공학박사 학위논문

GPU 상의 암호 알고리즘 병렬 구현

Parallel Implementations of

Cryptographic Algorithms on GPU

2013년 2월

서울대학교 대학원

전기·컴퓨터 공학부

김 정 우

GPU 상의 암호 알고리즘 병렬 구현

Parallel Implementations of Cryptographic Algorithms on GPU

지도교수 박 근 수

이 논문을 공학박사 학위논문으로 제출함
2012년 11월

서울대학교 대학원
전기·컴퓨터 공학부
김 정 우

김정우의 공학박사 학위논문을 인준함
2012년 12월

위 원 장 _____ (인)

부위원장 _____ (인)

위 원 _____ (인)

위 원 _____ (인)

위 원 _____ (인)

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, 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 thesis, we study parallel implementations of two cryptographic algorithms using GPU. First, 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 is about 1.36 and 2.18 times faster than other GPU-accelerated implementations, RainbowCrack and Cryptohaze, respectively.

Second, we present efficient implementations for NTRU Cryptosystem and their parallel ones. Convolution product is a dominant operation of NTRU, and we can speed up the convolution product computation based on the fact that repeating patterns in coefficients of an NTRU polynomial can be used for construction of look-up tables. We provide efficient convolution algorithms to implement this idea, and we analyze the complexity of the new algorithms using Markov chains. Also, we implemented the new algorithms over a CPU and a GPU. According to our analyses and experimental results, the new algorithms speed up the convolution product by up to 41%, and our GPU-accelerated implementations are more than 23 times faster than the implementations on

CPU.

Keywords: GPGPU, CUDA, Heterogeneous Computing, Cryptanalysis, Cryptanalytic Time-Memory Tradeoff, Rainbow Method, NTRU, convolution product, sliding window method

Student Number: 2006-21173

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

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 [7]. CUDA enables programmers to easily control GPUs by writing programs similar to C.

With the rise of GPGPU, commodity computers are complex heterogeneous GPU+CPU systems that provide high computational power [49, 14]. 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.

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 [45]. In 2008, Szerwinski and Güneysu made use of CUDA for GPGPU processing of asymmetric cryptosystems (RSA, DSA, ECC) [56]. In 2009, Bernstein et al. showed that GPU can be used for cryptanalysis as well as implementation of cryptographic algorithms [11]. They implemented the elliptic-curve method for integer factorization on GPUs. In 2010, NTRU cryptosystem was implemented on CUDA by Hermans et al. [23].

In this thesis, we study parallel implementations of two cryptographic algorithms using GPU. First, we present the efficient implementation of the rainbow method [51], which is known as the most efficient cryptanalytic time-memory tradeoff, in a GPU+CPU heterogeneous system. Second, we present methods to speed up NTRU operations [26] and their parallel implementations on GPU.

1.1.1 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 [7] 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

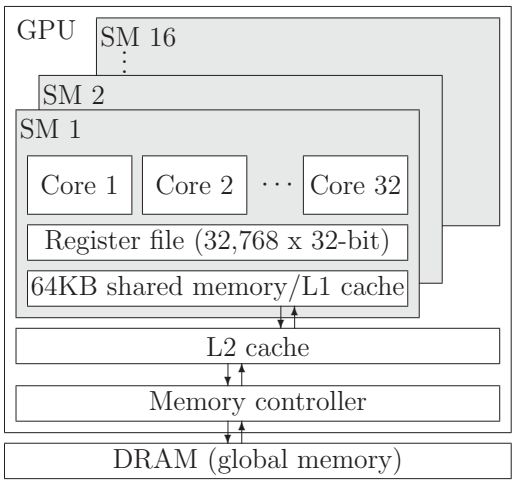

Figure 1.1: Fermi architecture

several improved versions of architectures since its first architecture, G80, and the newest one is called Fermi [5], 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 and GTX275. The specifications of these GPUs are shown in Table 1.1. For example, 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 1.1. 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

Table 1.1: Specifications of GTX580 and GTX275

	GTX580	GTX275
Clock rate (MHz)	1,544	1,404
# of SM	16	30
# of processor cores	32	8
Total # of cores	512	240

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* [6, 7], 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.

1.1.2 Cryptanalytic Time-Memory Tradeoff

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 [10, 12, 13, 17, 40, 55, 32, 47, 57, 30] are compromise solutions between time and memory. Cryptanalytic time-memory tradeoff was introduced by Hellman in 1980 [22]. Rivest proposed to apply *distinguished points* technique [16] 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 [51]. 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* [8]. 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 [1, 2, 3]. Among these, RainbowCrack [3] and Cryptohaze [1] provide GPU-accelerated implementations of the rainbow method, and they are significantly faster than any other implementations on CPU.

1.1.3 NTRU Cryptosystem[1]

The NTRU cryptosystem, or NTRUEncrypt [26], is a public key cryptosystem defined over a quotient ring of polynomials. Compared to existing public key schemes such as RSA and elliptic curve cryptosystems (ECCs), NTRU has several attractive features from the viewpoints of security and performance.

[1]This section is quoted from [41, 42].

Although there have been many attempts to attack NTRU [15, 37, 48, 34, 18, 19, 46], there is no known efficient algorithm to break the lattice problems on which the security of NTRU is based. The complexity of the most recent and best attack method is still exponential in the security parameter [33]. Moreover, the lattice problems are expected to remain hard even if a quantum computer is realized. Note that there are only exponential-time quantum algorithms to break NTRU problems [43], while there are already known polynomial-time quantum algorithms to solve the problems of integer factorization and discrete logarithms [54]. Another attractive feature of NTRU is that encryption and decryption can be done efficiently since they are based only on very simple polynomial arithmetic operations. These operations are believed to be much faster than those of RSA and elliptic curve cryptosystems. Thanks to these desirable characteristics, NTRUEncrypt is now gaining more public acceptance as an alternative public key cryptosystem, and it was standardized by the IEEE P1363 working group [36].

Since NTRU was introduced at the rump session of Crypto 96 [25], extensive research has been done to improve the performance of NTRU further. Hoffstein and Silverman [27, 28] proposed to use special forms of polynomials such as binary polynomials and product-form polynomials to reduce the amount of computations while preserving the security. Bailey et al. [9] showed that NTRU can be efficiently implemented over resource-constrained devices, and Gaubatz, Kaps and Sunar [20] presented an efficient hardware implementation of NTRU using only about 3,000 gates. Recently, Hermans, Vercauteren and Preneel [24] presented a parallelized implementation of NTRU over a GPU [50].

1.2 Contribution

In this thesis, we present parallel implementations for two well-known cryptographic algorithms: the rainbow method [51] and NTRU cryptosystem [26].

1.2.1 Implementation of the Rainbow Method

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 [38]. 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 is about 1.36 and 2.18 times faster than RainbowCrack and Cryptohaze, respectively. To the best of our knowledge, this is the first work implementing the rainbow method in a heterogeneous system.

1.2.2 Implementation of NTRU Cryptosystem

The dominant operation of NTRU is multiplication over a polynomial ring, called a convolution product. We present new methods for accelerating this operation [41, 42][2]. By precomputing the values related to repeated patterns in coefficients of an NTRU polynomial, the amount of computations in the convolution product can be significantly reduced. We prove the usefulness of our methods using Markov chains, and we verify the improvements using soft-

[2] A part of the main idea is a joint work with Mun-Kyu Lee, Jeong Eun Song, and Kunsoo Park.

ware implementations of NTRU for various settings. According to the analysis and experiments, the first and second methods speed up the encryption and decryption operations by up to 34% and 41% on a Pentium IV platform, respectively.[3] Also, we show that our optimization techniques are also useful in parallel processing environments. We implement the basic convolution algorithm and our sliding window method on GPU. According to our experimental results, the processing time for the main convolution operation is reduced by up to 26% by the sliding window method, although the overall speed-up is less than this value due to the bottleneck such as memory copy operations.

1.3 Organization

The rest of this thesis is organized as follows. In Chapter 2, we describe our fast implementations in a heterogeneous GPU+CPU system. In Chapter 3, we present the sliding window methods for NTRU and their implementations on GPU. Finally, we conclude in Chapter 4.

[3]The implementations on a Pentium IV platform were mainly done by Jeong Eun Song.

Chapter 2

Implementation of the Rainbow Method in a Heterogeneous GPU+CPU system

2.1 Preliminaries

We first introduce two well-known time-memory tradeoffs (Hellman's method and the rainbow method) and some techniques for false alarm detection and efficient storage.

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 $|\mathcal{N}| = N$.

2.1.1 Hellman's Method

In this section, we summarize Hellman's method [22].

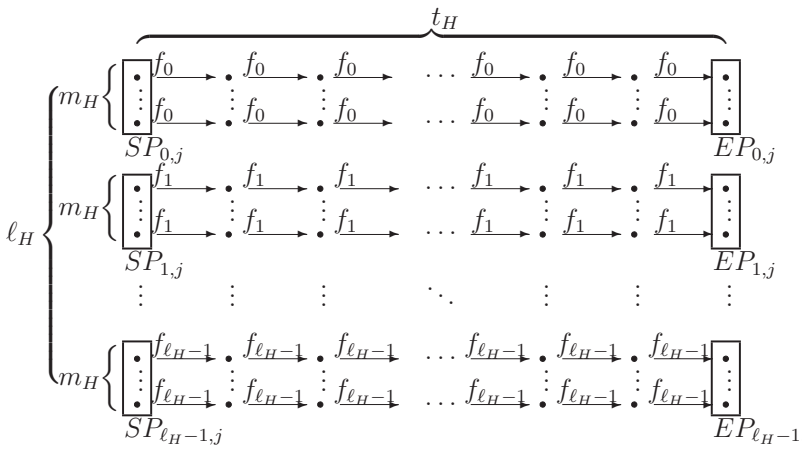

Figure 2.1: Hellman's table

Precomputation phase We randomly choose $\ell_H \cdot m_H$ start points in $\mathcal{N}$, labeled $SP_{i,j}$ for $0 \leq i < \ell_H$ and $0 \leq j < m_H$; $SP_{i,j}$ represents the start point at the j-th chain in the i-th table. For each $0 \leq i < \ell_H$ and $0 \leq j < m_H$, we set

$$x_{i,j,0} = SP_{i,j},$$

and compute

$$x_{i,j,k} = f_i(x_{i,j,k-1}) \text{ for } 1 \leq k \leq t_H,$$

recursively. That is, $\ell_H \cdot m_H$ chains of length t_H are produced using a function f_i for the i-th table. The last element x_{i,j,t_H} of each chain is called an end point ($EP_{i,j}$). The pairs of the start and end points, ($SP_{i,j}$, $EP_{i,j}$), are stored in a table, and they are sorted with respect to the end points. Note that all immediate points are discarded to reduce memory requirement.

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.

Hellman's method first finds the pre-image x_0 in the first table. If it fails to find the pre-image in the first table, then it searches the second one. This process is repeated until the pre-image is found.

At the first iteration in the i-th table, the online chain of length one is generated by computing $y_1 = R_i(y_0) = f_i(x_0)$, and we check whether it is an end point on the table by conducting a binary search. If $y_1 = EP_{i,j}$ for some j, which is referred to as an *alarm*, it means that x_0 is next to $EP_{i,j}$ in Figure 2.1 or $EP_{i,j}$ has more than one inverse images. The latter case is referred to as a *false alarm*. Therefore, we regenerate a chain starting from $SP_{i,j}$ to compute x_{i,j,t_H-1}, and check whether it is a false alarm or not by computing $g(x_{i,j,t_H-1}) = y_0$. If $g(x_{i,j,t_H-1}) = y_0$, we find the pre-image x_0, which is equal to x_{i,j,t_H-1}, and the online phase stops. If $y_1 \neq EP_{i,j}$ for all j or a false alarm occurred, then we compute $y_2 = f_i(y_1) = f_i(R_i(y_0))$, 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_H online chains fail to invert the given image y_0.

2.1.2 Rainbow Method

In this section, we summarize the rainbow method [51].

Precomputation phase We randomly choose m_R start points in $\mathcal{N}$, labeled SP_0, SP_1, ... , SP_{m_R-1}. For each $0 \leq i < m_R$, we set

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

and compute

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

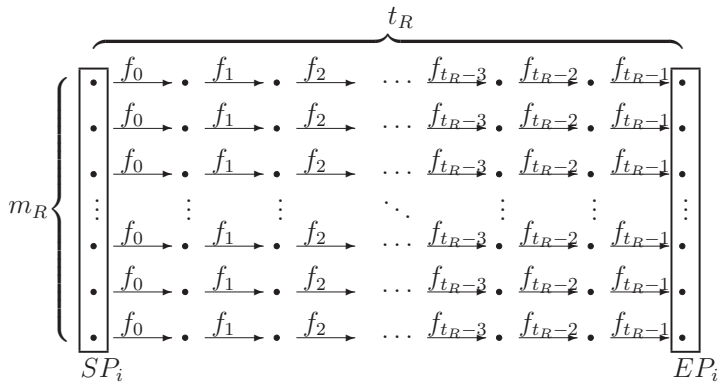

Figure 2.2: A rainbow table

recursively. In other words, m_R chains of length t_R are produced starting from SP_i $(0 \leq i < m_R)$ as shown in Figure 2.2. The last element x_{i,t_R} 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, as in Hellman's method.

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 k-th iteration, the online chain of length k, $y_k = f_{t_R-1}(f_{t_R-2}(\cdots(R_{t_R-k}(y_0))\cdots))$, is computed and we check whether $y_k = EP_i$ for some i. If alarms occur, we regenerate the chains of length $(t_R - k)$ starting from SP_i to compute x_{i,t_R-k} and we check whether $g(x_{i,t_R-k}) = y_0$. If so, we succeed in finding the pre-image x_0 which is equal to x_{i,t_R-k} and we stop the online phase. If $y_k \neq EP_i$ for all i or a false alarm occurs, the online phase carries out the $(k+1)$-th iteration.

To achieve high success probability, multiple tables can be used. The online

phase with the multiple tables is done by searching for the pre-image in the last column of each table and then searching sequentially through previous columns of all tables. Assume that ℓ_R is the number of rainbow tables, and then ℓ_R chains of length k are generated at the k-th iteration, which requires $(k-1) \cdot \ell_R$ number of one-way function applications.

Summary The online phase of time-memory tradeoffs (Hellman's and rainbow methods) can be divided into three parts: *online chain*, *lookup* and *regenerating chain*. The *online chain* procedure generates the online chain of length k at the k-th iteration. The *lookup* procedure checks whether each of these is an end point (alarm) through a binary search in the table. The *regenerating chain* procedure regenerates the chains of length $(t_\star - k)$, starting from start points for resolving alarms.[1] Because table lookup time through a binary search is negligible in comparison to the one-way function application time, the one-way function application is the dominant factor in the overall cost of the time-memory tradeoffs.

Note that time-memory tradeoffs are probabilistic algorithms. 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 table, the tradeoffs will succeed in finding it; Otherwise, they will fail. The success probability can be computed by the equation presented in [51, 44, 31].

[1] $t_\star$ is t_H or t_R, i.e., the chain length of Hellman's or the rainbow table, whose subscript is not explicitly specified.

2.1.3 Perfect Table

Many chains in the precomputed table could merge each other, and it reduces the success probabilities of time-memory tradeoffs. To increase the success probability under the same amount of memory, one can make the *perfect*[2] Hellman's or rainbow table in the precomputation phase.

In the case of the rainbow method, the perfect table can be easily constructed by leaving only one chain among the chains that have same end points in the table; The rest of the chains are removed. It is because two chains will have same end points if and only if the chains merge at a certain column in the rainbow table.

The perfect rainbow table generated with N start points, i.e., $m_R = N$, is called *maximal*. The maximal perfect table is impractical because the cost for constructing the table is too high.

2.1.4 False Alarm Detection with Checkpoints

By using *checkpoints* [8], the time for the regenerating chain procedure can be reduced. Not only the start and end points of the chains in the table but also the information of some intermediate points, i.e., *checkpoints*, are stored. The least significant bits of the intermediate points are usually stored. Using the information, one can detect false alarms in advance without regenerating the chains starting from start points. If alarms occur, one compares the information stored in the table with those of the online chain for each checkpoint. If they differ at least for one checkpoint, one knows for certain that this is a false

[2]The table that has no merging chains is called perfect.

alarm. If they are same for all checkpoints, the regenerating chain procedure should be executed as usual.

The analyses of the effects of checkpoints were given in [8, 29]. In [8], Avoine *et al.* analyzed the effects for the perfect rainbow method. Analyses for Hellman's method and the non-perfect rainbow method were done in [29]; However, for the rainbow method, the effect of only one checkpoint was analyzed. In Section 2.2, we give a complete analysis of the effect of multiple checkpoints for the non-perfect rainbow table for the optimal use of multiple checkpoints in our implementation of the rainbow method.

2.2 Checkpoints

By using checkpoints [8], 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 [8], 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 [29]. 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. We also analyze the performance improvements

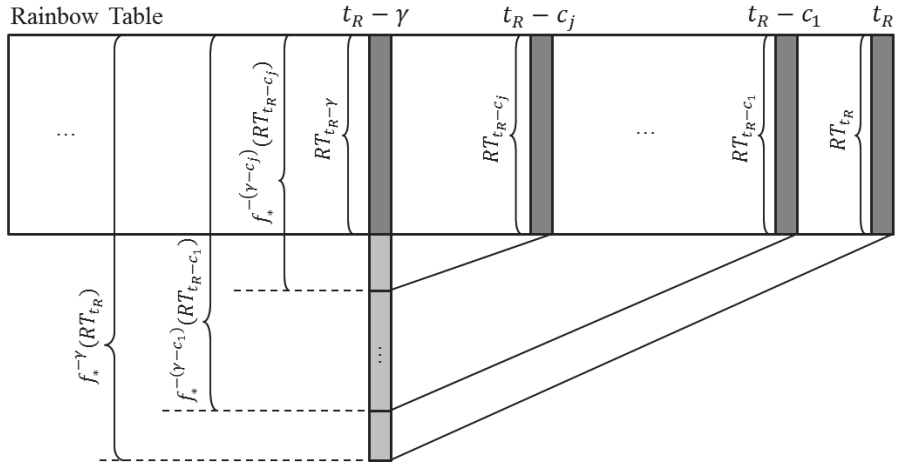

Figure 2.3: Sizes of the pre-images of end points at the $(t_R - \gamma)$-th column

for three cases in terms of the order of online chain generation. These results are used for efficient implementations in Section 2.3.

The multiset[3] of elements in the j-th column of the rainbow table is denoted by RT_j. 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 $(t_R - c_j)$-th columns of the table for $j = 1, \ldots, n$.

First, 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 $c_{n+1} = t_R$. This means that the pre-image x_0 is in $f_\star^{-\gamma}(RT_{t_R})$, where $f_\star$ is function f_j whose index j is not explicitly specified and $f_\star^{-\gamma}(RT_{t_R})$ is the multiset of pre-images under $f_\star^\gamma (= f_\star \circ \cdots \circ f_\star)$ of the

[3]The elements are allowed to appear more than once.

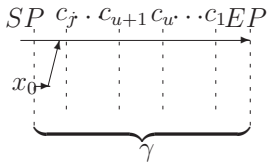

Figure 2.4: Merge before c_j

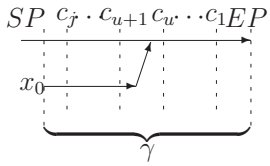

Figure 2.5: Merge between c_u and c_{u+1}

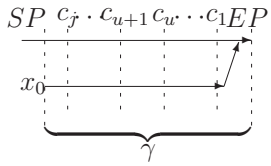

Figure 2.6: Merge after c_1

end points RT_{t_R}. As can be seen in Figure 2.3, the following relations hold:

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

We compute the probability of false alarms when checkpoints are used. For $\forall x_0 \in RT_{t_R-\gamma}$, x_0 can be found. For $\forall x_0 \in f_\star^{-(\gamma-c_j)}(RT_{t_R-c_j}) \setminus RT_{t_R-\gamma}$ (Figure 2.4), 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 $(t_R - c_j)$-th column, and j checkpoints are thus useless in detecting false alarms. For $\forall x_0 \in f_\star^{-(\gamma-c_u)}(RT_{t_R-c_u}) \setminus f_\star^{-(\gamma-c_{u+1})}(RT_{t_R-c_{u+1}})$ for $1 \le u \le j-1$ (Figure 2.5), 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 $\forall x_0 \in f_\star^{-\gamma}(RT_{t_R}) \setminus f_\star^{-(\gamma-c_1)}(RT_{t_R-c_1})$ (Figure 2.6), a false alarm occurs with probability $1/2^j$.

We now compute the improvement in the number of $f_\star$ applications due to checkpoints. Let $z_* = |RT_{t_R-\gamma}|$, $z_0 = |f_\star^{-\gamma}(RT_{t_R})|$, and $z_j = |f_\star^{-(\gamma-c_j)}(RT_{t_R-c_j})|$ for $j = 1, \ldots, n$. The expected number of false alarms without checkpoints when $x_0 \in f_\star^{-(\gamma-c_j)}(RT_{t_R-c_j}) \setminus RT_{t_R-\gamma}$ is

$$\frac{1}{N}\left|f_\star^{-(\gamma-c_j)}(RT_{t_R-c_j}) \setminus RT_{t_R-\gamma}\right| = \frac{1}{N}(z_j - z_*),$$

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 $x_0 \in f_\star^{-(\gamma-c_u)}(RT_{t_R-c_u}) \setminus f_\star^{-(\gamma-c_{u+1})}(RT_{t_R-c_{u+1}})$ is

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

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 $x_0 \in f_\star^{-\gamma}(RT_{t_R}) \setminus f_\star^{-(\gamma-c_1)}(RT_{t_R-c_1})$ is

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

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

$$\frac{1}{N}\Big\{(z_j - z_*) + \sum_{u=0}^{j-1} \frac{1}{2^{j-u}}(z_u - z_{u+1})\Big\}. \tag{2.1}$$

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

$$\frac{1}{N}(z_0 - z_*). \tag{2.2}$$

Hence, the expected decreasing number of false alarms due to checkpoints when an online chain of length γ is generated is (2.2) − (2.1), which simplifies to

$$\frac{1}{N}\Big\{(1 - \frac{1}{2^j})z_0 - \sum_{u=0}^{j-1} \frac{1}{2^{j-u}} z_{u+1}\Big\}. \tag{2.3}$$

According to Propositions 4 and 5 in [29], $z_0 \approx m_R(1+\gamma)$, $z_u \approx m_R(1+\gamma-c_u)$. Simplification of (2.3) using these approximations results in

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

We shall write $D(j)$ for this.

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 $(t_R - k + 1)$ is generated, i.e., the online chains are generated from longest to shortest; (iii) for a fixed $1 \leq \alpha \leq t_R$, if $k \leq \alpha$, the online chain of length k is generated; otherwise, the online chain of length $(t_R - k + \alpha + 1)$ is generated, i.e., the third case is a hybrid of (i) and (ii).

From shortest to longest online chains 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 $D(j)$ and the number of $f_\star$ applications for checking false alarms is $t_R - k + 1$[4]. 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

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

where m_j denotes the distinct number of elements in the j-th column of the rainbow table, i.e., $m_j \approx \frac{N}{N/m_R + j/2}$ [8]. Therefore, the expected number of $f_\star$ applications that can be removed through n 1-bit checkpoints is

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

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

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

From longest to shortest online chains At the k-th iteration, the online chain of length $(t_R - k + 1)$ is generated, i.e., $\gamma = t_R - k + 1$. Hence, for $c_j < \gamma \leq c_{j+1}$,i.e., $t_R + 1 - c_{j+1} \leq k < t_R + 1 - c_j$, 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 k. The probability that the k-th iteration is processed is

$$\prod_{i=0}^{k-2} \big(1 - \frac{m_i}{N}\big).$$

Therefore, the expected number of $f_\star$ applications that can be removed through n 1-bit checkpoints is

$$\sum_{j=1}^{n} \Big\{ \sum_{t_R+1-c_{j+1} \leq k < t_R+1-c_j} k \cdot D(j) \cdot \prod_{i=0}^{k-2} \big(1 - \frac{m_i}{N}\big) \Big\}.$$

Hybrid 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. At the k-th iteration such that $\alpha+1 \leq k \leq t_R$, the online chain of length $(t_R+\alpha+1-k)$ is generated, i.e., $\gamma = t_R+\alpha+1-k$. Hence, for $c_j < \gamma \leq c_{j+1}$, i.e., for $t_R + \alpha + 1 - c_{j+1} \leq k < t_R + \alpha + 1 - c_j$, the expected decreasing number of false alarms due to checkpoint is $D(j)$ and the number of $f_\star$ applications for checking false alarms is $k - \alpha$. The probability that the k-th iteration is processed is

$$\prod_{i=1}^{\alpha} \big(1 - \frac{m_{t_R-i}}{N}\big) \cdot \prod_{i=0}^{k-2-\alpha} \big(1 - \frac{m_i}{N}\big).$$

Therefore, the expected number of $f_\star$ applications that can be removed

through n 1-bit checkpoints is

$$\sum_{j=1}^{n}\Big\{\sum_{\substack{c_j<k\le c_{j+1}\\ 1\le k\le\alpha}}(t_R-k+1)\cdot D(j)\cdot\prod_{i=1}^{k-1}\big(1-\frac{m_{t_R-i}}{N}\big)+$$

$$\sum_{\substack{t_R+\alpha+1-c_{j+1}\le k<t_R+\alpha+1-c_j\\ \alpha+1\le k\le t_R}}(k-\alpha)\cdot D(j)\cdot\prod_{i=1}^{\alpha}\big(1-\frac{m_{t_R-i}}{N}\big)\cdot\prod_{i=0}^{k-\alpha-2}\big(1-\frac{m_i}{N}\big)\Big\}.$$

Tables 2.1, 2.2, 2.3 and 2.4 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 online chains, and a hybrid with $\alpha = 17,920$[5]), where $N = 3.58\times10^{12}$, $m_R = 80,530,636$, and $t_R = 73,403$. The optimal positions represent the ratio from the rightmost column of the table. We used Maple 12 [4] to obtain these positions. 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 3 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 3 of [29].

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

$$\sum_{k=1}^{t_R}k\cdot\frac{m_R}{N}(t_R-k+2)\cdot\prod_{i=0}^{k-2}\big(1-\frac{m_i}{N}\big).$$

Theorem 2. *Assume that, for a fixed* $1\le\alpha\le t_R$*, the online chain of length* k *is generated if* $k\le\alpha$ *and otherwise that of length* $(t_R-k+\alpha+1)$ *at the* k*-th*

[5]The reason that α is set as $17,920$ will be explained in Section 2.3.4.

Table 2.1: Expected numbers of $f_\star$ applications (unit: t_R^2) in the regenerating chain procedure and performance improvement due to checkpoints at the optimal positions when the online chains are generated from shortest to longest.

# 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	0.2791	0.2123	0.1732	0.1470	0.1281	0.1136	0.1022	0.0930
		0.3591	0.2827	0.2356	0.2028	0.1785	0.1597	0.1446
			0.4179	0.3379	0.2863	0.2495	0.2216	0.1996
				0.4637	0.3826	0.3287	0.2894	0.2590
					0.5008	0.4199	0.3649	0.3239
						0.5317	0.4517	0.3962
							0.5579	0.4792
								0.5806

Table 2.2: Expected numbers of $f_\star$ applications (unit: t_R^2) in the regenerating chain procedure and performance improvement due to checkpoints at the optimal positions when the online chains are generated from longest to shortest.

# of $f_\star$ applications without checkpoints (1)	0.1454							
Reduced # of $f_\star$ applications with checkpoints (2)	0.0345	0.0554	0.0696	0.0798	0.0875	0.0936	0.0985	0.1026
Improvement ((2)/(1))	23.7%	38.1%	47.9%	54.9%	60.2%	64.4%	67.7%	70.6%
Optimal positions	0.3937	0.3115	0.2589	0.2220	0.1945	0.1733	0.1563	0.1424
		0.4819	0.3958	0.3375	0.2948	0.2620	0.2360	0.2148
			0.5435	0.4583	0.3982	0.3528	0.3171	0.2882
				0.5895	0.5068	0.4467	0.4003	0.3631
					0.6253	0.5457	0.4866	0.4400
						0.6542	0.5778	0.5200
							0.6780	0.6047
								0.6981

Table 2.3: Expected numbers of $f_\star$ applications (unit: t_R^2) in the regenerating chain procedure and performance improvement due to checkpoints at the optimal positions. (Hybrid with $\alpha = 17,920$)

# of $f_\star$ applications without checkpoints (1)	0.1392							
Reduced # of $f_\star$ applications with checkpoints (2)	0.0240	0.0449	0.0562	0.0675	0.0741	0.0812	0.0855	0.0905
Improvement ((2)/(1))	17.2%	32.3%	40.4%	48.5%	53.2%	58.3%	61.4%	65.0%
Optimal positions	0.2123	0.1773	0.1400	0.1258	0.1053	0.0978	0.0846	0.0801
		0.4437	0.2234	0.1991	0.1647	0.1525	0.1310	0.1238
			0.4767	0.3968	0.2291	0.2112	0.1801	0.1699
				0.5443	0.4217	0.3695	0.2326	0.2188
					0.5625	0.4838	0.3888	0.3513
						0.6083	0.4993	0.4455
							0.6197	0.5447
								0.6534

Table 2.4: Reduced numbers of $f_\star$ applications (unit: t_R^2) in the regenerating chain procedure when 22 1-bit checkpoints are used. (STL: from shortest to longest online chains, LTS: from longest to shortest online chains)

	Reduced #	Improvement	Optimal positions
STL	0.1409	84.0%	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, 0.7381
LTS	0.1259	86.6%	0.0639, 0.0961, 0.1283, 0.1606, 0.1931, 0.2257, 0.2585, 0.2916, 0.3248, 0.3584, 0.3923, 0.4265, 0.4613, 0.4965, 0.5324, 0.5690, 0.6066, 0.6455, 0.6859, 0.7285, 0.7744, 0.8261
Hybrid ($\alpha = 17,920$)	0.1160	83.3%	0.0352, 0.0535, 0.0721, 0.0911, 0.1105, 0.1303, 0.1505, 0.1713, 0.1925, 0.2143, 0.2367, 0.3001, 0.3435, 0.3875, 0.4321, 0.4775, 0.5240, 0.5718, 0.6214, 0.6735, 0.7294, 0.7919

iteration. The number of $f_\star$ applications in the regenerating chain procedure without checkpoints is

$$\sum_{k=1}^{\alpha}(t_R-k+1)\cdot\frac{m_R}{N}(1+k)\cdot\prod_{i=1}^{k-1}\left(1-\frac{m_{t_R-i}}{N}\right)+$$
$$\sum_{k=\alpha+1}^{t_R}(k-\alpha)\cdot\frac{m_R}{N}(t_R-k+\alpha+2)\cdot\prod_{i=1}^{\alpha}\left(1-\frac{m_{t_R-i}}{N}\right)\cdot\prod_{i=0}^{t_R-\alpha-2}\left(1-\frac{m_i}{N}\right).$$

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 Tables 2.1, 2.2, and 2.3, $0.1676t^2$, $0.1454t^2$, and $0.1392t^2$ 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 checkpoints are reduced by about 27% and 13%, respectively, compared to STL.

2.3 Implementations Using GPU

In this chapter, 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, (ii) load balancing between GPU and CPU using checkpoints, and (iii) changing the order of the online chain generation.

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[6] rainbow table with 70% success probability, in which $m_R = 80,530,636$, $t_R = 73,403$. For reasons of efficient memory access, a start point of $\lceil \log_2 m_R \rceil = 27$ bits is stored in a 32-bit data type, `uint32_t`, and an end point of $\lceil \log_2 N \rceil = 42$ bits[7] 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 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.

2.3.1 Naive GPU

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 ($1 \leq i \leq t_R$) 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 $(t_R - i)$ and it checks whether the element in the $(t_R - i)$-th column is x_0 or a false alarm (the regenerating chain 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

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

[7]For the simple implementation, efficient storage techniques [31] such as the index file and the end point truncation were not considered.

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 2.5 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 2.5: Time of online phase when it fails (Naive GPU)

procedures	time	chain length
online chain+lookup+regenerating chain	143.6 sec	4.2×10^9
online chain+lookup	7.7 sec	2.7×10^9

As can be seen in Table 2.5, 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.

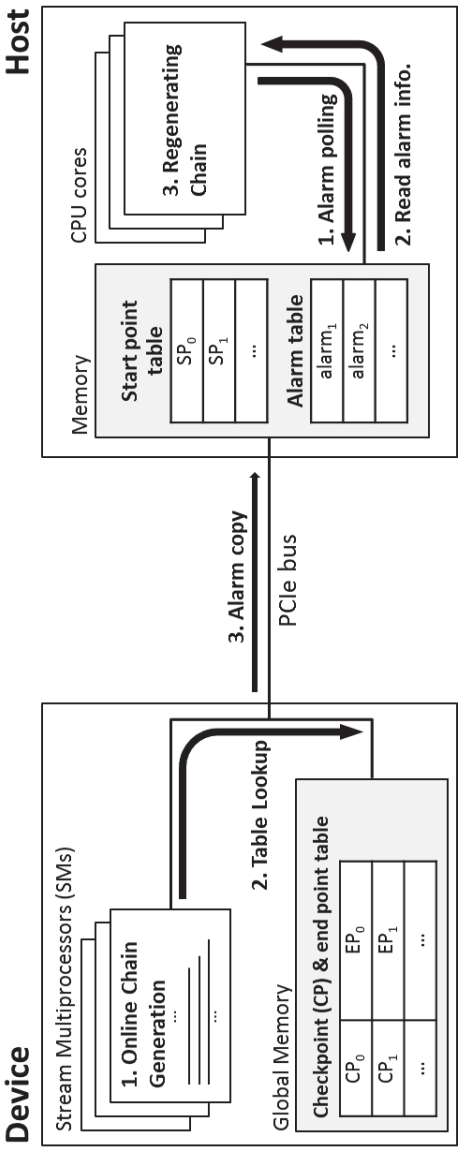

Figure 2.7: Implementation in a heterogeneous GPU+CPU system

Table 2.6: Time of online phase when it fails (GPU+CPU)

online chain+lookup (GPU)	regenerating chain (CPU)	total
7.7 sec	70 sec	70 sec

2.3.2 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 2.7. 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*.

The execution time of the GPU+CPU is shown in Table 2.6. The GPU processes A in 7.7 seconds, whereas on the CPU it takes 70 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.

2.3.3 GPU+CPU with checkpoints

We take advantage of checkpoints [8] 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 generated from shortest to longest, the 22 checkpoints are expected to decrease the number of $f_\star$ applications due to false alarms by about 84% and their optimal positions are in Table 2.2.

So far, we introduced three different kinds of implementations using the GPU: naive GPU, GPU+CPU and GPU+CPU with checkpoints. Figure 2.8 also shows the experimental results using the CPU, as well as those of the three implementations presented in this thesis. In the case of the CPU, the i-th thread generates the online chain of length i and regenerates the chain of length $(t_R - i)$ 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 18.3 times faster than the CPU and 10.9 times faster than the naive GPU. Also, the GPU+CPU with checkpoints is 4.07 times faster than the GPU+CPU.

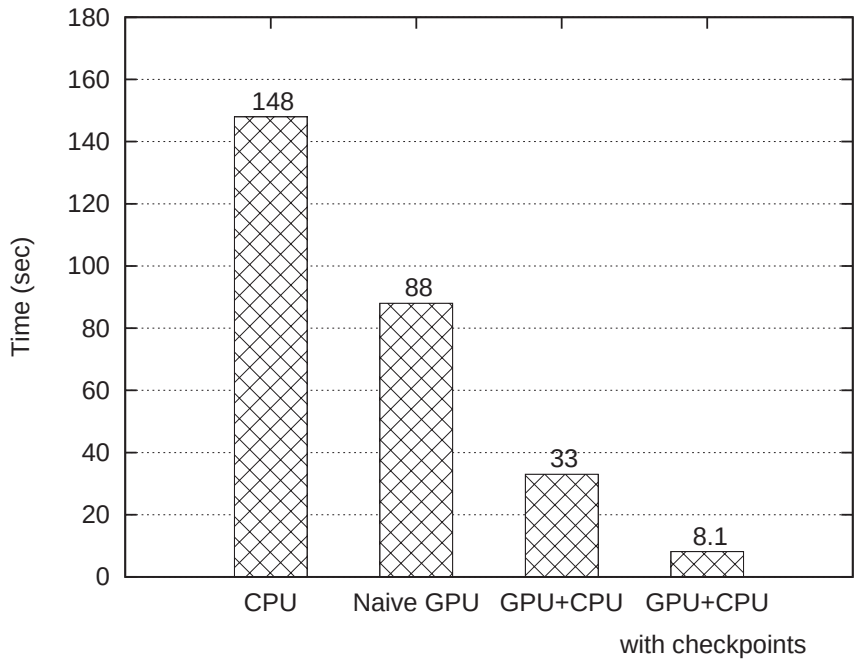

Figure 2.8: Timings of searching for a pre-image. Each bar represents the average time for the whole 50 experiments.

2.3.4 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.

Figure 2.9 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

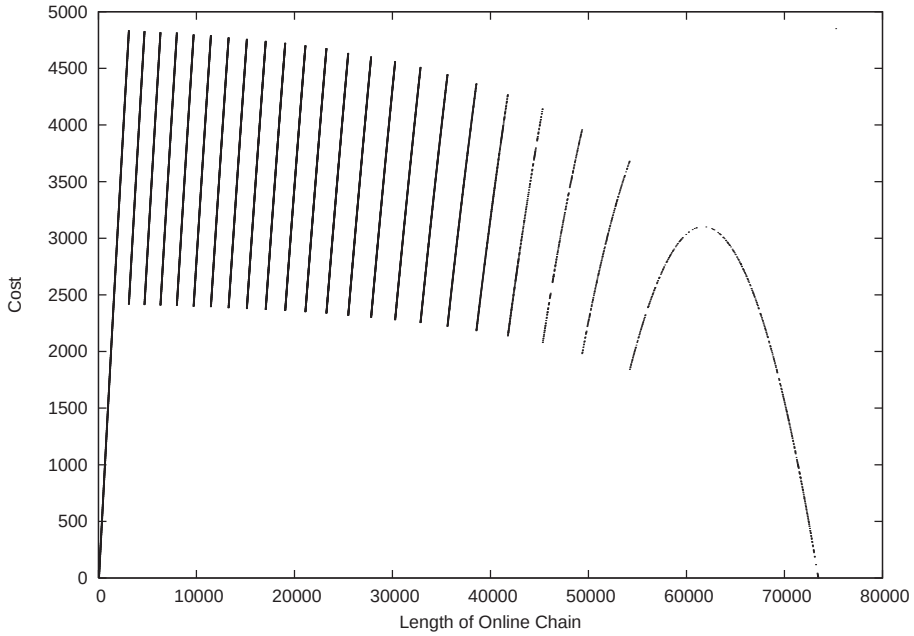

Figure 2.9: The expected number of $f_\star$ applications in the regenerating chain procedure when 22 1-bit checkpoints for STL are applied.

$\{m(\gamma+1)/N - D(j)\} \cdot (t-\gamma+1)$ when the online chain of length γ such that $c_j < \gamma \leq c_{j+1}$ is generated. In Figure 2.9, 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. According to Tables 2.1, 2.2 and 2.4, the expected total length of the chains generated in the regenerating chain procedure is reduced by about 27%.

Figure 2.10 shows the average times of the GPU+CPU with checkpoints

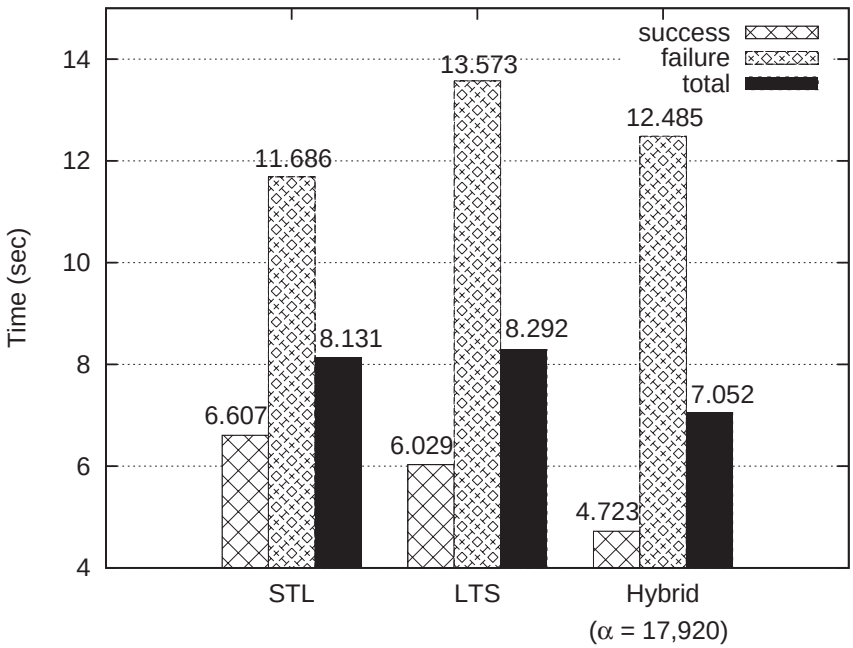

Figure 2.10: Average times of the GPU+CPU with 22 1-bit checkpoints to search for a pre-image in 50 experiments. (STL: from shortest to longest online chains, LTS: from longest to shortest online chains)

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 2.2. Contrary to our expectations, the GPU+CPU with checkpoints from longest to shortest online 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 GTX580, thousands of online chains could be generated in parallel. Therefore, the implementation from longest to shortest online chains takes longer start-up time than that from shortest to longest online chains. We can infer from the average times of the failure case in Figure 2.10 that the start-up time of the implementation from longest to shortest online chains is about $13.573 - 11.686 = 1.887$ 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 $1 \leq \alpha \leq t_R$, 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 $(t_R - k + \alpha + 1)$ 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 $\alpha = 17,920$ to minimize the average time of searching for a pre-image. According to Tables 2.1, 2.3 and 2.4, 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. As can be seen from Figure 2.10, the hybrid with $\alpha = 17,920$ improves the

Table 2.7: Timings of searching for a pre-image. (sec)

	online chain+lookup	regenerating chain	total
RainbowCrack	6.958	2.600	9.558
Cryptohaze	6.800	8.540	15.340
Ours (Hybrid)	5.285	7.052	7.052

performance of the GPU+CPU with checkpoints by about 13% by simply changing the order of online chain generation.

2.4 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 [1, 2, 3]. Ophcrack [2] provides only a CPU-accelerated implementation, whereas RainbowCrack [3] and Cryptohaze [1] provide not only a CPU-accelerated implementation but also a GPU-accelerated one. As the implementations on GPU are much faster than the implementations on CPU, we compare ours with only GPU-accelerated ones of RainbowCrack and Cryptohaze. We created the non-perfect rainbow table of the same size as ours, i.e., $m_R = 80,530,636$ and $t_R = 73,403$. All start and end points in the table can be loaded on the device memory of the GPU.

Table 2.7 shows the timings of RainbowCrack, Cryptohaze and ours of searching for a pre-image. 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, and our total time is equal to the maximum of the two. As a result, our implementation is about 1.36 and 2.18 times faster than RainbowCrack and Cryptohaze, respectively.

While we did not consider the perfect table in this thesis, our GPU+CPU with checkpoints would be better than other implementations for the perfect table. All key factors mentioned in this thesis for achieving good performance are still effective for the perfect table. Also, our implementation runs the online chain+lookup procedures on GPU and the regenerating chain procedure on CPU in parallel, and the total time of ours is the maximum of the times for the online chain+lookup procedures and the regenerating chain procedure, unlike RainbowCrack and Cryptohaze. Therefore, our implementation would have better performance for the perfect table.

Chapter 3

Implementation of NTRU Cryptosystem Using Sliding Window Methods

3.1 Preliminaries

3.1.1 Polynomial Ring and Convolution Product

Let Z be the set of integers. The polynomial ring over Z, denoted by $Z[X]$, is the set of all polynomials with coefficients in Z. We define R as the quotient ring $Z[X]/(X^N - 1)$. Then an element $a \in R$ can be written as a polynomial or a vector,

$$a(X) = \sum_{i=0}^{N-1} a_i X^i = [a_0, a_1, \ldots, a_{N-1}].$$

Throughout this thesis, we denote a polynomial $a(X) \in R$ as simply a, if there is no danger of confusion.

Multiplication of two polynomials $a, b \in R$ can be represented as a *convolution product* $c \in R$, which is given by $c(X) = a(X) * b(X)$ with

$$c_k = \sum_{i=0}^{k} a_i b_{k-i} + \sum_{i=k+1}^{N-1} a_i b_{N+k-i} = \sum_{i+j \equiv k \bmod N} a_i b_j,$$

since $X^N \equiv 1 \bmod (X^N - 1)$. The convolution product computation is the main operation of the NTRU cryptosystem.

3.1.2 NTRU Public-Key Cryptosystem

In this section, we briefly review the NTRU cryptosystem. NTRU has three public parameters (N, p, q), where N and q are integers and p is either a small prime number or a simple polynomial. There are several variants of NTRU according to the form of these parameters, but typical choices are $p = 2$, $p = 3$, or $p = X + 2$. Throughout this thesis, however, we only consider the case with $p = 2$ according to the description given in [35]. In this case, $gcd(p, q) = 1$ and $p \ll q$. For a polynomial $a(X) \in R$, we define reduction mod q as $a(X) \bmod q := \sum_{i=0}^{N-1} (a_i \bmod q) X^i$. The inverse of polynomial $f(X)$ mod q, denoted by $f^{-1}(X) \bmod q$, is defined as the polynomial satisfying $f(X) * f^{-1}(X) \equiv 1 \bmod q$.

Key Generation: Randomly choose polynomials $F, g \in R$ with small coefficients. Then compute the private key $f := 1 + pF$ and the public key $h := pf^{-1} * g \bmod q$.

Encryption: Let m be the polynomial representing a message. Then randomly choose a polynomial $r \in R$ with small coefficients, and compute the ciphertext $e := r * h + m \bmod q$.

Decryption: Given a ciphertext e, compute the product $a := f * e \bmod_A q$, where $\bmod_A q$ means that the coefficients are shifted into the interval $[A, A + q - 1]$ after reduction mod q. The value of A is fixed and is determined by a simple formula depending on the other parameters. Then recover the plaintext m as $m := a \bmod p$.

Correctness of decryption: The polynomial a satisfies

$$a \equiv f * e \equiv f * (r * h + m) \equiv pr * g + m * f \bmod q.$$

Consider the last polynomial $pr * g + m * f$. By an appropriate choice of parameters, one can adjust its coefficients to lie in an interval of length less than q. Hence we can recover

$$a = pr * g + m * f = pr * g + m * (1 + pF)$$

exactly, not merely mod q. In other words, $m \equiv a \bmod p$.

3.1.3 Basic Convolution Algorithm

The most time consuming part of NTRU is the convolution such as $r * h \bmod q$ and $F * e \bmod q$. The well known parameter selection method to reduce this complexity is to select r and F with only binary coefficients, i.e., 0 or 1.[1] Algorithm 3.1 shows an algorithm to compute $a * c$, where $a \in R$ is a binary polynomial with a small fixed Hamming weight $HW(a) = d$ and $c \in R$ is a general polynomial. The binary polynomial a is represented as an array of d locations for bit 1, and it represents a polynomial such as r and F in NTRU.

[1] In this thesis, we do not consider a more restricted form, i.e., *product-form polynomials* [27, 28].

Algorithm 3.1 Basic Convolution Algorithm

Input: b (array of d locations for '1' representing the binary polynomial $a(X)$); $c(X)$ (general polynomial); N (number of coefficients in a polynomial).
Output: t (array representing $t(X) = a(X) * c(X)$).

1: **for** $0 \leq j < 2N$ **do**
2: $\quad t_j \leftarrow 0$
3: **for** $0 \leq j < d$ **do**
4: $\quad$ **for** $0 \leq k < N$ **do**
5: $\quad\quad t_{k+b[j]} \leftarrow t_{k+b[j]} + c_k$
6: **for** $0 \leq j < N$ **do**
7: $\quad t_j \leftarrow (t_j + t_{j+N}) \bmod q$

Algorithm 3.1 eliminates the overhead for index computation (modulo N) at the cost of additional memory (t_N to t_{2N-1}) to store N integers. Consequently, the total amount of computations in Algorithm 3.1 is $(d+1)N$ additions and N reductions mod q.

3.2 Sliding Window Method for NTRU

In this section, we present our sliding window method for NTRU, which is based on the fact that if we find repeated patterns in the coefficients of a binary polynomial a, then we can accelerate the convolution $a * c$. We begin by examining the structure of a convolution operation. Note that multiplication of $a \in R$ and $c \in R$ can be represented as the convolution product $t \in R$, where $t_k = \sum_{i+j \equiv k \mod N} a_i c_j$. This operation can be rewritten as follows:

$$t(X) = \begin{pmatrix} a_0 & a_{N-1} & \cdots & a_2 & a_1 \\ a_1 & a_0 & \cdots & a_3 & a_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{N-1} & a_{N-2} & \cdots & a_1 & a_0 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{N-1} \end{pmatrix}.$$

We observe that each row in the $N \times N$ matrix is produced by rotating the previous row by one position.

Now we give a small example with $a(X) = X + X^2 + X^5 + X^6 + X^8 + X^9$ with $N = 10, d = 6$. For simplicity, we will write a binary polynomial as a bit string. Thus $a(X)$ will be written as 0110011011. Then $a(X) * c(X)$ can be written as

$$t(X) = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \\ c_7 \\ c_8 \\ c_9 \end{pmatrix}.$$

Hence t_0 will be computed as $t_0 = c_1 + c_2 + c_4 + c_5 + c_8 + c_9$, which requires six additions. Among these additions, now we concentrate on the term $c_1 + c_2$. We can see that this term also occurs in the computation of t_3 and t_7. This is because the pattern '11' is repeated three times in the binary representation of $a(X)$. To be more precise, $c_1 + c_2$ in the computation of t_0, t_3, t_7 corresponds to $\underline{11}_3$, $\underline{11}_1$, $\underline{11}_2$ in $a(X) = 0\underline{11}_1 00\underline{11}_2 0\underline{11}_3$, respectively. Since the term $c_1 + c_2$ occurs three times, we can compute this term only once, store it in a look-up table, and reuse it when it is required, which can reduce the number of additions by two. This reduction can also be applied to other terms related to the pattern '11'. For example, the term $c_2 + c_3$ occurs in the computation of

t_1, t_4 and t_8, the term $c_3 + c_4$ in t_2, t_5 and t_9, and so on.

The above idea can be applied to other patterns such as '101', '1001', '111', etc., if we can only find the occurrences of these patterns that don't share '1's in the binary representation of the polynomial. The following lemma shows a general rule for the relationship between pattern occurrences and the amount of computations, which states that it is desirable to find patterns that have many '1's and repeat many times.

Lemma 1. *Let $m, n \geq 1$. If a pattern containing n '1's occurs m times in the binary representation of a polynomial with N coefficients, then we can reduce the number of integer additions by $N(m-1)(n-1)$.*

Proof. It is straightforward since the number of integer additions related to such a pattern is reduced to $N(m+n-1)$ from Nmn. □

Since we cannot assign many resources to pattern searching, a reasonable heuristic to find repeated patterns is to partition the binary string into short blocks of a fixed length, w, except for the parts containing consecutive zeros. In fact, this is equivalent to what is called a *sliding window method* with window size w in the context of exponentiation [39]. Then a convolution algorithm is composed of three stages; the *recoding stage* where the binary polynomial is scanned and the indices of pattern occurrences are marked, the *precomputation stage* where a look-up table is constructed, and *convolution stage* where the coefficient computation is performed. The following example illustrates the recoding for $w = 3, N = 28$ when we scan the input bit string from right to left:

$$\underline{1}00\underline{011}\ \underline{001}000\underline{111}\underline{010}100000\underline{01}0. \tag{3.1}$$

However, the binary polynomials used in NTRU have quite a biased distribution of 1s. For example, F and r are sparse polynomials with $HW(F)/N = HW(r)/N < 0.2$ according to [35]. Then it is reasonable to conjecture that window blocks containing more than two '1's rarely occur, if we set the window size to practical values such as $w = 6$. Hence a simpler algorithm which only considers blocks with up to two '1's is expected to give more speed-ups, because it eliminates the unnecessary precomputation for rarely used table elements. Now we define bit patterns containing exactly two '1's as *simple patterns*, and denote them as $p_1 =$ '11', $p_2 =$ '101', $p_3 =$ '1001', and so on. Then, in the recoding stage, we will try to find simple patterns up to length w. Even if we find three or more '1's in a window block, we will use only two of them. We define a *reduced pattern set* with window size w as $P_w = \{p_0, p_1, \ldots, p_{w-1}\}$, where $p_0 =$ '1'. We will call this method a *sliding window method with reduced pattern set*, SWR for short. The method allowing general patterns with more than two 1s will be called a *general sliding window method*, SWG for short.

In this section, we will only explain the algorithm for SWR in more detail, because that for SWG is similar. First, we start with the recoding procedure. Our strategy for recoding is to use a finite state machine which implements a greedy algorithm. That is, if we encounter a '1' while scanning an input bit string from right to left, we look at the next $w - 1$ bits until we find another '1'. If there isn't, the encountered '1' becomes separated. The following is an example recoding for $w = 4, N = 28$.

$$\underline{1}0000\underline{1}\ \underline{1001}000\underline{1001}0\underline{101}0\underline{11}\ \underline{0001}.$$

The occurrences of p_0 through p_3 will be stored in arrays b_0 through b_3, re-

spectively, as follows:

$$b_0 = [27, 5, 0], b_1 = [23], b_2 = [20], b_3 = [16, 9].$$

Although the above algorithm is a very simple $O(N)$-time algorithm, it is optimal in the sense that there cannot be any recoding algorithm that finds more simple patterns.

Theorem 3. *The recoding algorithm above is an optimal algorithm to find the maximum number of simple patterns with length $\leq w$ in a bit string.*

Proof. Note that if there is an interval containing $w - 1$ or more consecutive zeros, then there cannot be any simple pattern with length $\leq w$ that overlaps this interval. Therefore these zero intervals partition the input string x into many segments, and the distance of two neighboring ones that belong to a same segment should always be less than $w - 1$. Now we only have to show that within a single segment, the greedy algorithm is optimal, which is straightforward since we can find k simple patterns in a segment with $2k$ or $2k + 1$ ones. □

Algorithm 3.2 is the procedure for SWR using P_w. Lines 1 through 8 are the precomputation stage, and Lines 9 through 22 are the convolution stage. The total amount of computations in Algorithm 3.2 is expressed as

$$[(w-1)N\ \mathbf{A_P}] + [(e_0 + \cdots + e_{w-1} + 1)N\ \mathbf{A_C}] + [N\ \mathbf{MR}], \tag{3.2}$$

where $\mathbf{A_P}, \mathbf{A_C}$ and $\mathbf{MR}$ represent the complexities of an addition in the precomputation stage, an addition in the convolution stage, and a modular reduction, respectively. On the other hand, the total amount of computations

in SWG is

$$[(2^{w-1}-1)N\ \mathbf{A_P}] + [(d_1 + \cdots + d_{2^w-1} + 1)N\ \mathbf{A_C}] + [N\ \mathbf{MR}], \qquad (3.3)$$

where d_i is the number of occurrences of the pattern q_i interpreted into integer i. For example, d_7 is the number of occurrences of q_7 ='111'. A notable fact in SWR is that the number of additions in the precomputation stage is linear in the window size w, while it is exponential for SWG as shown in (3.3). On the other hand, as will be analyzed in the next section, the number of additions in the convolution stage, i.e., the second term in (3.2), almost does not increase compared to (3.3), which reduces the total number of additions substantially.

The total amount of temporary memory required for the precomputation table is $(w-1)N$ integers. However, this memory requirement can be significantly reduced by interleaving the precomputation and convolution stages. That is, we just insert lines 5 through 7 between lines 13 and 14. (For $i = 0$, no precomputation is done.) Then we can replace all the T_i's by a single table T, since every table element is independent from the others. Then the amount of memory is reduced to N integers regardless of the window size w. Let this improved method be SWR_I. Using similar optimization, the total amount of temporary memory required for SWG may be reduced from $(2^{w-1}-1)N$ to N for $w = 2, 3, 4$ and $2N$ for $w = 5, 6$. Let this improved general method be SWG_I.

3.3 Performance Analysis

In this section, we analyze the performances of the two sliding window methods, SWG and SWR. Because their amounts of computations are (3.2) and

Algorithm 3.2 Sliding Window Method (SWR)

Input: b_i for $0 \leq i \leq w-1$ (arrays representing the e_i positions of p_i in the binary polynomial $a(X)$); $c(X)$ (general polynomial); N (number of coefficients in a polynomial); w (window size).
Output: t (array representing $t(X) = a(X) * c(X)$).

1: **for** $0 \leq j < w-1$ **do**
2: $\quad c_{j+N} \leftarrow c_j$
3: **for** $1 \leq i \leq w-1$ **do**
4: $\quad$ **for** $0 \leq j < N$ **do**
5: $\quad\quad T_i[j] \leftarrow c_j + c_{j+i}$
6: Let $T_0 = c$, i.e., $T_0[k] = c_k$ for $0 \leq k < N$.
7: **for** $0 \leq j < 2N$ **do**
8: $\quad t_j \leftarrow 0$
9: **for** $0 \leq i \leq w-1$ **do**
10: $\quad$ **for** $0 \leq j < e_i$ **do**
11: $\quad\quad$ **for** $0 \leq k < N$ **do**
12: $\quad\quad\quad t_{k+b_i[j]} \leftarrow t_{k+b_i[j]} + T_i[k]$
13: **for** $0 \leq j < N$ **do**
14: $\quad t_j \leftarrow (t_j + t_{j+N}) \bmod q$

(3.3), it is necessary to estimate d_i and e_i, which are the number of pattern occurrences. Note that d_i and e_i are determined by recoding procedures. Hence we need to analyze the behavior of these recoding algorithms. We use the Markov chain [21] which is a well-known technique for the analysis of random processes. Markov chains have also been used often for the analysis of exponentiation and scalar multiplication, e.g., [52] and [53], and we remark that our analysis is motivated by [53].

We begin by examining the coefficient selection process for binary polynomials. As described in Section 3.1, a binary polynomial a, i.e., r or F, is randomly selected such that exactly d out of N coefficients are equal to 1. That is, we choose $a_{N-1}, a_{N-2}, \ldots, a_0$ sequentially without replacement from N bits, of which d coefficients are '1' and $N-d$ are '0'. Although this sampling

is done without replacement, it is approximately the same as sampling with replacement for the first m coefficients $a_{N-1}, \ldots, a_{N-m}$, if we assume that d and N are sufficiently large compared to m. Then the probability that $a_i =$'1' is d/N for $i = N-1, N-2, \ldots, N-m$, regardless of the previous coefficients.

Now we study the behavior of recoding procedure for SWR. To help understanding, we start with a small window size, $w = 3$, and model the recoding procedure as a random process $(X_0, X_1, \ldots)$ with state space $S = \{s_0 =$ '0', $s_1 =$ '001', $s_2 =$ '011', $s_3 =$ '101', $s_4 =$ '111'$\}$. If we approximate the probability that $a_i =$ '1' to $p = d/N$, it is easy to see that the 5×5 transition matrix P is determined by $S = \{s_0 =$ '0', $s_1 =$ '001', $s_2 =$ '11', $s_3 =$ '101'$\}$, which represents the policy that if the scanning procedure encounters a '11', then it produces a window block immediately without looking at the third bit. If we approximate the probability that $a_i =$ '1' as $p = d/N$, the transition matrix is

$$P = \begin{pmatrix} 1-p & p(1-p)^2 & p^2 & p^2(1-p) \\ 1-p & p(1-p)^2 & p^2 & p^2(1-p) \\ 1-p & p(1-p)^2 & p^2 & p^2(1-p) \\ 1-p & p(1-p)^2 & p^2 & p^2(1-p) \end{pmatrix},$$

which constitutes an irreducible and aperiodic Markov chain. Then, if N is sufficiently large, the distribution of window blocks becomes stationary, i.e., we obtain a $\pi = (\pi_0, \ldots, \pi_4)$ such that $\pi P = \pi$ [21].[2] We can easily compute π as an eigenvector of P such that $\pi P = \pi$ and $\sum_{i=0}^{k-1} \pi_i = 1$. Let $l(s_i)$ be the bit length of s_i, i.e., $l(s_0) = 1$ and $l(s_i) = 3$ for $i = 1, 2, 3, 4$. Then the *density* of state s_i, denoted by $den(s_i)$ for $i = 0, \ldots, 4$, can be computed as

[2]Note that in the analysis of exponentiation, a similar assumption that the bit length of an exponent is sufficiently long is used [53].

$den(s_i) = \pi_i/(\sum_{i=0}^{4} \pi_i l(s_i))$, and the expected number of occurrences of s_i in a binary polynomial with N coefficients can be estimated by $N \times den(s_i)$. We can generalize this result to larger values of w, and obtain the following theorem.

Theorem 4. *Let $w \geq 2$ and let $p = d/N$. Let s_0 = '0' and $s_i = p_{i-1}$ for $i = 1, 2, \ldots, w$. Let P be a $(w+1) \times (w+1)$ matrix such that $P_{i,0} = 1 - p$ and $P_{i,1} = p(1-p)^{w-1}$ for $i = 0, 1, \ldots, w$ and $P_{i,j} = p^2(1-p)^{j-2}$ for $i = 0, 1, \ldots, w$ and $j = 2, 3, \ldots, w$. Let $\pi = (\pi_0, \pi_1, \ldots, \pi_w)$ be a row vector such that $\pi P = \pi$ and $\sum_{i=0}^{w} \pi_i = 1$. Then the density of s_i is*

$$den(s_i) = \pi_i/(\textstyle\sum_{i=0}^{w} \pi_i l(s_i)) \tag{3.4}$$

for $i = 0, 1, \ldots, w$, if N is sufficiently large.

Proof. The proof is straightforward. □

The analysis of SWG is similar, and we obtain the following theorem. First, we define $HW_w(x)$ for an integer x as the Hamming weight of x when x is represented as a w-bit number.

Theorem 5. *Let $w \geq 2$ and let $p = d/N$. Let s'_0 = '0' and $s'_i = q_{2i-1}$ for $i = 1, 2, \ldots, 2^{w-1}$. Let P be a $(2^{w-1}+1) \times (2^{w-1}+1)$ matrix such that $P_{i,0} = 1-p$ for $i = 0, 1, \ldots, 2^{w-1}$ and $P_{i,j} = p^{HW_w(j)}(1-p)^{w-HW_w(j)}$ for $i = 0, 1, \ldots, 2^{w-1}$ and $j = 1, 2, \ldots, 2^{w-1}$. Let $\pi = (\pi_0, \pi_1, \ldots, \pi_{2^{w-1}})$ be a row vector such that $\pi P = \pi$ and $\sum_{i=0}^{2^{w-1}} \pi_i = 1$. Then the density of s'_i is*

$$den(s'_i) = \pi_i/(\textstyle\sum_{i=0}^{2^{w-1}} \pi_i l(s'_i)) \tag{3.5}$$

for $i = 0, 1, \ldots, 2^{w-1}$, if N is sufficiently large.

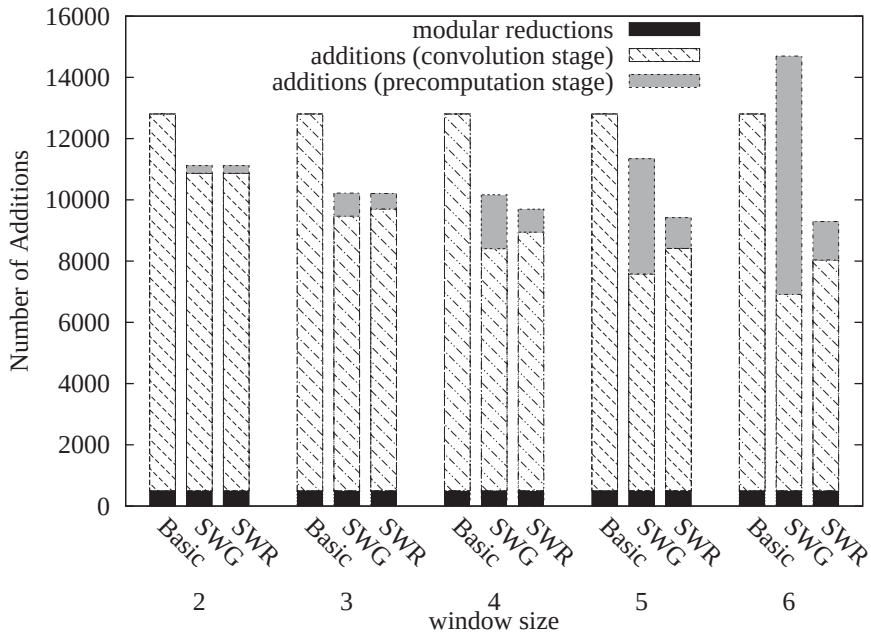

Figure 3.1: Performance Analysis of convolution algorithms for $(N, d) = (251, 48)$

Table 3.1: Performance comparison of convolution methods (Number of integer additions)

(N, d)	Basic	SWG/SWG_I	SWR/SWR_I
$(251, 48)$	$51N$	$40.501N(w = 4)$	$36.905N(w = 7)$
$(347, 66)$	$69N$	$52.022N(w = 4)$	$47.254N(w = 8)$
$(397, 74)$	$77N$	$57.460N(w = 4)$	$51.929N(w = 9)$
$(491, 91)$	$94N$	$68.483N(w = 4)$	$61.388N(w = 9)$
$(587, 108)$	$111N$	$79.589N(w = 4)$	$70.709N(w = 10)$
$(787, 140)$	$143N$	$99.797N(w = 5)$	$88.308N(w = 12)$

Proof. The proof is straightforward. □

Now, the values e_i and d_i can be estimated by $e_i = N \times den(s_{i+1})$ for $i = 0, 1, \ldots, w-1$ and $d_i = N \times den(s'_{\lceil i/2 \rceil})$ for $i = 1, 3, \ldots, 2^w - 1$ using (3.4) and (3.5). Hence we may compare the complexity of convolution algorithms using (3.2) and (3.3). In Table 3.1, we compare the performance of the basic method (Algorithm 3.1) with those of SWG and SWR. We assume $\mathbf{A_P} \approx \mathbf{A_C}$ at this point for the sake of simplicity in analysis. Note that this assumption is a conservative one. We will see in Section 3.5 that the performance gain of SWG and SWR measured from experiments is greater than the estimation based on this assumption. We also set 1 $\mathbf{MR} \approx 2$ $\mathbf{A_C}$, since the ratio of the time for one reduction mod q to the time for one addition is roughly 2 according to our experiments. Figure 3.1 shows the estimated numbers of operations for the parameter set $(N, d) = (251, 48)$. We plotted the numbers of additions for the precomputation stage and the convolution stage separately.

Algorithm 3.3 *Parallel* Basic Convolution Algorithm

Input: b (array of d locations for '1' representing the binary polynomial $a(X)$); $c(X)$ (general polynomial); N (number of coefficients in a polynomial).
Output: t (array representing $t(X) = a(X) * c(X)$).

1: **parallel for** $0 \leq j < 2N$ **do**
2: $t_j \leftarrow 0$
3: **for** $0 \leq j < d$ **do**
4: **parallel for** $0 \leq k < N$ **do**
5: $t_{k+b[j]} \leftarrow t_{k+b[j]} + c_k$
6: **parallel for** $0 \leq j < N$ **do**
7: $t_j \leftarrow (t_j + t_{j+N}) \bmod q$

According to our estimation, the two sliding window methods, SWG and SWR, are expected to reduce the number of additions by up to 30.2% and 38.3%, respectively. The gain becomes greater in parameter sets with large N such as $N = 787$. As visually verified in Figure 3.1, SWR is expected to show much better performance than SWG, because the former dramatically cuts down the number of additions in the precomputation stage with only a small additional cost in the convolution stage compared to the latter.

3.4 Implementations Using GPU

In this section, we propose the parallel implementation of the convolution algorithms in GPU. Our experiment was done on a CUDA-based GPGPU platform equipped with NVIDIA's GTX275 GPU. The GTX275 GPU has 30 multiprocessors, each of which contains 8 scalar processors. The convolution operation has a structure suitable for parallelization by its nature. Parallelism is realized at two levels: one encryption is done in parallel by 8 scalar processors on a multiprocessor and each multiprocessor performs independent encryption

Algorithm 3.4 *Parallel* Sliding Window Method

Input: b_i for $0 \leq i \leq w-1$ (arrays representing the e_i positions of p_i in the binary polynomial $a(X)$); $c(X)$ (general polynomial); N (number of coefficients in a polynomial); w (window size).
Output: t (array representing $t(X) = a(X) * c(X)$).

1: **for** $0 \leq j < w-1$ **do**
2: $\quad c_{j+N} \leftarrow c_j$
3: **for** $1 \leq i \leq w-1$ **do**
4: $\quad$ **parallel for** $0 \leq j < N$ **do**
5: $\quad\quad T_i[j] \leftarrow c_j + c_{j+i}$
6: Let $T_0 = c$, i.e., $T_0[k] = c_k$ for $0 \leq k < N$.
7: **parallel for** $0 \leq j < 2N$ **do**
8: $\quad t_j \leftarrow 0$
9: **for** $0 \leq i \leq w-1$ **do**
10: $\quad$ **for** $0 \leq j < e_i$ **do**
11: $\quad\quad$ **parallel for** $0 \leq k < N$ **do**
12: $\quad\quad\quad t_{k+b_i[j]} \leftarrow t_{k+b_i[j]} + T_i[k]$
13: **parallel for** $0 \leq j < N$ **do**
14: $\quad t_j \leftarrow (t_j + t_{j+N}) \bmod q$

using a distinct input. As a result, 30 encryptions are performed at the same time. We designed parallel threads for scalar processors in such a way that the k-th thread is in charge of accumulation of c_k. The parallel version of Algorithm 3.1 is shown in Algorithm 3.3. That is, the k-th thread computes $t_{k+b[0]} \leftarrow t_{k+b[0]} + c_0$ when $j = 0$, $t_{k+b[1]} \leftarrow t_{k+b[1]} + c_0$ when $j = 1$, and so on. However, the recoding procedures are not suitable for parallelization, because if they are parallelized, arrays b and b_i should be accessed by many concurrent threads. Thus, we decided to perform recoding on the CPU outside the GPU, and the recoding information is copied to the device memory in the GPU.

The parallel version of Algorithm 3.2 (SWR) is also shown in Algorithm 3.4. Similar to the basic convolution algorithm, both the precomputation stage and the convolution stage can be easily parallelized, but the recoding stage is seri-

Table 3.2: Timings for NTRU encryption on Pentium (μs)

(N, d)	Basic	SWG_I	SWR_I
$(251, 48)$	101.5	78.4 $(w = 4)$	72.2 $(w = 7)$
$(347, 66)$	186.3	136.8 $(w = 4)$	124.5 $(w = 8)$
$(397, 74)$	236.4	171.0 $(w = 4)$	154.3 $(w = 9)$
$(491, 91)$	354.4	249.7 $(w = 4)$	223.3 $(w = 9)$
$(587, 108)$	500.6	345.0 $(w = 4)$	305.5 $(w = 10)$
$(787, 140)$	865.2	569.1 $(w = 5)$	507.1 $(w = 12)$

ally executed on the CPU. Because the parallel algorithm of SWG is similar to that of SWR, we omit it.

3.5 Experimental Results

In this section, we present experimental results to verify the performance gains of the new convolution algorithms. Our experiments are done on two distinct platforms, i.e., a Pentium IV 3.0GHz CPU with 1.0GB RAM and an NVIDIA GTX275 GPGPU platform with 240 processing units [50]. The experimental results on a CPU (Tables 3.2 and 3.3) are provided by Jeong Eun Song.

First, Table 3.2 shows the experimental results over a CPU. According to the experimental results, SWG_I and SWR_I reduced the NTRU encryption time by 22.8% $(N = 251, w = 4)$ to 34.2% $(N = 787, w = 5)$ and 28.9% $(N = 251, w = 7)$ to 41.4% $(N = 787, w = 12)$, respectively. Among the two sliding window methods, SWR_I shows much better performance than SWG_I,

Table 3.3: Performance of NTRU encryption on Pentium with off-line precomputation: timing (number of precomputed values)

(N, d)	w	SWG	SWR
$(251, 48)$	4	68.0μs $(7N = 1757)$	70.9μs $(3N = 753)$
	6	57.7μs $(31N = 7781)$	65.6μs $(5N = 1255)$
$(787, 140)$	4	551.0μs $(7N = 5509)$	581.4μs $(3N = 2361)$
	6	456.6μs $(31N = 24397)$	521.7μs $(5N = 3935)$

which coincides with the analysis in the previous section.

In some situations, information related to the recipient's public key can be preprocessed. By setting $c(X) \leftarrow h(X)$, table T_i in Algorithm 3.2 can be computed off-line. In this case, we cannot use the interleaving methods such as SWG_I and SWR_I. Table 3.3 demonstrates the experimental results for the variants with off-line precomputation. The off-line precomputation reduces the encryption time to almost a half of the basic method using a reasonable amount of memory.

Now, we present the experimental results for the parallel versions of the convolution algorithms. Our experiment was done on a CUDA-based GPGPU platform equipped with NVIDIA's GTX275 GPU. Table 3.4 shows the measured timings. 'Recoding' and 'Main' in encryption represent the timings required to recode a random polynomial r on the CPU and perform the main operation $e = r * h + m \bmod q$ on the GPU, respectively. 'Copy' represents the timings for copying m's, h, and recoded r's from the main memory to the GPU memory. For the encryption with general sliding window method,

Table 3.4: Comparison of encryption and decryption speeds using CUDA over NVIDIA GTX275 GPU with parameter $(N, d) = (251, 48)$ (μs/encryption and μs/decryption)

Method	Encryption	Decryption
	Recoding + Main + Copy	Main + Copy
Basic	$0.743 + 0.743 + 1.655 = 3.141$	$0.749 + 1.188 = 1.937$
SWG^*	$0.587 + 0.528 + 1.648 = 2.763$	$0.631 + 1.206 = 1.838$
$SWR^\dagger$	$0.811 + 0.553 + 1.643 = 3.007$	$0.553 + 1.204 = 1.757$

* off-line precomputation for encryption ($w = 5$) and on-line precomputation for decryption ($w = 4$), $\dagger$ $w = 7$

we only considered the version with off-line precomputation. In this case, the maximum value of possible w is $w = 5$ due to the constrained shared memory of the GPU. According to the table, the main encryptions of the basic method are accelerated by up to 28.9% and 25.6% using SWG and SWR, respectively. This result implies that our sliding window method is equally effective for parallel processing. However, the overall speed-up is less than this value due to the bottlenecks such as memory copy operations. The overall speed-up in decryption is larger than that in encryption, because the recoding of a private F can be done off-line. In addition, the copying time in decryption is less than that of encryption, because a decryption requires fewer elements than an encryption. Meanwhile, the parallel encryptions of Basic convolution, SWG and SWR on GPU is 32, 23, 24 times faster than the serial versions of those

on CPU, respectively.

Most parts in this chapter are quoted from [41, 42], which is allowed by my co-authors. The sliding window methods in Section 3.2 are proposed by Mun-Kyu Lee and myself, and the implementations and their results on a CPU in Section 3.5 are provided by Jeong Eun Song. The theoretical meaning of the improvement by SWR, which reduces the asymptotic complexity of the precomputation stage to a linear order from the exponential order of SWG, was discovered by Kunsoo Park.

Chapter 4

Conclusion

In this thesis, we have considered and presented the following two results on parallel implementations of two cryptographic algorithms, the rainbow method and the convolution product, using GPU.

First, we proposed the parallel implementations of the rainbow method in a GPU+CPU heterogeneous system. For achieving the best performance, we split the online phase into two procedures: the online chain+lookup procedure and the regenerating chain procedure, which eliminates the warp serialization. We gave a complete analysis of the effect of multiple checkpoints for the non-perfect rainbow table, and we made use of it for load balancing between GPU and CPU. Also, we changed the order of the online chain generation for the heterogeneous system. According to our experimental result, our implementation is faster than any other implementations on GPU.

Second, we proposed sliding window methods for accelerating convolution product computation in the NTRU cryptosystem. The first proposed method speeds up the computation by reusing fixed-length patterns in the binary coef-

ficients of an NTRU polynomial. The second one further improves performance by using the fact that the distributions of '1's and '0's are not symmetric in NTRU. It considers only the special patterns with exactly two '1's, and reduces the overheads for window construction from an exponential order in w to a linear order. Our Markov chain analysis proves that our methods significantly reduce the amount of computations, and our various experiments back up this analysis. According to the experimental results, our second sliding window method accelerates the encryption and decryption operations by up to 41% on a Pentium IV. We also presented the parallel algorithms of our proposed methods, and we showed that our optimization techniques are also useful in parallel processing environments. According to our experimental results on GPU, the main convolution operation is accelerated by up to 26% by the sliding window method.

Bibliography

[1] Cryptohaze gpu rainbow cracker, `https://www.cryptohaze.com`.

[2] Ophcrack, `http://ophcrack.sourceforge.net`.

[3] RainbowCrack Project, `http://project-rainbowcrack.com`.

[4] Maplesoft, Maple 12 user manual, 2007.

[5] Nvidia, Nvidia's next generation CUDA compute architecture: Fermi, 2009.

[6] Nvidia, CUDA best practices guide, 2012.

[7] Nvidia, CUDA C programming guide, 2012.

[8] Gildas Avoine, Pascal Junod, and Philippe Oechslin. Characterization and improvement of time-memory trade-off based on perfect tables. *ACM Trans. Inf. Syst. Secur.*, 11(4), 2008.

[9] Daniel V. Bailey, Daniel Coffin, Adam Elbirt, Joseph H. Silverman, and Adam D. Woodbury. NTRU in constrained devices. In *Cryptographic Hardware and Embedded Systems – CHES 2001*, volume 2162 of *LNCS*, pages 262–272. Springer, 2001.

[10] Elad Barkan, Eli Biham, and Adi Shamir. Rigorous bounds on cryptanalytic time/memory tradeoffs. In *CRYPTO*, pages 1–21, 2006.

[11] Daniel J. Bernstein, Tien-Ren Chen, Chen-Mou Cheng, Tanja Lange, and Bo-Yin Yang. ECM on graphics cards. In *EUROCRYPT*, pages 483–501, 2009.

[12] Alex Biryukov, Sourav Mukhopadhyay, and Palash Sarkar. Improved time-memory trade-offs with multiple data. In *Selected Areas in Cryptography*, pages 110–127, 2005.

[13] Johan Borst, Bart Preneel, and Joos 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.

[14] André Rigland Brodtkorb, Christopher Dyken, Trond Runar Hagen, Jon M. Hjelmervik, and Olaf O. Storaasli. State-of-the-art in heterogeneous computing. *Scientific Programming*, 18(1):1–33, 2010.

[15] Don Coppersmith and Adi Shamir. Lattice attacks on NTRU. In *Eurocrypt 97*, volume 1233 of *LNCS*, pages 52–61. Springer, 1997.

[16] Dorothy E. Denning. *Cryptography and Data Security.* Addison-Wesley, 1982. p.100.

[17] Amos Fiat and Moni Naor. Rigorous time/space trade-offs for inverting functions. *SIAM J. Comput.*, 29(3):790–803, 1999.

[18] Nicolas Gama, Nick Howgrave-Graham, and Phong Q. Nguyen. Symplectic lattice reduction and NTRU. In *Eurocrypt 2006*, volume 4004 of *LNCS*, pages 233–253. Springer, 2006.

[19] Nicolas Gama and Phong Q. Nguyen. New chosen-ciphertext attacks on NTRU. In *PKC 2007*, volume 4450 of *LNCS*, pages 89–106. Springer, 2007.

[20] Gunnar Gaubatz, Jens-Peter Kaps, and Berk Sunar. Public key cryptography in sensor networks-revisited. In *ESAS 2004*, volume 3313 of *LNCS*, pages 2–18. Springer, 2004.

[21] Olle Häggström. *Finite Markov chains and algorithmic applications*. Cambridge University Press, 2002.

[22] Martin E. Hellman. A cryptanalytic time-memory trade-off. *Information Theory, IEEE Transactions on*, 26(4):401 – 406, July 1980.

[23] Jens Hermans, Frederik Vercauteren, and Bart Preneel. Speed records for NTRU. In *CT-RSA*, pages 73–88, 2010.

[24] Jens Hermans, Frederik Vercauteren, and Bart Preneel. Speed records for NTRU. In *Topics in Cryptology – CT-RSA 2010*, volume 5985 of *LNCS*, pages 73–88. Springer, 2010.

[25] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. NTRU: A new high speed public key cryptosystem. Preprint; presented at the rump session of Crypto 96.

[26] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. NTRU: A ring-based public key cryptosystem. In *Algorithmic Number Theory – ANTS III*, volume 1423 of *LNCS*, pages 267–288. Springer, 1998.

[27] Jeffrey Hoffstein and Joseph H. Silverman. Optimizations for NTRU. In *Proceedings of Public-Key Cryptography and Computational Number Theory*, 2000.

[28] Jeffrey Hoffstein and Joseph H. Silverman. Random small Hamming weight products with applications to cryptography. *Discrete Applied Mathematics*, 130:37–49, 2003.

[29] Jin Hong. The cost of false alarms in Hellman and rainbow tradeoffs. *Des. Codes Cryptography*, 57(3):293–327, 2010.

[30] Jin Hong, Ga Won Lee, and Daegun Ma. Analysis of the parallel distinguished point tradeoff. In *INDOCRYPT*, pages 161–180, 2011.

[31] Jin Hong and Sunghwan Moon. A comparison of cryptanalytic tradeoff algorithms. *Journal of Cryptology*. To appear.

[32] Jin Hong and Palash Sarkar. New applications of time memory data tradeoffs. In *ASIACRYPT*, pages 353–372, 2005.

[33] Nick Howgrave-Graham. A hybrid lattice-reduction and meet-in-the-middle attack against NTRU. In *Advances in Cryptology – Crypto 2007*, volume 4622 of *LNCS*, pages 150–169. Springer, 2007.

[34] Nick Howgrave-Graham, Phong Q. Nguyen, David Pointcheval, John Proos, Joseph H. Silverman, Ari Singer, and William Whyte. The impact

of decryption failures on the security of NTRU encryption. In *Crypto 2003*, volume 2729 of *LNCS*, pages 226–246. Springer, 2003.

[35] Nick Howgrave-Graham, Joseph H. Silverman, and William Whyte. Choosing parameter sets for NTRUEncrypt with NAEP and SVES-3. In *CT-RSA 2005*, volume 3376 of *LNCS*, pages 118–135. Springer, 2005.

[36] IEEE P1363.1. IEEE standard specification for public key cryptographic techniques based on hard problems over lattices, 2009.

[37] Éliane Jaulmes and Antoine Joux. A chosen-ciphertext attack against NTRU. In *Advances in Cryptology – Crypto 2000*, volume 1880 of *LNCS*, pages 20–35. Springer, 2000.

[38] Jung Woo Kim, Jungjoo Seo, Jin Hong, Kunsoo Park, and Sung-Ryul Kim. High-speed parallel implementations of the rainbow method in a heterogeneous system. In *INDOCRYPT 2012*, volume 7668 of *LNCS*, pages 303–316. Springer, 2012.

[39] Donald Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, Reading, Massachusetts, 3rd edition, 1998.

[40] Koji Kusuda and Tsutomu 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.

[41] Mun-Kyu Lee, Jung Woo Kim, Jeong Eun Song, and Kunsoo Park. Sliding window mothod for NTRU. In *Applied Cryptography and Network*

Security – ACNS 2007, volume 4521 of *LNCS*, pages 432–442. Springer, 2007.

[42] Mun-Kyu Lee, Jung Woo Kim, Jeong Eun Song, and Kunsoo Park. Efficient implementation of ntru cryptosystem using sliding window methods. *IEICE Transactions*, 2013. To appear.

[43] Christoph Ludwig. A faster lattice reduction method using quantum search. In *Algorithms and Computation – ISAAC 2003*, volume 2906 of *LNCS*, pages 199–208. Springer, 2003.

[44] Daegun Ma and Jin Hong. Success probability of the hellman trade-off. *Information Processing Letters*, 109(7):347 – 351, 2009.

[45] Svetilin A. Manavski. CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In *ICSPC*, 2007.

[46] Petros Mol and Moti Yung. Recovering NTRU secret key from inversion oracles. In *PKC 2008*, volume 4939 of *LNCS*, pages 18–36. Springer, 2008.

[47] Sourav Mukhopadhyay and Palash Sarkar. Application of LFSRs in time/memory trade-off cryptanalysis. In *WISA*, pages 25–37, 2006.

[48] Phong Q. Nguyen and David Pointcheval. Analysis and improvements of NTRU encryption paddings. In *Advances in Cryptology – Crypto 2000*, volume 2442 of *LNCS*, pages 210–225. Springer, 2002.

[49] John Nickolls and William J. Dally. The GPU computing era. *IEEE Micro*, 30(2):56–69, 2010.

[50] NVIDIA. Compute unified device architecture programming guide (2007). http://developer.nvidia.com.

[51] Philippe Oechslin. Making a faster cryptanalytic time-memory trade-off. In *CRYPTO*, pages 617–630, 2003.

[52] Heejin Park, Kunsoo Park, and Yookun Cho. Analysis of the variable length nonzero window method for exponentiation. *Computers and Mathematics with Applications*, 37(7):21–29, 1999.

[53] Katja Schmidt-Samoa, Olivier Semay, and Tsuyoshi Takagi. Analysis of fractional window recoding methods and their application to elliptic curve cryptosystems. *IEEE Transactions on Computers*, 55(1):48–57, 2006.

[54] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1997.

[55] François-Xavier Standaert, Gaël Rouvroy, Jean-Jacques Quisquater, and Jean-Didier Legat. A time-memory tradeoff using distinguished points: New analysis & FPGA results. In *CHES*, pages 593–609, 2002.

[56] Robert Szerwinski and Tim Güneysu. Exploiting the power of GPUs for asymmetric cryptography. In *CHES*, pages 79–99, 2008.

[57] Wenhao Wang, Dongdai Lin, Zhenqi Li, and Tianze Wang. Improvement and analysis of VDP method in time/memory tradeoff applications. In *ICICS*, pages 282–296, 2011.

초 록

그래픽 처리 장치(GPU)의 컴퓨팅 파워는 급속도로 증가하고 있으며, RSA, ECC, NTRU, 그리고 AES와 같은 암호 알고리즘의 GPU에서의 범용 컴퓨팅(GPGPU)에 대한 광범위한 연구가 진행되어 왔다. GPGPU의 등장으로 상용 컴퓨터는 복잡한 이기종 GPU+CPU 시스템이 되었다. 이 새로운 아키텍처는 새로운 도전과 고성능 컴퓨팅의 기회를 우리에게 안겨주고 있다.

이 논문에서 우리는 암호 알고리즘의 GPU를 이용한 병렬 구현에 대한 두 가지 연구 결과를 제시한다. 첫째, 우리는 이기종 GPU+CPU 시스템에서 가장 효율적인 시간-메모리 절충 암호분석 방법으로 알려진 Rainbow 방법의 고속 병렬 구현 방법을 제시한다. 또한 오경보(false alarm) 비용의 절감을 위한 checkpoint 기법을 분석하고, 이를 GPU와 CPU 사이의 부하조절(load balancing)을 위해 활용한다. 우리의 구현 방법이 GPU를 이용한 다른 구현인 RainbowCrack과 Cryptohaze에 비해 각각 1.36배와 2.18배 빠르다.

둘째, 우리는 NTRU 암호시스템의 효율적인 구현들과 그것들의 병렬화에 대한 연구 결과를 제시한다. 합성곱은 NTRU에서 가장 많은 시간이 소요되는 연산이며, 우리는 NTRU 다항식 계수의 반복되는 패턴이 나타나는 특징을 이용하여 합성곱 연산의 속도를 빠르게 할 수 있다. 우리는 이를 이용한 합성곱의 효율적인 알고리즘을 제시하고, 새 알고리즘의 성능을 마코프 체인을 이용하여 분석한다. 또한 우리는 이를 CPU와 GPU 상에서 구현한다.

분석과 실험 결과에 따르면, 우리가 제시한 새 알고리즘은 기존의 합성곱 연산에 비해 41%까지 빠른 성능을 보인다. 그리고 GPU를 이용한 구현은 CPU를 이용한 구현에 비해 23배 이상 나은 성능을 보인다.

주요어: GPGPU, CUDA, 이기종 계산, 암호해독, 시간-메모리 절충 기법, Rainbow 방법, NTRU, 합성곱, 슬라이딩 윈도우 방법
학 번: 2006- 21173