Heterogeneous Computing; Cryptanalysis; Cryptanalytic Time-Memory Tradeoff; Rainbow Method

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 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 [2]. 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 [3]. In 2008, Szerwinski and Güneysu made use of CUDA for GPGPU processing of asymmetric cryptosystems (RSA, DSA, ECC) [4]. In 2009, Bernstein et al. showed that GPU can be used for cryptanalysis as well as implementation of

[*]Correspondence to: Kunsoo Park, Department of Computer Science and Engineering and Institute of Computer Technology, Seoul National University, Seoul 151-747, Korea. E-mail: kpark@theory.snu.ac.kr
[†]This article shares much of its material with our previous work [1], presented at INDOCRYPT 2012. However, this work treats the perfect table case, whereas the previous work covered the non-perfect table case.

cryptographic algorithms [5]. They implemented the elliptic-curve method for integer factorization on GPUs. In 2010, NTRU cryptosystem was implemented on CUDA by Hermans et al. [6].

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 [7, 8, 9, 10, 11, 12, 13, 14, 15] are compromise solutions between time and memory. Cryptanalytic time-memory tradeoff was introduced by Hellman in 1980 [16]. Rivest proposed to apply *distinguished points* [17] to Hellman's method which reduce the number of table lookup operations. Borst et al. suggested to use a *perfect* table [18] where no merging chain exists. It is more efficient time-memory tradeoff than the non-perfect table, although more precomputation effort is required. In 2003, a new method, which is referred to as the *rainbow method*, was suggested by Oechslin [19]. 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* [20]. Using the technique, the cost of false alarms is reduced with a minute amount of memory.

The rainbow method has been used widely in practice for cracking passwords, and there are some executable files publicly available [21, 22, 23]. Among these, RainbowCrack [23] and Cryptohaze [21] provide GPU-accelerated implementations of the rainbow method, and they are significantly faster than any other implementations on CPU.

With the rise of GPGPU, commodity computers are complex heterogeneous GPU +CPU systems that provide high computational power [24, 25]. 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 based on perfect tables 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 perfect rainbow table, and take advantage of it for load balancing between GPU and CPU. We compare the performance of our implementation with those of RainbowCrack and Cryptohaze in two platforms (GTX460 and GTX580). For GTX460, the proposed implementation is about 1.86 and 3.25 times faster than RainbowCrack and Cryptohaze, respectively, and for GTX580, 1.53 and 2.40 times faster. To the best of our knowledge, this is the fastest implementation of the rainbow method so far.

The rest of the paper is organized as follows. We begin with preliminaries including an overview of modern GPUs and a brief review of the rainbow method in Section 2. In Section 3, we analyze the checkpoint technique. In Section 4, we describe our fast implementations in a heterogeneous GPU+CPU system. Finally, Section 5 compares our implementation with other implementations on GPU, and Section 6 concludes the paper.

2. PRELIMINARIES

2.1. 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 [2] 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

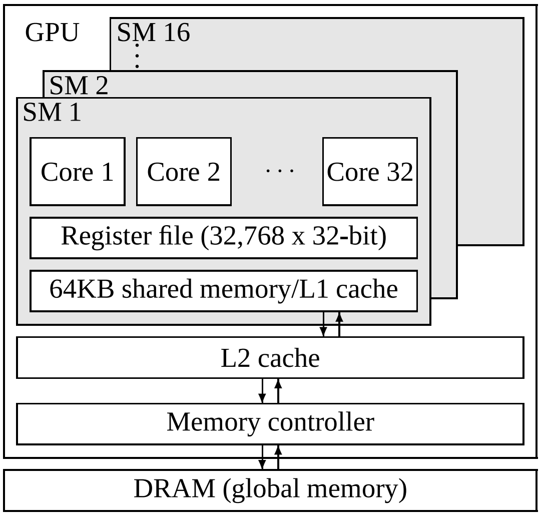

Figure 1. Fermi architecture

versions of architectures since its first architecture, G80, and the newest one is called Fermi [26], 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. For example, the GeForce 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.

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* [27, 2], 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.

DOI: 10.1002/spe

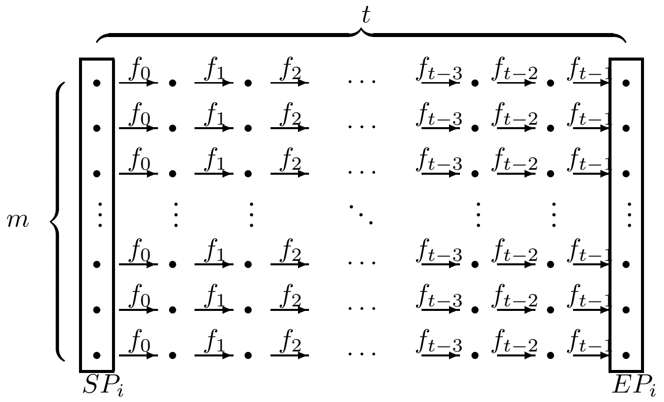

Figure 2. A rainbow table

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.

2.2. Rainbow Method

In this section, we summarize the rainbow method [19]. 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 $|\mathcal{N}| = N$.

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, $\dots$, 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. To make the table *perfect*, only one chain among the chains that have same end points is stored; the rest of chains are removed. In a perfect table, therefore, all end points are distinct.

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*. For $1 \leq k \leq t$, the *online chain* procedure generates the online chain of length k. 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 $(t - k)$, starting from start points for resolving alarms. Because table lookup time through a binary search is

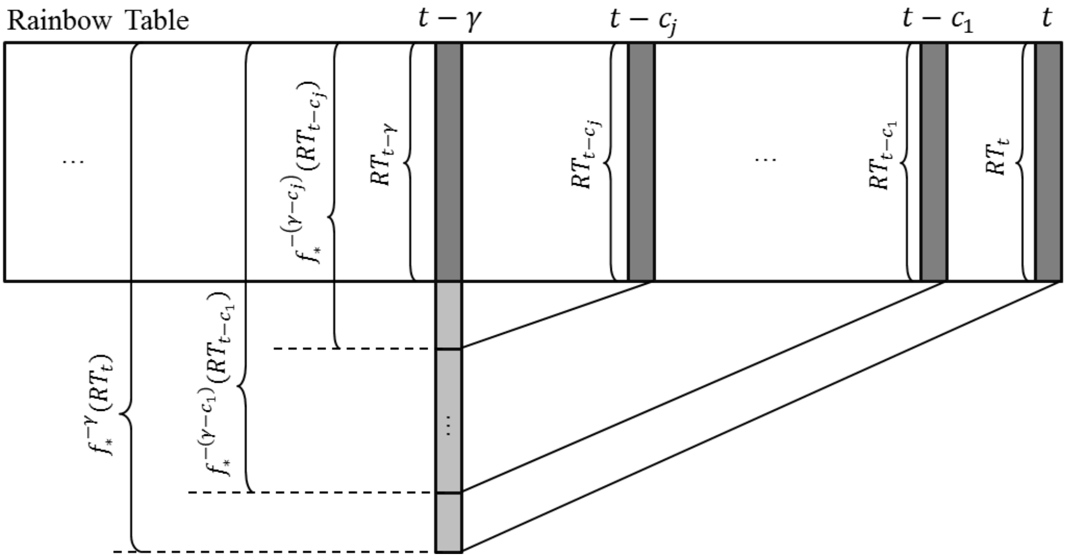

Figure 3. Sizes of the pre-images of end points at the $(t-\gamma)$-th column

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 [28, 19]. 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.

3. CHECKPOINTS

By using checkpoints [20], 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 [20], Avoine et al. analyzed the effect of checkpoints for the *maximal* perfect rainbow table. Hong [29] took an approach different from that of [20]. In [29], the effect of multiple checkpoints is analyzed for Hellman's table, but for the rainbow table only a single checkpoint is analyzed. In this section, we extend the analysis of [29] into multiple checkpoints for the perfect rainbow table. We also analyze the performance improvements for three cases in terms of the order of online chain generation. These results are used for efficient implementations in Section 4.

The set of elements in the j-th column of the rainbow table is denoted by RT_j, whose size is m for all $1 \leq j \leq t$ in the perfect table. 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 $j = 1, \ldots, n$. If an online chain of length γ is generated such that $\gamma \leq c_1$, the checkpoints cannot filter out false alarms. Thus, we assume that an alarm is observed when an online chain of length γ is generated such that $c_j < \gamma \leq c_{j+1}$ for $j = 1, \ldots, n$, where $c_{n+1} = t$. This means that the pre-image x_0 is in $f_\star^{-\gamma}(RT_t)$, where $f_\star$ is function f_j whose index j is not explicitly specified and $f_\star^{-\gamma}(RT_t)$ is the

Copyright © 2013 John Wiley & Sons, Ltd.
Prepared using speauth.cls

DOI: 10.1002/spe

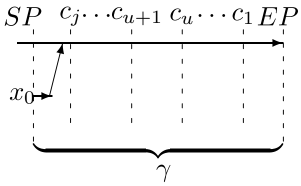

Figure 4. Merge before c_j

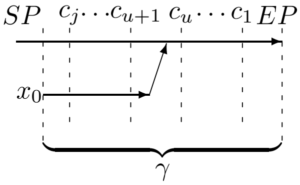

Figure 5. Merge between c_u and c_{u+1}

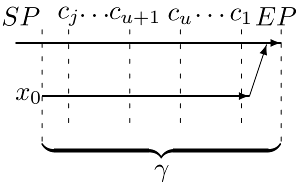

Figure 6. Merge after c_1

set of pre-images under $f_\star^\gamma (= f_\star \circ \cdots \circ f_\star)$ of the end points RT_t. As can be seen in Figure 3, the following relations hold:

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

Lemma 1

Let $z_{0,\gamma} = |f_\star^{-\gamma}(RT_t)|$ and $z_{j,\gamma} = |f_\star^{-(\gamma-c_j)}(RT_{t-c_j})|$ for $j = 1, \ldots, n$. The expected decreasing number of false alarms due to checkpoints when an online chain of length γ such that $c_j < \gamma \leq c_{j+1}$ is generated is

$$D(j,\gamma) = \frac{1}{N}\Big\{(1-\frac{1}{2^j})z_{0,\gamma} - \sum_{u=0}^{j-1}\frac{1}{2^{j-u}}z_{u+1,\gamma}\Big\},$$

where $z_{0,\gamma} \approx m(1+\gamma)(1-\frac{m\gamma}{4N})$ and $z_{u,\gamma} \approx m(1+\gamma-c_u) + \frac{(\gamma-c_u)(\gamma-c_u+2)}{c_u^2}\{m \cdot c_u + 2N\ln(1-\frac{m \cdot c_u}{2N})\}$.

Proof

We compute the cost of false alarms when checkpoints are used. For $\forall x_0 \in f_\star^{-(\gamma-c_j)}(RT_{t-c_j}) \setminus RT_{t-\gamma}$ (Figure 4), 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. For $\forall x_0 \in f_\star^{-(\gamma-c_u)}(RT_{t-c_u}) \setminus f_\star^{-(\gamma-c_{u+1})}(RT_{t-c_{u+1}})$ for $1 \leq u \leq j-1$ (Figure 5), 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, for $\forall x_0 \in f_\star^{-\gamma}(RT_t) \setminus f_\star^{-(\gamma-c_1)}(RT_{t-c_1})$ (Figure 6), a false alarm occurs with probability $1/2^j$.

We now compute the improvement in the number of $f_\star$ applications due to checkpoints. The expected number of false alarms without checkpoints when $x_0 \in f_\star^{-(\gamma-c_j)}(RT_{t-c_j}) \setminus RT_{t-\gamma}$ is

$$\frac{1}{N}\Big|f_\star^{-(\gamma-c_j)}(RT_{t-c_j}) \setminus RT_{t-\gamma}\Big| = \frac{1}{N}(z_{j,\gamma} - m),$$

where N is the size of $\mathcal{N}$. In this case, a false alarm always occurs. The expected number of false alarms without checkpoints when $x_0 \in f_\star^{-(\gamma-c_u)}\ (RT_{t-c_u}) \setminus f_\star^{-(\gamma-c_{u+1})}(RT_{t-c_{u+1}})$ is

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

In this case, the $(j-u)$ checkpoints cannot filter out false alarms with probability $1/2^{j-u}$. The expected number false alarms without checkpoints when $x_0 \in f_\star^{-\gamma}(RT_t) \setminus f_\star^{-(\gamma-c_1)}(RT_{t-c_1})$ is

$$\frac{1}{N}\Big|f_\star^{-\gamma}(RT_t) \setminus f_\star^{-(\gamma-c_1)}(RT_{t-c_1})\Big| = \frac{1}{N}(z_{0,\gamma} - z_{1,\gamma})$$

In this case, the j checkpoints cannot filter out false alarms with probability $1/2^j$. Therefore, the expected number of false alarms when an online chain of length γ is generated such that

$c_j < \gamma \leq c_{j+1}$ $(j = 1, \ldots, n)$ can be written as

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

Also, the expected number of false alarms without checkpoints when an online chain of length γ is generated is

$$\frac{1}{N}(z_{0,\gamma} - m). \tag{2}$$

Hence, the expected decreasing number of false alarms due to checkpoints when an online chain of length γ is generated is $(2) - (1)$, which simplifies to the claimed value. In addition, according to Propositions 4 and 5 in [29], $z_{0,\gamma} \approx m(1+\gamma)(1-\frac{m\gamma}{4N})$, $z_{u,\gamma} \approx m(1+\gamma-c_u) + \frac{(\gamma-c_u)(\gamma-c_u+2)}{c_u^2}\{m \cdot c_u + 2N\ln(1-\frac{m \cdot c_u}{2N})\}$. □

Now, we analyze the performance improvement for three cases in terms of the order of online chain generation: at the k-th iteration, (i) the online chain of length k is generated, i.e., the online chains are generated from shortest to longest; (ii) the online chain of length $(t-k+1)$ is generated, i.e., the online chains are generated from longest to shortest; (iii) for a fixed $1 \leq \alpha \leq t$, if $k \leq \alpha$, the online chain of length k is generated; otherwise, the online chain of length $(t-k+\alpha+1)$ is generated, i.e., the third case is a hybrid of (i) and (ii).

Theorem 2 (**From shortest to longest online chains**)
Assume that the online chain of length k is generated at the k-th iteration. The expected number of $f_\star$ applications that can be removed through n 1-bit checkpoints is

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

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

Proof
At the k-th iteration, the online chain of length k is generated, i.e., $\gamma = k$. Hence, for $c_j < \gamma = k \leq c_{j+1}$, the expected decreasing number of false alarms due to checkpoints is $D(j,k)$ and the number of $f_\star$ applications for checking false alarms is $t-k+1$[†]. 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 $\big(1 - \frac{m}{N}\big)^{k-1}$. Therefore, we obtain the claimed value. □

Theorem 3 (**From longest to shortest online chains**)
Assume that the online chain of length $(t-k+1)$ is generated at the k-th iteration. The expected number of $f_\star$ applications that can be removed through n 1-bit checkpoints is

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

Proof
At the k-th iteration, the online chain of length $(t-k+1)$ is generated, i.e., $\gamma = t-k+1$. Hence, for $c_j < \gamma \leq c_{j+1}$,i.e., $t+1-c_{j+1} \leq k < t+1-c_j$, the expected decreasing number of false alarms due to checkpoints is $D(j, t-k+1)$ and the number of $f_\star$ applications for checking false alarms is k. The probability that the k-th iteration is processed is $\big(1 - \frac{m}{N}\big)^{k-1}$. Therefore, we obtain the claimed value. □

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

Theorem 4 (**Hybrid**)
Assume that, for a fixed $1 \leq \alpha \leq t$, the online chain of length k is generated if $k \leq \alpha$ and otherwise that of length $(t-k+\alpha+1)$ at the k-th iteration. The expected number of $f_\star$ applications that can be removed through n 1-bit checkpoints is

$$\sum_{j=1}^{n} \Big\{ \sum_{\substack{c_j < k \leq c_{j+1} \\ 1 \leq k \leq \alpha}} (t-k+1) \cdot D(j,k) \cdot \Big(1 - \frac{m}{N}\Big)^{k-1} + \sum_{\substack{t+\alpha+1-c_{j+1} \leq k < t+\alpha+1-c_j \\ \alpha+1 \leq k \leq t}} (k-\alpha) \cdot D(j, t+\alpha+1-k) \cdot \Big(1 - \frac{m}{N}\Big)^{k-1} \Big\}.$$

Proof
At the k-th iteration such that $1 \leq k \leq \alpha$, the expected number of $f_\star$ applications that can be removed through n 1-bit checkpoints is the same as the first case (from shortest to longtest). At the k-th iteration such that $\alpha + 1 \leq k \leq t$, the online chain of length $(t + \alpha + 1 - k)$ is generated, i.e., $\gamma = t + \alpha + 1 - k$. Hence, for $c_j < \gamma \leq c_{j+1}$, i.e., for $t + \alpha + 1 - c_{j+1} \leq k < t + \alpha + 1 - c_j$, the expected decreasing number of false alarms due to checkpoint is $D(j, t + \alpha + 1 - k)$ and the number of $f_\star$ applications for checking false alarms is $k - \alpha$. The probability that the k-th iteration is processed is $\big(1 - \frac{m}{N}\big)^{k-1}$. Therefore, we obtain the claimed value. □

Tables I and II show the performance improvement due to the checkpoints and the optimal positions of those for three cases (from shortest to longest online chains, from longest to shortest, and a hybrid with $\alpha = 15,360$)[‡], where $N = 3.58 \times 10^{12}$, $m = 80,530,636$, and $t = 71,535$. The optimal positions represent the ratio from the rightmost column of the table. One way to find the optimal positions of checkpoints is to test all possible combinations of the checkpoints and choose the positions that maximize the improvement due to the checkpoints. However, it is too time consuming to test all $\binom{t}{n}$ combinations. We make use of an approximate algorithm. At the first stage, we globally find an approximate solution. Let $s = 5,000$ be an initial interval, and we test all n-combinations of $\{i \cdot s + 1 : i = 0, ..., 14\}$, i.e., the set of points in [1..71,535] starting from 1 with intervals of 5,000. Let p_i for $i = 1, \ldots, n$ be the points we obtain from the first stage. We repeatedly find a more accurate solution near p_i's by reducing the interval. At the next stage, the interval is set to $\frac{s}{2}$ ($s \leftarrow \frac{s}{2}$), and we test the points $p_i \pm j \cdot s$ for $i = 1, \ldots, n$ and $j = 0, 1, 2$, and then update p_i's to the values that maximize the improvement. This process is repeated while $s > 10$. We used our CUDA C program to calculate the improvements in parallel. We verified that the approximate solution was the same as the exact solution computed by all possible combinations for $n = 1, 2, \ldots, 7$ (and $t = 71,535$).

The number of $f_\star$ applications in the regenerating chain procedure without checkpoints for the first case (from shortest to longest) can be calculated from Theorem 2 of [29], and those for the other cases (from longest to shortest and hybrid) can be calculated from the following theorems. These theorems can be easily obtained in a way similar to Theorem 2 of [29].

Theorem 5 (**From longest to shortest online chains)**
Assume that the online chain of length $(t-k+1)$ is generated at the k-th iteration. The number of $f_\star$ applications in the regenerating chain procedure without checkpoints is

$$\sum_{k=1}^{t} k \cdot \frac{m}{N}(t-k+2)\Big\{1 - \frac{m(t-k+1)}{4N}\Big\} \cdot \Big(1 - \frac{m}{N}\Big)^{k-1}.$$

Theorem 6 (**Hybrid)**
Assume that, for a fixed $1 \leq \alpha \leq t$, the online chain of length k is generated if $k \leq \alpha$ and otherwise

[‡] The reason that α is set as $15,360$ will be explained in Section 4.

DOI: 10.1002/spe

Table I. Expected numbers of $f_\star$ applications (unit: t^2) in the regenerating chain procedure and performance improvement due to checkpoints at the optimal positions. (STL: from shortest to longest online chains, LTS: from longest to shortest online chains)

# of checkpoints		1	2	3	4	5	6	7
STL	# of $f_\star$ applications without checkpoints (1)	0.1062						
	Reduced # of $f_\star$ applications with checkpoints (2)	0.0198	0.0327	0.0419	0.0490	0.0545	0.0591	0.0628
	Improvement ((2)/(1))	18.6%	30.8%	39.5%	46.1%	51.3%	55.6%	59.1%
	Optimal positions	0.2412	0.1809 0.3188	0.1472 0.2471 0.3767	0.1250 0.2048 0.2998 0.4224	0.1091 0.1760 0.2525 0.3433 0.4599	0.0970 0.1548 0.2193 0.2929 0.3800 0.4915	0.0874 0.1384 0.1944 0.2566 0.3275 0.4114 0.5184
LTS	# of $f_\star$ applications without checkpoints (3)	0.0981						
	Reduced # of $f_\star$ applications with checkpoints (4)	0.0210	0.0342	0.0433	0.0501	0.0553	0.0594	0.0628
	Improvement ((4)/(3))	21.4%	34.9%	44.1%	51.1%	56.4%	60.6%	64.0%
	Optimal positions	0.3994	0.3165 0.4902	0.2647 0.4036 0.5522	0.2284 0.3459 0.4669 0.5980	0.2013 0.3038 0.4077 0.5157 0.6335	0.1804 0.2715 0.3632 0.4567 0.5546 0.6621	0.1635 0.2457 0.3280 0.4113 0.4967 0.5865 0.6855
Hybrid ($\alpha = 15,360$)	# of $f_\star$ applications without checkpoints (5)	0.0882						
	Reduced # of $f_\star$ applications with checkpoints (6)	0.0149	0.0249	0.0328	0.0384	0.0434	0.0470	0.0505
	Improvement ((6)/(5))	16.9%	28.2%	37.2%	43.5%	49.2%	53.3%	57.3%
	Optimal positions	0.3994	0.1443 0.4389	0.1316 0.3583 0.5199	0.1036 0.1662 0.3856 0.5395	0.0974 0.1557 0.3352 0.4587 0.5920	0.0812 0.1279 0.1786 0.3552 0.4744 0.6034	0.0777 0.1221 0.1702 0.3197 0.4208 0.5262 0.6412

that of length $(t - k + \alpha + 1)$ at the k-th iteration. The number of $f_\star$ applications in the regenerating

Table II. Reduced numbers of $f_\star$ applications (unit: t^2) in the regenerating chain procedure when 22 1-bit checkpoints are used.

	Reduced #	Improvement	Optimal positions
STL	0.0861	81.1%	0.0363, 0.0555, 0.0754, 0.0957, 0.1167, 0.1385, 0.1609, 0.1843, 0.2084, 0.2334, 0.2596, 0.2871, 0.3159, 0.3463, 0.3785, 0.4128, 0.4496, 0.4895, 0.5334, 0.5826, 0.6396, 0.7102
LTS	0.0826	84.2%	0.0692, 0.1036, 0.1379, 0.1719, 0.2058, 0.2396, 0.2733, 0.3070, 0.3406, 0.3743, 0.4081, 0.4421, 0.4763, 0.5109, 0.5459, 0.5816, 0.6180, 0.6555, 0.6946, 0.7359, 0.7804, 0.8308
Hybrid ($\alpha = 15,360$)	0.0707	80.2%	0.0318, 0.0483, 0.0655, 0.0830, 0.1011, 0.1195, 0.1385, 0.1580, 0.1782, 0.1990, 0.2661, 0.3102, 0.3542, 0.3985, 0.4430, 0.4880, 0.5338, 0.5808, 0.6293, 0.6801, 0.7346, 0.7961

chain procedure without checkpoints is

$$\sum_{k=1}^{\alpha}(t-k+1)\cdot\frac{m}{N}(1+k)(1-\frac{mk}{4N})\cdot\Big(1-\frac{m}{N}\Big)^{k-1}+ \sum_{k=\alpha+1}^{t}(k-\alpha)\cdot\frac{m}{N}(t-k+\alpha+2)\Big\{1-\frac{m(t-k+\alpha+2)}{4N}\Big\}\cdot\Big(1-\frac{m}{N}\Big)^{k-1}.$$

Note that the required numbers of $f_\star$ applications in the regenerating chain procedure without checkpoints vary with the orders of online chain generation. (According to Table I, $0.1062t^2$, $0.0981t^2$, and $0.0882t^2$ for STL, LTS, and hybrid, respectively) Also, the effect of checkpoints are similar regardless of the orders of online chain generation. As a result, for LTS and hybrid, the expected total lengths of the chains generated in the regenerating chain procedure with 22 checkpoints are reduced by about 23% ($= \frac{(0.1062-0.0861)-(0.0981-0.0826)}{(0.1062-0.0861)}$) and 13% ($= \frac{(0.1062-0.0861)-(0.0882-0.0707)}{(0.1062-0.0861)}$), respectively, compared to STL.

Table III shows the effects of LTS and Hybrid over STL in the regenerating chain procedure without checkpoints. The effects are measured for various values of m and N. For hybrid, we select the optimal α that maximizes the improvement. The ratio of the optimal position of α from the rightmost column is 0.1676 for all values of m and N that we tested[§]. As can be seen from the table, the effects do not depend on m and N. The cost of LTS and Hybrid in the regenerating chain procedure are reduced by about 7.6% and 17.6% compared to STL for all m and N, respectively. Therefore, changing the order would be effective for all m and N.

4. IMPLEMENTATION IN A HETEROGENEOUS GPU+CPU SYSTEM

In this section, we describe our implementations of the perfect rainbow method 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, (ii) load balancing between GPU and CPU using checkpoints, (iii) changing the order of the online chain generation, and (iv) fully parallelizing the rainbow method by reinvoking GPU for resolving false alarms.

[§] The value of α can be easily computed by using Maple [30].

Table III. Effects of LTS and Hybrid over STL in the regenerating chain procedure without checkpoints for various m and N, where $m_0 = 80,530,636$.

m	Order	N				
		2^{40}	2^{45}	2^{50}	2^{55}	2^{60}
$m_0/4$	LTS	7.656%	7.656%	7.656%	7.656%	7.656%
	Hybrid	17.612%	17.612%	17.612%	17.612%	17.612%
m_0	LTS	7.656%	7.656%	7.656%	7.656%	7.656%
	Hybrid	17.612%	17.612%	17.612%	17.612%	17.612%
$4m_0$	LTS	7.656%	7.656%	7.656%	7.656%	7.656%
	Hybrid	17.612%	17.612%	17.612%	17.612%	17.612%
$8m_0$	LTS	7.656%	7.656%	7.656%	7.656%	7.656%
	Hybrid	17.612%	17.612%	17.612%	17.612%	17.612%

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 intend to create a single perfect rainbow table with 80% success probability with $m = 80,530,636$ and $t = 71,535$. According to [31], we can make such a table with $m_0 = 412,383,272$ precomputed chains. After removing the chains, except only one, that have same end points in the precomputation phase, $80,529,164$ chains among the m_0 chains actually remained. As a result, we used a perfect table of $m = 80,529,164$ and $t = 71,535$ in our experiment. For reasons of efficient memory access, a start point of $\lceil \log_2 m_0 \rceil = 29$ bits is stored in a 32-bit data type, uint32_t, and an end point of $\lceil \log_2 N \rceil = 42$ bits[¶] is stored in a 64-bit data type, uint64_t. Thus, the total size of the table is about 0.9 GB. Throughout this section, we conducted our experiments on two Intel Xeon E5506 2.13GHz quad-core CPUs (8 cores in total) 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 ($1 \leq i \leq t$) 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 procedure). We created 640 threads per SM, i.e., total $640 \times 16 = 10,240$ threads. Thus, at first, threads generate the online chains whose lengths are between 1 and 10,240, 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 640 online chains, whose lengths are between 10,241 and 10,880, are assigned to the SM. We call this implementation *the Naive GPU*.

Table IV 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 IV. Time of the online phase when it fails

procedures	time	chain length
online chain+lookup+regenerating chain	96.6 sec	3.6×10^9
online chain+lookup	5.6 sec	2.6×10^9

[¶] For the simple implementation, efficient storage techniques [28] such as the index file and the end point truncation were not considered.

DOI: 10.1002/spe

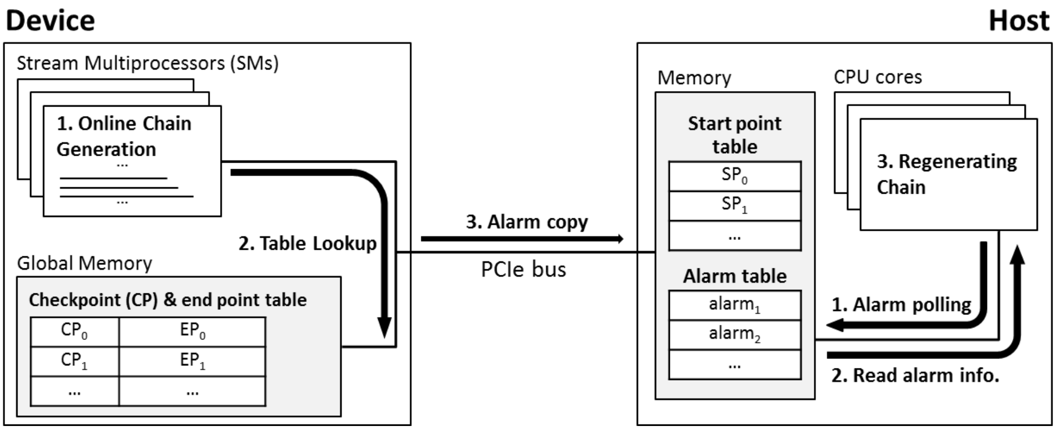

Figure 7. Implementation in a heterogeneous GPU+CPU system

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. [29] As can be seen in Table IV, the sum of chain lengths in the online chain procedure (2.6×10^9) is larger than that in the regenerating chain (1.0×10^9). 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.

GPU+CPU. 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 7. 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 occurred in the Naive GPU. We call this implementation *the GPU+CPU*.

Table V. Time of the online phase when it fails

online chain+lookup (GPU)	regenerating chain (CPU)	total
5.6 sec	52 sec	52 sec

The execution time of the GPU+CPU is shown in Table V. The GPU processes A in 5.6 seconds, whereas on the CPU it takes 52 seconds to process B. While the workload on the GPU is heavier than that on the CPU, the computing 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.

Load Balancing Through Checkpoints. We take advantage of checkpoints [20] 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. 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 above. 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. When the online chains are

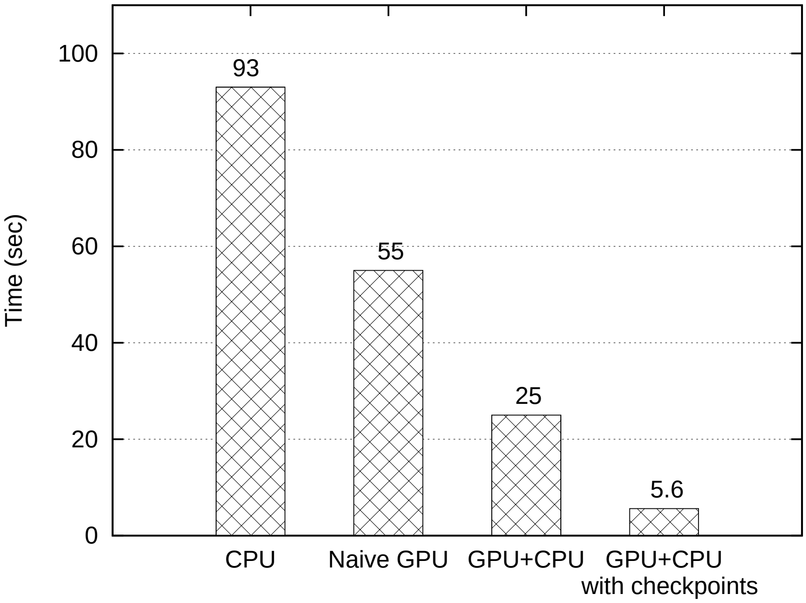

Figure 8. Timings of searching for a pre-image. Each bar represents the average time for the whole 50 experiments.

generated from shortest to longest, the 22 checkpoints are expected to decrease the number of $f_\star$ applications due to false alarms by about 81.1% and their optimal positions are in Table II.

So far, we introduced three different kinds of implementations using the GPU: naive GPU, GPU+CPU and GPU+CPU with checkpoints. Figure 8 also 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 two Intel Xeon E5506 CPUs 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. As can be seen from the figure, the GPU+CPU with checkpoints is 17 times faster than the CPU and 9.8 times faster than the naive GPU. Also, the GPU+CPU with checkpoints is 4.5 times faster than the GPU+CPU.

Order of Online Chain Generation. We can further improve the performance by changing the order of online chain generation. Generally, it is efficient to generate the online chains from shortest to longest. However, it is not true in the GPU+CPU implementations. Because the computing power of the CPU is much worse than that of the GPU, it is important to reduce the workload on the CPU, i.e., the number of $f_\star$ applications in the regenerating chain procedure.

Figure 9 shows the expected number of $f_\star$ applications in the regenerating chain procedure with respect to the length of an online chain when 22 1-bit checkpoints are applied. The cost of the regenerating chain procedure is $\{\frac{m}{N}(1+\gamma)(1-\frac{m\gamma}{4N}) - D(j,\gamma)\} \cdot (t-\gamma+1)$ when the online chain of length γ such that $c_j < \gamma \leq c_{j+1}$ is generated. In Figure 9, some decreasing steps occur at the positions of checkpoints, and there is a clear trend of decreasing cost as the length of the online chain increases. Therefore, in order to reduce the expected length of chains created in the regenerating chain procedure, it should generate the online chains from longest to shortest. As explained in Section 3, the expected total length of the chains generated in the regenerating chain procedure is reduced by about 23%, compared to STL.

Figure 10 shows the average times of the GPU+CPU with checkpoints to search for a pre-image in three cases in terms of the order of online chain generation. Every case uses its optimal checkpoints calculated in Table II. The first and the second bars represent the average times only for success and failure cases, respectively. The third bar represents the average time for all 50 experiments. Contrary to our expectations, the GPU+CPU with checkpoints from longest to shortest online

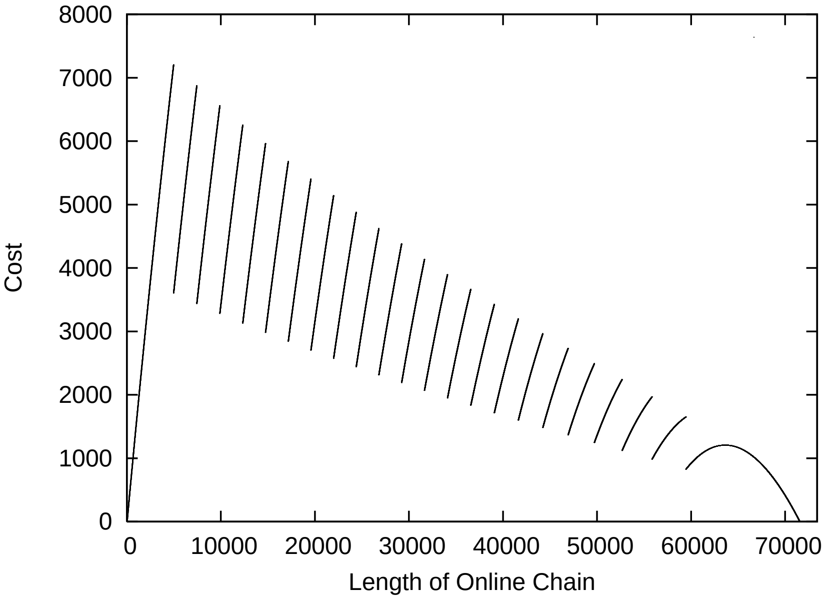

Figure 9. The expected number of $f_{\star}$ applications in the regenerating chain procedure when 22 1-bit checkpoints for LTS are applied.

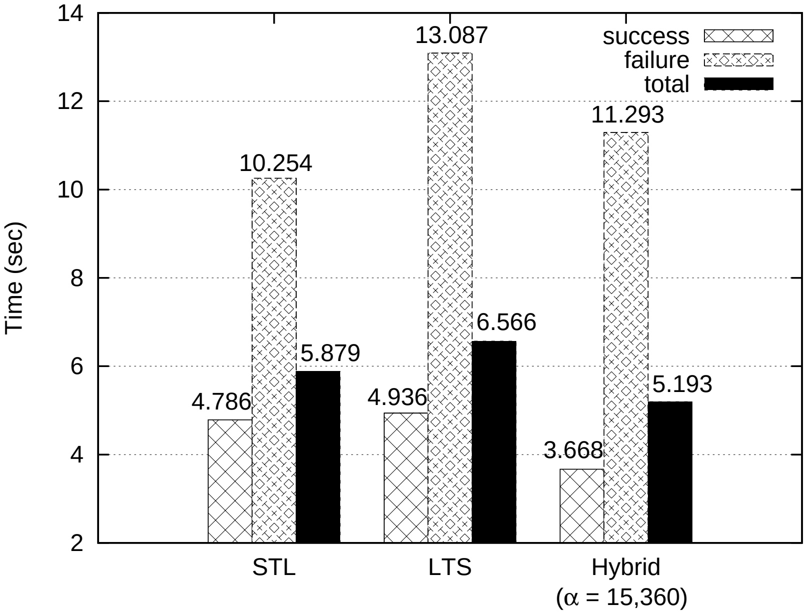

Figure 10. Average time of the GPU+CPU with 22 1-bit checkpoints to search for a pre-image in 50 experiments. (STL: from shortest to longest online chains, LTS: from longest to shortest online chains)

chains is slightly slower than that from shortest to longest online chains. It is owing to the long start-up time of the regenerating chain procedure on the CPU (i.e., the time until the first alarm occurs and the alarm information is copied to the alarm table in the host memory). In the case that online chains are generated from longest to shortest, it takes long time to generate the first online chain. However, if we generate online chains from shortest to longest, a large number of online chains are quickly generated at the same time in the GPU, and a sufficient number of alarms occur not to make the CPU idle. In GPU, thousands of online chains could be generated in parallel. Therefore, the implementation from longest to shortest online chains takes longer start-up time than

DOI: 10.1002/spe

that from shortest to longest online chains. We can infer from the average times of the failure case in Figure 10 that the start-up time of the implementation from longest to shortest online chains is about $13.087 - 10.254 = 2.833$ seconds longer than that of the implementation from shortest to longest online chains.

The best solution (hybrid) is to combine the two ordering ways above in order to reduce both the start-up time and the workload on the CPU. At the k-th iteration, if $k \leq \alpha$ for a fixed $1 \leq \alpha \leq t$, we generate online chains from shortest to longest in order to reduce the start-up time of CPU; otherwise, we generate online chains from longest to shortest in order to reduce the workload on CPU. That is, if $k \leq \alpha$, the online chain of length k is generated; otherwise, the online chain of length $(t - k + \alpha + 1)$ is generated. By setting α as a large integer, we can reduce the start-up time, but the workload on CPU increases. Hence, it is important to choose appropriate value α to balance the start-up time and the workload on CPU. We empirically found out $\alpha = 15,360$ to minimize the average time of searching for a pre-image. As explained in Section 3, the expected total length of the chains generated in the regenerating chain procedure is reduced by about 13%, compared with the implementation from shortest to longest online chains. Figure 10 shows that the hybrid with $\alpha = 15,360$ improves the performance of the GPU+CPU with checkpoints by about 13% by simply changing the order of online chain generation.

Reinvoking GPU. In the GPU+CPU, the GPU generates online chains and delivers their alarm information to the CPU; the CPU regenerates chains using the alarm information received from the GPU. Hence, the GPU usually finishes earlier than the CPU. Table VI shows the average time of the GPU+CPU with checkpoints in the hybrid order. As can be seen from Table VI, the GPU finishes 1.174 seconds earlier than the CPU on average, and the GPU becomes idle for this time. Therefore, to achieve the best performance, it is required to fully exploit the computation power of the GPU.

Table VI. Average time of the online phase for 50 experiments(Hybrid with $\alpha = 15,360$)

online chain+lookup (GPU)	regenerating chain (CPU)	total
4.019 sec	5.193 sec	5.193 sec

We reinvoke the GPU for resolving false alarms, i.e., the GPU and the CPU could regenerate chains together. If the regenerating chain procedure on the CPU is not finished yet after the online chain+lookup procedure on the GPU is finished, the GPU reads the alarm information from the alarm table and regenerates chains for resolving false alarms. Since the CPU is also regenerating chains at this moment, the GPU reads the alarm table in the reverse order of the CPU. Each thread in the GPU first reads each of the last 3,840 entries of the alarm table and regenerates chains for resolving false alarms with the threads in the CPU.

Table VII shows the average time of the implementation, reinvoking GPU. From the table, we can see that the time for the regenerating chain procedure is reduced by about 10%, and so is the total time.

5. COMPARISON

In this section, we compare the performance of ours with those of other GPU-accelerated implementations. There are several implementations of the rainbow method publicly available now [21, 22, 23]. Ophcrack [22] provides only a CPU-accelerated implementation, whereas RainbowCrack [23] and Cryptohaze [21] provide not only a CPU-accelerated implementation but also a GPU-accelerated one. As the implementations on GPU are much faster than the

Table VII. Average time of online phase for 50 experiments (Reinvoking GPU)

online chain+lookup	regenerating chain	total
4.059 sec	4.665 sec	4.665 sec

DOI: 10.1002/spe

implementations on CPU, we compare ours with only GPU-accelerated ones of RainbowCrack and Cryptohaze. The source codes of RainbowCrack and Cryptohaze are not publicly available (only their executable files are available), and thus we can just run their executable files for making the perfect tables and cracking the target images.

Table VIII. Tradeoff parameters and size (GB) of the perfect tables

	RainbowCrack	Cryptohaze	Ours
m_0	412,383,272	562,383,272	412,383,272
m	80,532,743	80,555,916	80,529,164
t	71,535	71,535	71,535
Size (GB)	1.2	1.6	0.9
Success Prob.	80.00%	80.01%	80.00%

We created the perfect rainbow tables for each implementation: RainbowCrack, Cryptohaze, and ours. Table VIII shows the tradeoff parameters and the table size in GB. To make the perfect table of 80% success probability for $N = 3.58 \times 10^{12}$, in which $m = 80,530,636$ and $t = 71,535$, we created m_0 precomputed chains of length $71,535$ for the three implementations, where $m_0 = 412,383,272$ for RainbowCrack and ours but $m_0 = 562,383,272$ for Cryptohaze. About 1.4 times more precomputed chains for Cryptohaze were generated to make the perfect table with m distinct end points, i.e., the cost of the precomputation phase of Cryptohaze is 1.4 times larger than those of RainbowCrack and ours. The success probabilities of the three perfect tables are approximately equal to 80%. The size of the perfect table of Cryptohaze is the biggest among the three implementations. The checkpoint technique is not available in RainbowCrack and Cryptohaze, and our implementation uses the table of 0.9 GB including checkpoints as explained in Section 4.

Table IX. Specifications of GTX460 and GTX580

	GTX460	GTX580
Clock rate (Mhz)	1,430	1,544
# of SM	7	16
# of cores per SM	48	32
Total # of cores	336	512

We tested the three implementations using two different GPUs: GTX460 and GTX580. The specifications of these GPUs are shown in Table IX. For example, the GTX580 accommodates 16 SMs, each of which consists of 32 cores operating in the clock rate 1,544 MHz. The GTX580 has better performance than the GTX460. With each of these GPUs, we used two Intel Xeon E5506 2.13 GHz quad-core CPUs.

Table X shows the timings of RainbowCrack, Cryptohaze and ours of searching for a pre-image. We randomly generated 200 input images and executed the three implementations using these input images. The 80.5%, 80.5% and 80.0% out of the input images were actually succeeded for RainbowCrack, Cryptohaze and ours, respectively. The timings were measured as an average value over 200 trials. The second to the fourth rows represent the timings over two Xeon CPUs and GTX460. The fifth to the seventh rows represent the timings over two Xeon CPUs and GTX580. RainbowCrack and Cryptohaze regenerate chains on GPU for resolving false alarms after the online chain and lookup procedures are finished, whereas the online chain+lookup procedures and the regenerating chain procedure are simultaneously executed in GPU and CPU in our implementation. Hence, the total time of RainbowCrack and Cryptohaze is the sum of the times for the online chain+lookup and the regenerating chain procedures, but our total time is equal to the maximum of the two. As a result, for GTX460, our implementation is about 1.86 and 3.25 times faster than RainbowCrack and Cryptohaze, respectively, and for GTX580, 1.53 and 2.40 times faster. Our implementation fully exploits GPU and CPU but the others take advantage of only GPU. Hence, the better the relative performance of CPU to GPU is, the better is our implementation, compared

DOI: 10.1002/spe

Table X. Timings of searching for a pre-image. (sec)

GPU	implementation	online chain+lookup	regenerating chain	total
GTX460	RainbowCrack	11.181	2.027 (0.383)	13.208 (11.564)
	Cryptohaze	10.505	12.525 (2.367)	23.030 (12.872)
	Ours	6.602	7.091	7.091
GTX580	RainbowCrack	6.268	1.118 (0.211)	7.386 (6.479)
	Cryptohaze	5.445	6.170 (1.166)	11.615 (6.611)
	Ours	4.238	4.832	4.832

to the others. The values in parentheses are hypothetic timings of RainbowCrack and Cryptohaze, assuming that the 22 checkpoints for STL are applied and thus the cost for the regenerating chain procedure is reduced by 81.1%. Our implementation would be better than the other implementations even under this assumption (for GTX460, 1.63 and 1.82 times better than RainbowCrack and Cryptohaze, respectively).

6. CONCLUSION

In this paper, we proposed the parallel implementations of the rainbow method based on perfect tables in a GPU+CPU heterogeneous system. For achieving the best performance, we first split the online phase into two procedures: the online chain+lookup procedure and the regenerating chain procedure. Second, we gave a complete analysis of the effect of multiple checkpoints for the perfect rainbow table, and we made use of it for load balancing between GPU and CPU. Third, we changed the order of the online chain generation for the heterogeneous system. Finally, we fully exploited a GPU+CPU heterogeneous system by reinvoking GPU for resolving false alarms. According to our experimental result, our implementation is faster than any other implementations on GPU.

ACKNOWLEDGEMENTS

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No. 20120006492) and the Basic Science Research Program through NRF funded by MEST (2012R1A1B4003379).

REFERENCES

1. Kim JW, Seo J, Hong J, Park K, Kim SR. High-speed parallel implementations of the rainbow method in a heterogeneous system. *INDOCRYPT*, 2012; 303–316.
2. Nvidia, CUDA C programming guide 2012.
3. Manavski SA. CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. *ICSPC*, 2007.
4. Szerwinski R, Güneysu T. Exploiting the power of GPUs for asymmetric cryptography. *CHES*, 2008; 79–99.
5. Bernstein DJ, Chen TR, Cheng CM, Lange T, Yang BY. ECM on graphics cards. *EUROCRYPT*, 2009; 483–501.
6. Hermans J, Vercauteren F, Preneel B. Speed records for NTRU. *CT-RSA*, 2010; 73–88.
7. Barkan E, Biham E, Shamir A. Rigorous bounds on cryptanalytic time/memory tradeoffs. *CRYPTO*, 2006; 1–21.
8. Biryukov A, Mukhopadhyay S, Sarkar P. Improved time-memory trade-offs with multiple data. *Selected Areas in Cryptography*, 2005; 110–127.
9. Fiat A, Naor M. Rigorous time/space trade-offs for inverting functions. *SIAM J. Comput.* 1999; **29**(3):790–803.
10. Kusuda K, Matsumoto T. 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* 1996; **E79-A**(1):35–48.
11. Standaert FX, Rouvroy G, Quisquater JJ, Legat JD. A time-memory tradeoff using distinguished points: New analysis & FPGA results. *CHES*, 2002; 593–609.
12. Hong J, Sarkar P. New applications of time memory data tradeoffs. *ASIACRYPT*, 2005; 353–372.
13. Mukhopadhyay S, Sarkar P. Application of LFSRs in time/memory trade-off cryptanalysis. *WISA*, 2006; 25–37.
14. Wang W, Lin D, Li Z, Wang T. Improvement and analysis of VDP method in time/memory tradeoff applications. *ICICS*, 2011; 282–296.
15. Hong J, Lee GW, Ma D. Analysis of the parallel distinguished point tradeoff. *INDOCRYPT*, 2011; 161–180.
16. Hellman M. A cryptanalytic time-memory trade-off. *Information Theory, IEEE Transactions on* Jul 1980; **26**(4):401 – 406, doi:10.1109/TIT.1980.1056220.

17. Denning DE. *Cryptography and Data Security*. Addison-Wesley, 1982. P.100.
18. Borst J, Preneel B, Vandewalle J. On the time-memory tradeoff between exhaustive key search and table precomputation. *Proc. of the 19th Symposium in Information Theory in the Benelux, WIC*, 1998; 111–118.
19. Oechslin P. Making a faster cryptanalytic time-memory trade-off. *CRYPTO*, 2003; 617–630.
20. Avoine G, Junod P, Oechslin P. Characterization and improvement of time-memory trade-off based on perfect tables. *ACM Trans. Inf. Syst. Secur.* 2008; **11**(4).
21. Cryptohaze gpu rainbow cracker, `https://www.cryptohaze.com`. [25 January 2013].
22. Ophcrack, `http://ophcrack.sourceforge.net`. [25 January 2013].
23. RainbowCrack Project, `http://project-rainbowcrack.com`. [25 January 2013].
24. Nickolls J, Dally WJ. The GPU computing era. *IEEE Micro* 2010; **30**(2):56–69.
25. Brodtkorb AR, Dyken C, Hagen TR, Hjelmervik JM, Storaasli OO. State-of-the-art in heterogeneous computing. *Scientific Programming* 2010; **18**(1):1–33.
26. Nvidia, Nvidia's next generation CUDA compute architecture: Fermi 2009.
27. Nvidia, CUDA best practices guide 2012.
28. Hong J, Moon S. A comparison of cryptanalytic tradeoff algorithms. *Journal of Cryptology* 2012; doi:10.1007/s00145-012-9128-3.
29. Hong J. The cost of false alarms in Hellman and rainbow tradeoffs. *Des. Codes Cryptography* 2010; **57**(3):293–327.
30. Maplesoft, Maple 12 user manual 2007.
31. Lee GW, Hong J. A comparison of perfect table cryptanalytic tradeoff algorithms. *Technical Report*, Cryptology ePrint Archive, Report 2012/540 2012.
32. Warren HS. *Hacker's Delight (2nd Edition)*, chap. Integer division by constants. Addison-Wesley, 2003.

A. OPTIMIZED DIVISION & MODULAR ARITHMETIC

The iterating function f_i in our implementation consists of miscellaneous procedures such as the reduction function R_i. In such procedures other than the one-way function g, a number of division and modular arithmetic operations on 64-bit integers are executed. These two operations cause significant performance degradation since their costs are much more expensive than the other simple primitive instructions such as addition and logical operations.

Table XI. Timings of searching for a pre-image in 50 experiments

	Previous version [1]	Reinvoking GPU w/o optimization	Reinvoking GPU w/ optimization
Time	6.599 sec	6.211 sec	4.665 sec

In our current implementation throughout this paper, the division and the modular arithmetic operations are replaced by a couple of operations that consist of addition and logical shift operations. Table XII shows the optimized procedure that performs the division and the modular arithmetic operations for a constant divisor to compute quotient q and remainder r on dividend a. In our implementation, the division and the modular arithmetic operations with divisor 62 are frequently used when the points, which are represented as integers between 0 and $N-1$, are converted to their corresponding passwords, where 62 is the number of symbols (a-z, A-Z, and 0-9). We refer to [32] for the detailed explanation of the optimized procedure. The performance improvement on GTX580 is shown in Table XI. The same perfect table of ours was used, and the average times of searching for a pre-image on the same random images of Section 4 were measured. The second column represents the time of GPU+CPU with checkpoints in hybrid order without optimization. The third and fourth columns represent the time of reinvoking GPU with checkpoints in hybrid order. The optimized procedure reduces the time by about 25%.

Table XII. Optimized integer division and modular arithmetic

Previous version [1]	Optimized one
$q = a/62$ $r = a\%62$	$q = (a >> 5)$ $q = (q >> 5) + q$ $q = (q >> 10) + q$ $q = (q >> 20) + q$ $q = (q >> 20) + q$ $r = ((((((((q << 1) + q) << 1) + q) << 1) + q) << 1) + q) << 1$ $r = a - r$ if $(r >= 124)$ $\{$ $q+ = 2$ $r- = 124$ $\}$ else if $(r >= 62)$ $\{$ $q + +$ $r- = 62$ $\}$

DOI: 10.1002/spe