GPU-BASED PARALLEL COMPUTING MODELS AND IMPLEMENTATIONS FOR TWO-PARTY PRIVACY-PRESERVING PROTOCOLS

A Dissertation

by

SHI PU

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	Jyh-Charn Liu
Committee Members,	Riccardo Bettati
	Guofei Gu
	Peng Li
Head of Department,	Nancy Amato

December 2013

Major Subject: Computer Science

ABSTRACT

In (two-party) privacy-preserving-based applications, two users use encrypted inputs to compute a function without giving out plaintext of their input values. Privacy-preserving computing algorithms have to utilize a large amount of computing resources to handle the encryption-decryption operations. In this dissertation, we study optimal utilization of computing resources on the graphic processor unit (GPU) architecture for privacy-preserving protocols based on *secure function evaluation* (SFE) and the *Elliptic Curve Cryptographic* (ECC) and related algorithms. A number of privacy-preserving protocols are implemented, including *private set intersection* (PSI), *secret handshaking* (SH), secure *Edit distance* (ED) and *Smith-Waterman* (SW) problems. PSI is chosen to represent ECC point multiplication related computations, SH for bilinear pairing, and the last two for SFE-based *dynamic programming* (DP) problems. They represent different types of computations, so that in-depth understanding of the benefits and limitations of the GPU architecture for privacy preserving protocols is gained.

For SFE-based ED and SW problems, a *wavefront* parallel computing model on the CPU-GPU architecture under the semi-honest security model is proposed. Low level parallelization techniques for GPU-based gate (de-)garbler, synchronized parallel memory access, pipelining, and general GPU resource mapping policies are developed. This dissertation shows that the GPU architecture can be fully utilized to speed up SFE-based ED and SW algorithms, which are constructed with billions of garbled gates, on a contemporary GPU card GTX-680, with very little waste of processing cycles or memory space.

For PSI and SH protocols and underlying ECC algorithms, the analysis in this research shows that the conventional Montgomery-based number system is more friendly to the GPU architecture than the Residue Number System (RNS) is. Analysis on experiment results further shows that the *lazy reduction* in higher extension fields can have performance benefits only when the GPU architecture has enough fast memory. The resulting Elliptic curve Arithmetic GPU Library (EAGL) can run 3350.9 *R-ate* (bilinear) pairing/sec, and 47000 point multiplication/sec at the 128-bit security level, on one GTX-680 card. The primary performance bottleneck is found to be lacking of advanced memory management functions in the contemporary GPU architecture for bilinear pairing operations. Substantial performance gain can be expected when the on-chip memory size and/or more advanced memory prefetching mechanisms are supported in future generations of GPUs.

ACKNOWLEDGEMENTS

I would like to thank my advisor and committee chair, Dr. Jyh-charn Liu, for his supervision and support in last five years. I joined the group in 2008 and worked in several aspects before Dr. Liu suggested me to work on the research area related to parallelization model design for cryptographic primitives. This area is an interesting subject to study and full of theoretical and practical challenge, but it fits my professional strength in mathematics and architecture so I enjoy the research a lot. As such, I appreciate him for his advice, suggestion, and generous supports.

I would like to thank my committee members, Dr. Bettati, Dr. Gu, and Dr. Li for suggestions and help on research and comments and feedback on dissertation. I also would like to thank Dr. Papanikolas in the Mathematic department of TAMU for his lecturing on the correctness of the conventional Montgomery number system.

I want to thank Pu Duan, who was a senior student in our group and an expert in Elliptic Curve Cryptography. Pu has given me lots of help after I started my research subject in this area. He is very nice and patient with all my questions.

I want to thank every member in the group who also helped a lot and I could not finish this work without their help.

Finally, I would like to thank my mother and father for their consistent support and encouragement.

NOMENCLATURE

ECC	Elliptic Curve Cryptography
PSI	Private Set Intersection
SH	Secret Handshake
CUDA	Compute Unified Device Architecture
DP	Dynamic Programming
SFE	Secure Function Evaluation
PIR	Private Information Retrieval
DH	Diffie-Hellman
ED	Edit-Distance
SW	Smith-Waterman
GC	Garbled Circuit
ML	Mill’s Loop
FE	Final Exponentiation

TABLE OF CONTENTS

Page

ABSTRACT ii

ACKNOWLEDGEMENTS iv

NOMENCLATURE v

TABLE OF CONTENTS vi

LIST OF FIGURES viii

LIST OF TABLES x

CHAPTER I INTRODUCTION 1

1.1 Summary of Research Tasks 3

CHAPTER II BACKGROUND KNOWLEDGE 9

2.1 Secure Function Evaluation 9
2.2 Private Set Intersection and Secret Handshake 12
2.3 Elliptic Curve Cryptographic Algorithms 13
2.4 Modern GPU Architecture 16

CHAPTER III RELATED WORK 19

3.1 Secure Function Evaluation 19
3.2 Point Multiplication and Bilinear Pairing 21

CHAPTER IV LARGE SCALE PRIVACY-PRESERVING ED/SW PROBLEMS ON GPU 24

4.1 GC-level: GPU-based Gate (De-) Garbler 27
4.2 DP-level: Computing SFE-based Edit-Distance 31
4.2.1 GPU Mapping Policies 34
4.2.2 Memory Management and Pipelined Scheduling 38
4.2.3 Experiment Result and Analysis 43
4.3 DP-level: Computing SFE-based Smith-Waterman 45
4.3.1 GPU Mapping Policies 49
4.3.2 Pipelined Scheduling 51

4.3.3 Experiment Results and Analysis 53
4.4 Summary 54

CHAPTER V PARALLEL ECC ALGORITHMS ON GPU 57

5.1 Computation Requirements in PSI and SH 58
5.2 Computing Model of Multi-precision Arithmetic 63
5.3 Sliding Window–based Point Multiplication 75
5.4 Optimization of Arithmetic in Extension Fields 81
5.5 Experiment Results of Point Multiplication 87
5.6 Experiment Results of Bilinear Pairing 90
5.7 Analysis of System Bottleneck 93

CHAPTER VI CONCLUSION AND FUTURE WORK 98

REFERENCES 102

LIST OF FIGURES

FIGURE Page

1.1 The Design Space of SFE, PSI and SH Implementations 4

1.2 Research Overview 6

2.1 An Example of SFDL and SHDL 11

2.2 (a) Point Addition, (b) Point Doubling on $E(F_q)$ 14

2.3 Miller's Algorithm for *R-ate* Pairing 15

4.1 Parallel Computing Models for the (a) ED, (b) SW problems …………………….. 25

4.2 Optimization on Shared Memory Access 28

4.3 The SFE Building Block (a GC-slot at DP[i][j]) for ED 31

4.4 The Generator's Resource Mapping Policy: CSMP 34

4.5 The Evaluator's GPU Resource Mapping Policy: *slice-by-slice* 37

4.6 The Pipelined Garbling & De-garbling Process (ED) 41

4.7 The Revised Smith-Waterman Algorithm 46

4.8 The SFE Building Block (a GC-slot at DP[i][j]) for SW 47

4.9 The Generator's CSMP (SW) for (a) Line 7 and (b) Lines 8-11 in Figure 4.7 ….. 49

4.10 The Evaluator's slice-by-slice Mapping Policy (SW) for (a) Line 7 and (b) Lines 8-11 in Figure 4.7 50

4.11 The Pipelined Garbling & De-garbling Process (SW) 52

5.1 The Secret Handshake between Alice and the RRS 59

5.2 The Private Sect Intersection between Alice and the RRS 61

5.3 Collision-free Memory Access of a 256-bit Variable (CI-2thread). 67

5.4 Parallel Multi-precision Multiplication $T = a \times b$ 69

5.5 Profiling Result of Bank Conflict for $T = a \times b$ (CI-2thread) 71

5.6 Implementation of *Reduction*(*T*) 72

5.7 GPU-based Point Multiplication (the Same P among Instances) 78

5.8 Online Generation of Pre-calculated Table (Different Points among Instances) .. 79

5.9 Shared/Global Memory (s/g mem) Size of Memory Usage per CI of in EAGL's Point Multiplication and Bilinear Pairing 95

LIST OF TABLES

TABLE Page

4.1 Maximum Possible Values of Inputs and Intermediate Results in DP[*i*][*j*] 33

4.2 Pipeline Execution Time Break Down (ED) 42

4.3 Major Memory Utilization on the Generator Side (ED) 44

5.1 Execution Time of INT32 and SPF arithmetic on GTX-680 63

5.2 Performance of 11938*mul* + 8321*reduct* in Various Thread Counts per Instance. 73

5.3 Performance of 1000 Modular Multiplication in F_q^4 82

5.4 Throughput Fluctuation as Shared Memory Usage per CI Changes 87

5.5 Comparison of Throughputs of Point Multiplication among GPU Implementations 88

5.6 CPU vs. GPU-based Point Multiplication Implementations 89

5.7 Comparison of Execution Time Among GPU Implementations 91

5.8 Comparison of Throughput, EAGL vs. CPU-based Solutions 92

CHAPTER I

INTRODUCTION

In their seminal work [96] published in 1982, Yao *et al.* formulated the *Billionaire* problem, in which two principals want to compare their wealth without giving each other their actual amounts, and proposed a very general way to do so, called *Secure Function Evaluation* (SFE). In SFE, the two principals, who are commonly called the *generator* and the *evaluator*, use encrypted inputs to jointly compute an arbitrary function on two interconnected computers. The final computing result is revealed to one or both principal(s), but plain-text of the inputs are unrevealed to each other. A rich body of literature has been developed from the original Billionaire problem to the field of *privacy-preserving computing*. Major privacy-preserving protocols include, but are not limited to SFE, *Private Set Intersection* (PSI) [45][26][31][53][55] where two principals jointly compute the common element(s) for their two input sets without giving out to each other the plain-text of all distinct elements, *Secret handshake* (SH) [26][50] where two principals mutually authenticate each other based on privacy-preserving equality checking of a one-time registered *group secret* [26], *Private Information Retrieval* (PIR) [20] where the index of a database query is unrevealed to the owner of the database in a query transaction, and *Homomorphic encryption* (HE) [33] where two principals use encrypted inputs to jointly compute an arithmetic function.

Some examples of privacy-preserving applications aim to answer questions, such as "which input is bigger: ($x_1 \lessgtr x_2$)", "Edit distance of two strings: ED(s_1, s_2)", "intersection of two sets of strings: COMMON(S_1, S_2)", etc. While most applications can be implemented by SFE, non-SFE based solutions have also been developed. For example, several studies [26][31][53][55] implemented PSI based on *Elliptic Curve Cryptography* (ECC) [56], although PSI could also be formulated as an SFE function COMMON(S_1, S_2) [45]. Another important example of non-SFE based privacy-preserving protocols is SH, ECC [26][50], RSA [84] or *Diffie-Hellman* (DH) [25] crypto systems can be used to implement SH.

In this dissertation, we focus on the design and the performance evaluation of SFE and ECC-based PSI and SH on an integrated CPU-GPU system architecture. For SFE, the main performance challenge is to run a vast number (billions or more) of *garbled gates* [96], which are composed of *oblivious-transfer* (OT) [82] and block-cipher operations (for example, SHA [73] and AES [74]), with few waste of computational strength provided by the CPU-GPU system architecture. The main performance challenges for ECC to achieve the highest throughput on GPU include (1) the low level data access model for arithmetic operands, (2) matching of the number system with respect to architecture, (3) the parallel computing model for arithmetic operations, and (4) the efficiency of arithmetic optimization techniques. Overall, for optimal performance outcomes, the system needs to have tight synchronization of processing steps, memory accesses, and GPU-CPU memory swapping, so that idle cycles and swapping overheads are minimized, while the degree of parallelism is maximized. For ECC, the four factors are closely related to each other. As such, our design process is coupled with a system performance characteristics process, so that lessons learnt from the study can help identify system bottlenecks for future improvement of architectures and algorithms.

1.1 Summary of Research Tasks

An overview of the overall design space explored in this dissertation is illustrated in Figure 1.1, in which green blocks represent major technical elements of our work. To investigate SFE related design problems on the CPU-GPU architecture, we adopt the specification languages SHDL (*Secure Hardware Description Language*) proposed in *Fairplay* [65] to describe the SFE functions studied in this dissertation. For the study cases, we select privacy-preserving computation of the *dynamic programming* (DP) for *Edit Distance* (ED) and *Smith-Waterman* (SW) [5]. ED (SW) has been used for privacy-preserving assessment of dissimilarity (similarity) of two genomic sequences [52]. By using SHDL, the SFE-based ED problem and the SFE-based SW problem are represented by two networks of interconnected garbled (Boolean) gates, respectively. In this dissertation, these two problems are solved based on a divide-and-conquer strategy to partition the interconnected garbled gates, and encryption/decryption operations are executed on the GPU in batches. The resulting parallel code follows a *wavefront* computing pattern, which needs highly synchronized memory accesses. Parallelization of the two studied cases represent much higher challenges than parallelization of SFE-based AES [43], Hamming distance [43][52], RSA [30], or Dot product [46], whose intermediate results are immediately re-used and then discarded. According to [52], the memory requirement for the SFE-based ED (SW) problem can be tens of Giga Bytes when the input length of the ED (SW) problem exceeds 10^3 (10^2). As such, to study spatial efficient memory management mechanisms for SFE and the associated low-cost synchronization strategies for reading and writing intermediate results on the CPU-GPU architecture, we set 5000×5000 as the problem size of the ED problem, and 60×60 as the problem size of the SW problem.

In terms of the security model for SFE, we adopt the semi-honest security model [44][52], where principals follow the SFE protocol but may infer the input data from cipher-text of intermediate results produced during protocol steps. Following [52], we adopt the *ultra-short* security level proposed in TASTY [40], which is equivalent to the 80-bit security level.

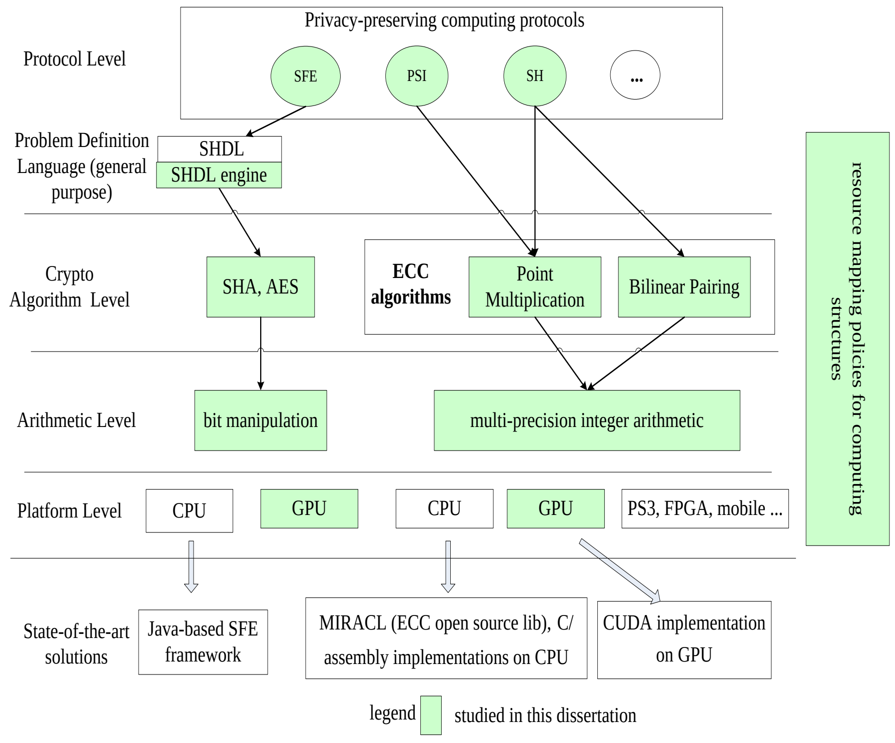

Privacy-preserving computing protocols
Protocol Level
SFE
PSI
SH
...
Problem Definition Language (general purpose)
SHDL
SHDL engine
Crypto Algorithm Level
SHA, AES
ECC algorithms
Point Multiplication
Bilinear Pairing
Arithmetic Level
bit manipulation
multi-precision integer arithmetic
Platform Level
CPU
GPU
CPU
GPU
PS3, FPGA, mobile ...
State-of-the-art solutions
Java-based SFE framework
MIRACL (ECC open source lib), C/ assembly implementations on CPU
CUDA implementation on GPU
resource mapping policies for computing structures
legend
studied in this dissertation

Figure 1.1 The Design Space of SFE, PSI and SH Implementations.

For our study of ECC-based PSI and SH, ECC *point multiplication* [56] is chosen for PSI, and *bilinear pairing* [87] in ECC for SH, because the vast majority of privacy-preserving protocols is based on ECC. A *Barreto-Naehrig* (BN) curve [10] is chosen for both PSI and SH as the underlying elliptic curve due to its computational efficiency [32]. BN-curve-based *R-ate* pairing [60] is adopted as the bilinear pairing algorithm due to its less (or similar) computational complexity than other variants [34][42][66]([99]) of bilinear pairing on BN curves. To achieve the 128-bit security level, $-(2^{62}+2^{55}+1)$ is used as the construction parameter of the BN curve, and the twelfth *extension field* [87] as the highest extension field [10].

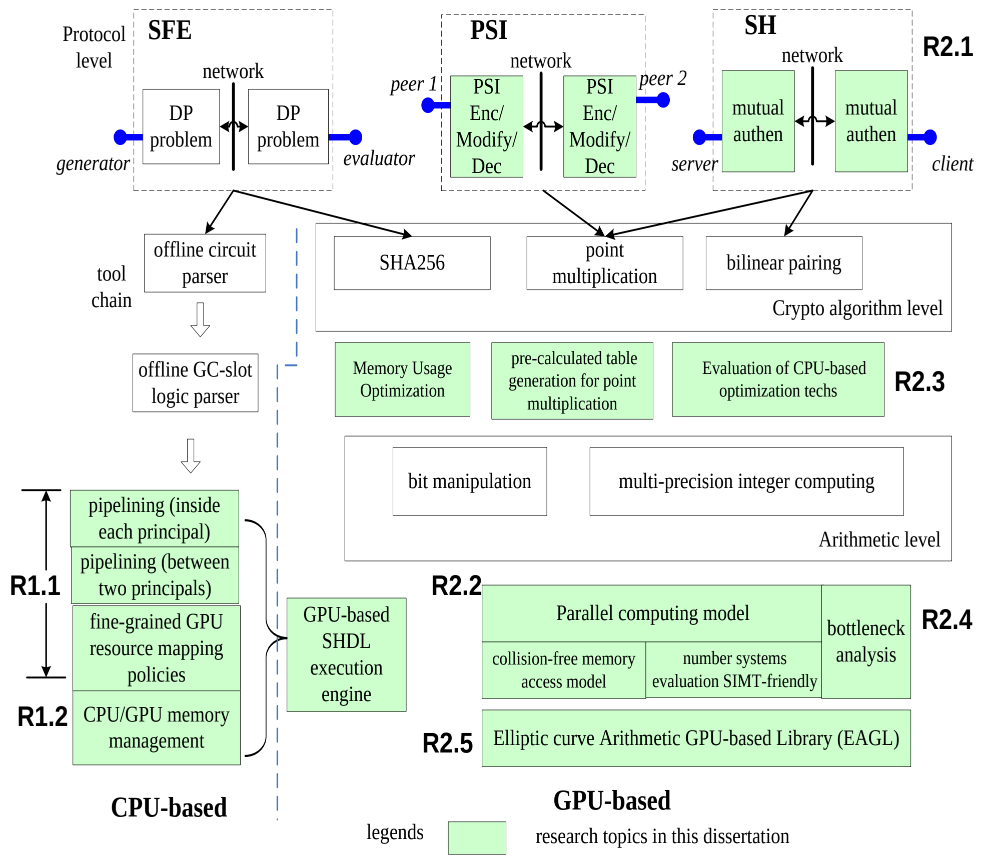
In the CPU-GPU system architecture, the GPU works as a co-processor for the CPU to perform designated computations. For SFE and ECC-based PSI and SH, computing tasks designated to run on the GPU need to be optimized with respect to the *single instruction multi-thread* (SIMT) architecture and the memory hierarchy of GPU, as well as the CPU-GPU control mechanisms, for best performance. These cross-layer resource management issues are highlighted at right side of Figure 1.1.

The major research tasks in this dissertation are listed below, and a graphic illustration of these tasks is given in Figure 1.2. Research tasks for SFE and ECC-related PSI and SH follow different paths to reflect their very different computing structures.

SFE Related Research Tasks

- (**R1.1**) Investigation of the partition policies for a network with billions of garbled gates, and the associated GPU-based resource mapping policies in order to maximize the system throughput; the pipelining mechanism of the encryption/decryption operations is also studied;

- (**R1.2**) Investigation of memory management mechanisms on CPU and GPU, and associated synchronization strategies for storing and re-using inputs/outputs of encryption/decryption operations, and intermediate de-garbled results on GPU.


Protocol level
SFE
network
DP problem
DP problem
generator
evaluator
PSI
network
peer 1
peer 2
PSI Enc/ Modify/ Dec
PSI Enc/ Modify/ Dec
SH
network
mutual authen
mutual authen
server
client
R2.1
tool chain
offline circuit parser
offline GC-slot logic parser
SHA256
point multiplication
bilinear pairing
Crypto algorithm level
Memory Usage Optimization
pre-calculated table generation for point multiplication
Evaluation of CPU-based optimization techs
R2.3
bit manipulation
multi-precision integer computing
Arithmetic level
pipelining (inside each principal)
pipelining (between two principals)
fine-grained GPU resource mapping policies
CPU/GPU memory management
R1.1
R1.2
GPU-based SHDL execution engine
R2.2
Parallel computing model
collision-free memory access model
number systems evaluation SIMT-friendly
bottleneck analysis
R2.4
R2.5
Elliptic curve Arithmetic GPU-based Library (EAGL)
CPU-based
GPU-based
legends
🟩 research topics in this dissertation

- **Figure 1.2** Research Overview

ECC Related Research Tasks

- (**R2.1**) Studying and implementing PSI and SH protocols in a case study [77] to understand the usage of parallelized point multiplication and bilinear pairing in real-world privacy-preserving applications;
- (**R2.2**) Exploring an efficient parallel computing model of low-level arithmetic operations on GPU, including the evaluation of number systems and data storage formats;
- (**R2.3**) Evaluating the suitability of optimization techniques for point multiplication and bilinear on the contemporary GPU architecture;
- (**R2.4**) Identifying major performance bottlenecks for computing point multiplication and bilinear pairing on the contemporary GPU architecture;
- (**R2.5**) Developing a GPU-based library for point multiplication and bilinear pairing, which can fit future GPU architecture without major code changes;

Some critical conclusions derived from this research are highlighted below. For SFE, the results of research tasks (**R1.1**) and (**R1.2**) show that the GK104 *Kepler* chip (marketed by NVIDIA in 2012) can be fully utilized to speed up the SFE-based ED algorithm and the SFE-based SW algorithm, with very little waste of processing cycles or memory space. For ECC, low level resource management techniques are designed and tested to eliminate major resource wastes. It is discovered in (**R2.2**) that, the *conventional Montgomery number system* [68] is more GPU-friendly than the *Residue Number System* (RNS) based number system [2] on the GK104 chip. Through (**R2.3**), it is found that the acceleration effect of the *lazy reduction* [3] technique has the best performance when it is applied to the quadratic extension field on the GK104 chip. It is found through (**R2.4**) that, since a large number of data swapping between fast on-chip cache and slow off-chip device memory are triggered by the complex computations steps of bilinear pairing, the primary performance bottleneck for bilinear pairing on the GK104 chip is lack of advanced device memory management functions. However, the GK104 chip is quite effective for speed up of point multiplication and arithmetic in the quadratic extension field. Last, but not least, is that through (**R2.5**), a library, called *Elliptic curve Arithmetic GPU Library* (EAGL), is produced to empower future generation of research in this area.

The rest of the dissertation is organized as follows. In Chapter 2, we provide an overview of SFE protocol, point multiplication and bilinear pairing algorithms, and modern GPU architecture. Chapter 3 summarizes previous studies. Chapter 4 presents the parallel computing model for the ED problem and the SW problem, respectively. In Chapter 5, we first study the usage of point multiplication and bilinear pairing in PSI and SH protocols. Then, we discuss the parallel computing model for point multiplication and bilinear pairing. We conclude with some final marks in Chapter 6.

CHAPTER II

BACKGROUND KNOWLEDGE

In this chapter, we first present the background knowledge of SFE, PSI and SH, and then the ECC algorithms used to construct ECC-based PSI and SH protocols. In the end, the contemporary GPU architecture, the *Kepler* GPU, is introduced.

2.1 Secure Function Evaluation

The Secure Function Evaluation (SFE) proposed to use the term *garbled circuit*s (GC) to implement privacy-preserving applications, such as secret auctions [17][72], biometric or genomic computation [43][44][52], facial recognition [28][40][76][85] and encryption [40][43][58]. In SFE, the *garbled circuit*s (GC), and its the fundamental build block *garbled gates*, and the roles of the two principals (the *generator* and the *evaluator*) are defined as follows:

An *n*-bit-in, 1-bit-out garbled gate G implements an *n* variable "secure" (or, "privacy-preserving") Boolean function. Same as a regular gate, a garbled gate has a truth table specified based on the Boolean function. Each input or output bit for both types of gates can be represented as a *"wire"*, but in a garbled gate each wire is associated with a pair of random integers, called a pair of *wire labels*, rather than a single 1-bit value 0/1 as in a regular gate.

Taking the 2-bit input, 1-bit output garbled gate, denoted by $c=\mathsf{G}(a,b)$, as an example, its wire labels are denoted by $\{k_a^0,k_a^1\}$,$\{k_b^0,k_b^1\}$, $\{k_c^0,k_c^1\}$, where each entry, say k_a^0, is a unique random number assigned to the value 0 at the "a" input of c=G(a,b). In the secure computing process of G, the value of its 1-bit input "*wire*" a (b) is provided by the generator (the evaluator), and the generator and the evaluator jointly compute the wire labels of c.

To compute c=G(a,b) jointly, a should receive the generator's 1-bit value x, and b the evaluator's 1-bit value y. Here x and y can be either 0 or 1, and we denoted y' as the negation of y. The generator first generates $\{k_a^0,k_a^1\},\{k_b^0,k_b^1\}, \{k_c^0,k_c^1\}$ for possible values {0, 1} of wires a, b, and c. The encrypted *wire label*s for the evaluator's inputs, $\{k_b^y, k_b^{y'}\}$, are transferred from the generator to the evaluator via *Oblivious Transfer* (OT) [82]. In the end of OT, the evaluator only knows k_b^y and the generator only knows that one of $\{k_b^y, k_b^{y'}\}$ has been chosen by the evaluator. The generator also sends the wire label k_a^x to the evaluator.

If a garbled gate G(a,b) accepts one direct 1-bit input from the evaluator, one OT transaction is needed in SFE. Therefore, computing a single SFE instance may need multiple OT transactions. Based on a random oracle model proposed in [48], a virtually unlimited number of OT computations can be encoded into 80 1-out-of-2 OT transactions, where 80 is a security parameter. Huang *et al.* [43] reported that such 80 1-out-of-2 OT transactions can be computed in 0.6s. Therefore, the primary computing bottleneck is caused by the block-cipher operations for the vast number of garbled gates, not by the OT. As such, the discussion of OT is not included in this dissertation.

The computing logic of c=G(a,b) is a four-entry truth table T{T_{00}, T_{01}, T_{10}, T_{11}}, where each entry, say T_{00}, is a value 0/1 and is mapped the output wire label k_c^0 / k_c^1.

The garbled truth table is a random permutation of the four cipher-texts: $\{\mathsf{E}k_a^0(\mathsf{E}k_b^0(k_c^{T00})),\ \mathsf{E}k_a^0(\mathsf{E}k_b^1(k_c^{T01})),\ \mathsf{E}k_a^1(\mathsf{E}k_b^0(k_c^{T10})),\ \mathsf{E}k_a^1(\mathsf{E}k_b^1(k_c^{T11}))\}$, here E denotes the encryptor (also known as garbler), and the encryption/decryption operation is also known as *garbling*/*de-garbling*. In the end, with k_a^x and k_b^y, the evaluator learns the wire label k_c^{Txy} that represents the real value of G given inputs a=x and b=y, while it does not know x.

```
program And { // SFDL
        const N=2;
        type Len = Int<N>;
        type AliceInput = Len;
        type BobInput = Len;
        type AliceOutput = Len;
        type Input = struct {AliceInput alice,  BobInput bob};
        type Output = struct {AliceOutput alice};

        function Output output(Input input) {
           output.alice = (input.bob & input.alice);
        }
}
```

```
SHDL (compiled from SFDL):
0 input         //output$input.bob$0
1 input         //output$input.bob$1
2 input         //output$input.alice$0
3 input         //output$input.alice$1
4 output gate arity 2 table [ 0 0 0 1 ] inputs [ 2 0 ] //
output$output.alice$0
5 output gate arity 2 table [ 0 0 0 1 ] inputs [ 3 1 ] //
output$output.alice$1
```

Figure 2.1 An Example of SFDL and SHDL

In Fairplay [65], a GC and the ownership of its I/O data can be specified in the Secure Function Description Language (SFDL) and the Secure Hardware Description Language (SHDL). A simple example of SFDL and SHDL is illustrated in Figure 2.1, which describes the logic "(a AND b)", where a and b are 2-bit inputs belonging to the generator (Alice) and the evaluator (Bob), respectively. Gate 4 and gate 5 in the SHDL code are examples of G; and "[0 0 0 1]" in the description of Gate 4 is an instance of truth table T{T_{00}, T_{01}, T_{10}, T_{11}}. The "output" descriptors in the front of Gate 4 and Gate 5 specify that the output wires of these two gates are the output of this GC. Computing (*a* AND *b*) is equivalent to separately computing Gate 4 and Gate 5 and exporting their de-garbled results as the de-garbled results of this GC.

2.2 Private Set Intersection and Secret Handshake

Private Set Intersection (PSI) and *Secret Handshake* (SH) are two widely used privacy-preserving computing protocols, where PSI is used by two principals to compare the common element(s) in their two input sets without giving out the plain-text of all distinct elements to each other, and SH is used by two principals to mutually authenticate each other based on a one-time registered group secret. In an ECC-based implementation of PSI, point multiplication operations are executed O(*k*) times per PSI instance, where a malicious adversary has 1/*k* possibility to guess the correct result using a brute force method. Besides ECC and SFE, PSI can also be implemented based oblivious pseudo-random function evaluation [39][51].

SH was first proposed in Balfanz *et al.* [9]. In SH, if the two principals know a common group secret, their SH session will succeed and they know they are in the same group. Otherwise, the group secret of each principal is unrevealed to the other. The two critical properties of SH are *un-linkable* and *re-usable*. “Un-linkable” indicates that when one player A gives two (different) copies of his encrypted group secret to two other players B and C in two separated authentication sessions, B and C do not know they are authenticating with the same person A when they compare the encrypted data received from A. “Re-usable” means A can re-use its group secret in multiple SH sessions. Ateniese *et al.* [6] and Jarecki *et al.* [50] implemented un-linkable SH schemes with re-usable group secrets involving a third-party CA. Duan *et al.* [26] proposed a bilinear-pairing-based SH that satisfied both un-linkable and re-usable properties without using CA. In their protocol, to complete a single SH instance, each principal needs to run one bilinear pairing operation.

2.3 Elliptic Curve Cryptographic Algorithms

Elliptic Curve Cryptography (ECC) was proposed by Koblitz *et al.* [56] and Miller *et al.* [67]. They independently suggested the use of elliptic curve groups in public key cryptography. Comparing with RSA [84] or *Diffie-Hellman* (DH) [25] of the same secure strength, ECC needs a much shorter key size. For example, 256-bit key size in ECC has the same secure strength as 3072-bit key size in RSA [38] (equivalent to 128-bit AES secure strength).

Let K be a finite field and $E(K)$ an additive group of points on an elliptic curve E over K, $E(K)$ is defined as the set of points (x, y), $\forall x, y \in K$, a point (x, y) satisfies the *Weierstrass* equation:

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \qquad (2\text{-}1)$$

where $a_1, a_2, a_3, a_4, a_6 \in K$. $E(K)$ also includes an extra point O, called *point at infinity*. The number of points on $E(K)$ is called the *order of $E(K)$* and is denoted by $\#E$. In this dissertation, q is chosen as a large prime number, $K = F_q$ is a finite prime field, and the Weierstrass equation (2-1) is selected as:

$$y^2 = x^3 + ax + b \qquad \text{, where } a, b \in F_q \qquad (2\text{-}2)$$

Point Multiplication is the multiplication between a multi-precision integer k and a point P on $E(F_q)$, in a format of $[k]\ P$. The result of $[k]\ P$ is also on $E(F_q)$. Computing $[k]\ P$ is an accumulation process of *point addition* of k number of point P. Let $P(x_1, y_1)$, $Q(x_2, y_2)$ be two points on $E(F_q)$, the point addition $R'(x_3, y_3) = P(x_1, y_1) + Q(x_2, y_2)$ is computed as follows:

$$x_3 = \lambda^2 - x_1 - x_2,\ y_3 = \lambda\,(x_1 - x_3) - y_1 \quad (2\text{-}3)$$

$$\lambda = (y_2 - y_1)/(x_2 - x_1) \text{ if } P \neq Q, \text{ OR } \lambda = 3(x_1^2 + a)/2y_1 \text{ if } P = Q \quad (2\text{-}4)$$

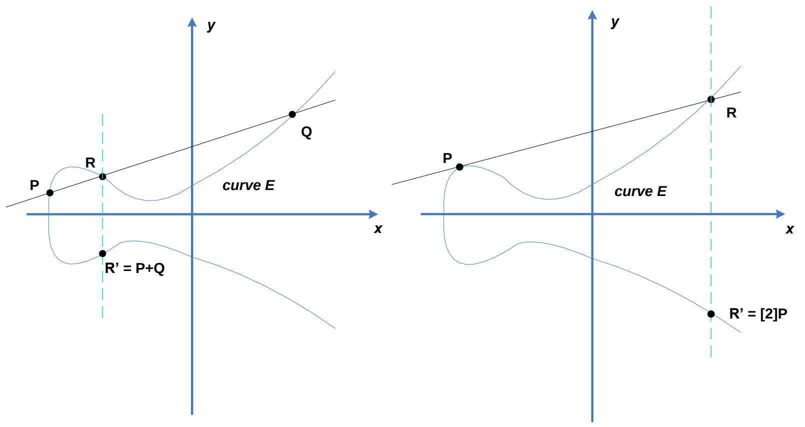


Figure 2.2 (a) Point Addition, (b) Point Doubling on $E(F_q)$

The effect of (2-3) and (2-4) in the affine co-ordinate system is illustrated in Figure **2.2**: the straight line joining P and Q intersects $E(F_q)$ at another point R, the point R' = P+Q is obtained by negating the y-axis co-ordinate of R. A special case of point addition $R' = P + P$ occurs when $P = Q$, which is also illustrated in (2-4) and Figure **2.2**. When $P = Q$, by taking the tangent to $E(F_q)$ at P, there is one tangent line that must intersect $E(F_q)$ at a point R, then the point $R' = P + P = [2]\ P$ is obtained by negating the y-axis co-ordinate. A *point subtraction* $R' = P - Q$ is equivalent to a point addition after negating subtrahend Q's Y-coordinate y_2, and the point subtraction has the property: $P = (P+Q) - Q$.

The basic *bilinear pairing* algorithm $e(P, Q)$ is a bilinear mapping: $G_1 \times G_2 \rightarrow G_T$, where $G_1=G_2=G_T=E(F_q^k)$, k is an integer and F_q^k an extension field of F_q. Its bilinear property, which is represented as $e([a]P, [b]Q) = e(P,Q)^{ab} = e([a]P, Q)^b = e(P, [b]Q)^a$, where a and $b \in F_q$, is usually used to construct privacy-preserving protocols. One of the most popular *ordinary* [32] curve families used to implement bilinear pairing is the BN curve family, which has the form of $y^2 = x^3 + b$ defined over F_q. Given the construction parameter u, a BN curve is constructed as follows: the trace of Frobenius over F_q is $t(u)= 6u^2+1$, the modulus $q(u) = 36u^4+36u^3+24u^2+6u+1$, the order $n(u) = 36u^4+36u^3+18u^2+6u+1$. When u reaches 64-bit and the *embedding degree* k=12, the security strength of the bilinear pairing computation is equivalent to 128-bit AES [10].

```
Input: $P$ in $E(F_q)[r]$, $Q$ in $E(F_q^k)[r] \cap Ker(\pi_q-[q])$,
 and $a=6u+2=\sum_{i=0} a_i 2^i$, $a$ has $L$ effective bits
Output: $Ra(Q, P)$
1:  $T \leftarrow Q$, $f \leftarrow 1$;
2:  for $i = L-2$ to 0, step is 1, do:
3:     $T \leftarrow 2T$;
4:     $f \leftarrow f^2 \cdot l_{T,T}(P)$
5:     if $a_i==1$:
6:         $T \leftarrow T+Q$;
7:         $f \leftarrow f \cdot l_{T,Q}(P)$
8:     end if
9:  end for
10: return $f \leftarrow (f \cdot (f \cdot l_{aQ,Q}(P))^q \cdot l_{\pi(aQ+Q),aQ}(P))^{(q \wedge k-1)/r}$
```

Figure 2.3 Miller's Algorithm for *R-ate* Pairing

The implementation of bilinear pairing is called *Miller's algorithm* [87]. It has several optimized variants, based on different families of elliptic curves. The variant adopted in this dissertation is *R-ate* pairing. Miller's algorithm for *R-ate* pairing is displayed in Figure 2.3. In Figure 2.3, F_q is a finite field with modulus q, k the embedding degree, E/K an elliptic curve E over a field K, $E[r]$ the group of all r-torsion points of E, and $E(K)[r]$ the K-rational group of r-torsion points of E over field K. Let π_q be the q-power Frobenius endomorphism on $E(F_q)$, $\mathrm{G}_1 = E(F_q)[r]$, $\mathrm{G}_2 = E(F_q)[r] \cap Ker(\pi_q-[q])$, t the trace of π_q, $P \in \mathrm{G}_1$ and $Q \in \mathrm{G}_2$, the function of *R-ate* pairing [60] over BN curves [10] is $Ra(Q, P) = (f \cdot (f \cdot l_{aQ,Q}(P))^q \cdot l_{\pi(aQ+Q),\mathrm{aQ}}(P))^{(q^\wedge k\text{-}1)/r}$, where $a = 6u+2$, u is the BN curve construction parameter, $f = f_{a,Q}(P)$ the rational function and $l_{\mathrm{A,B}}$ the line function [87] through point A and B. The algorithm is commonly divided into two major steps: lines 2-9, called *Miller Loop* (ML), and line 10, known as *Final Exponential* (FE). In FE, The exponentiation by $(q^k\text{-}1)/r$ promises a unique result.

2.4 Modern GPU Architecture

The experimental devices used in our work are based on the *Compute Unified Device Architecture* (CUDA) [36]. GTX-680 is a GK104 generation device [30][36], which contains 8 *streaming multiprocessors* (SMX). Each SMX can concurrently run multiple GPU threads. These threads are grouped into 32 parallel threads, called *warp*s. Each SMX has 192 CUDA cores along with 32 load/store units, which allow for a total 32 threads per clock to be processed. However, for better utilization of the pipeline, it usually simultaneously runs multiple warps of threads in one SMX. Warps of threads assigned to the same SMX and dispatched to run at the same time are called a *block* of GPU threads. Each SMX also contains four *warp scheduler*s with eight *dispatch unit*s that process 64 concurrent threads (2 warps) to the cores. For the fast data storage, each SMX has 64K 32-bit registers and 64KB on-chip *shared memory*/L1 cache. The shared memory resides on 32 64-bit banks. The on-chip fast shared memory is usually used as cache or shared variables among threads in the same block. Two SMXs share one global *memory controller*, and each memory controller ties with a 128KB L2 cache. In total, there are four memory controllers and 512 KB L2 cache. Via these memory controllers, SMXs could access the 2GB slow *global memory*. The global memory clock is 1502MHz, and the global memory bandwidth is 256-bit.

A program on GPU is called a *kernel function*. Its input setup, parallelism configuration, launching and output read-back are controlled by a host thread on CPU. Once a kernel function is launched, its host thread could release the CPU time-slice and be waken until the kernel function completes. At runtime, following the *Single Instruction Multi Threads* (SIMT) architecture, each GPU thread runs one instance of the kernel function. The degree of parallelism is determined by (1) the register usage per thread, (2) the shared memory usage per thread. To fully utilize the computational strength of the Kepler GPU, the degree of parallelism needs to be raised as much as possible to cover the memory accessing latency and other overheads in the pipeline of CUDA cores. Any shared memory access bank conflicts, code path divergence and explicit synchronization command will stall the concurrent execution of a warp and thus should be avoided. Succinctly put, key design objectives include maximizing the degree of parallelism, minimizing buffer usage, four-cycle-cost synchronization across threads in the same block, branch divergence, and shared memory bank conflicts.

CHAPTER III

RELATED WORK

In this chapter, we first discuss previous work on optimization techniques for SFE, point multiplication and bilinear pairing. Then, we report existing CPU-based benchmarks of the SFE-based ED problem and the SFE-based SW problem, and CPU-based or GPU-based benchmarks of point multiplication, and bilinear pairing.

3.1 Secure Function Evaluation

Besides the Fairplay [65] introduced in the previous chapter, TASTY [40] was another work that studied the composibility of privacy-preserving applications. TASTY provided a programming language to construct privacy-preserving applications via SFE-based GCs and HE-based arithmetic functions. To optimize the computation of SFE, several techniques had been proposed: *free-XOR* [57] which replaced block-cipher operations by XOR operations for the XOR gate, "*permute-and-encrypt*" [65] which reduced the de-garbling process of a garbled gate to one block-cipher operation, the *m*-to-*n* garbled lookup table and the *compact-circuit* design [43] which reduced the number of garbled gates for a number of SFE-based problems, and the *garbled-row-reduction* (GRR) [79] which reduced 25% space of the garbled result for each gate.

Jha *et al.* [52] proposed three protocols for the SFE-based ED problem and the SFE-based SW problem. Their *protocol-3* solved a 200×200, 8-bit alphabet ED (60×60 SW) problem in 658 (1000) seconds. Later, by using all the optimization techniques mentioned in the previous paragraph, Huang *et al*. [43] computed a 2000×10000 8-bit alphabet SFE-based ED problem in 223 minutes, and a 60×60 SFE-based SW problem in 415 seconds, both on two computes with Intel Duo E8400 3GHz CPUs. For CPU-based benchmarks of SFE-based problem in the malicious model, we refer readers to [46][58].

Frederiksen *et al*. [30] parallelized the OT transactions and multiple instances of one garbled circuit on GPU, in the malicious security model. But our work was different from theirs. Their privacy-preserving applications were limited to one GC, which was insufficient to describe the SFE-based ED problem or the SFE-based SW problem. Their parallelization strategy is launching thousands of independent GPU threads while each GPU thread runs an independent SFE instance. Comparing with their work, our work focuses on parallelization of a large scale SFE instance on GPU. And our parallelization strategy partitions independent blocks of GCs of the single SFE instance and run these independent blocks simultaneously. Due to the difference on parallelization strategies, our implementation requires more complicated synchronization mechanisms that theirs. CUDASW++ [62] was one of the most famous open source projects that ran the plain-text SW problem on GPU. In CUDASW++, the storage of DNA query scores was re-organized to minimize overheads for memory access. Because real values of scores are replaced by paired wire labels in SFE, we do not follow their storage scheme. Instead, we develop our collision-free storage model for wire labels and eliminate the memory interface competition. And to minimize memory heap operations, we develop a *static* memory management scheme to maintain wire labels and intermediate results of the garbling/de-garbling process. A slice-by-slice GPU resource mapping policy was proposed in CUDASW++. As we observe that the privacy-preserving requirement of SFE introduces new inter-dependency for computation steps on the generator side, we develop a new GPU resource mapping policy for the generator. And for the computations on the evaluator side, since the privacy-preserving requirement of SFE only complicates the computation steps, we develop a more fine-grained GPU resource mapping policy than that in CUDASW++.

3.2 Point Multiplication and Bilinear Pairing

Elliptic curves over finite fields can be divided into two types: the *supersingular* (SS) curves, and the ordinary (non-supersingular) curves. Let the *trace* of E/F_q be $t = q+1-\#E(F_q)$. If the greatest common divisor of t and q equals 1, then E is ordinary, otherwise E is SS. For their simplicity and ease of modular multiplication, SS curves have been proposed to construct pairing-based cryptographic protocols. But SS curves have limitation on the potential values of the embedding degree k, and it requires to use curves of characteristic 3 when its embedding degree $k = 6$ [1][24]. As proved in [54], implementing characteristic 3-based arithmetic operations on GPU brings either more space cost, or more complicated logic and thus harder for parallelization. Therefore, we prefer to use ordinary curves and compute the pairing algorithm under characteristic 2.

One ordinary curve can be claimed as a *pairing-friendly* curve when two conditions are satisfied: a prime $r \geq \sqrt{q}$ dividing $\#E(F_q)$, and k with respect to r is less than $log2(r) / 8$ [32]. One of the most popular pairing-friendly ordinary curve families is the *Barreto-Naehrig* (BN) [10] curves, in which the curve parameters are represented by polynomials of the construction parameter *u*. Recently, Pereira *et al.* [78] proposed an implementation-friendly subclass of BN curves that brought better computational efficiency.

A major approach to accelerate Miller's algorithm is by reducing the length of Miller Loop. Barreto *et al.* [11] extended the Duursma-Lee method [27] to supersingular abelian varieties using the η_T pairing. The *Ate* pairing on hyper-elliptic curves [34], the twisted *Ate* pairing [42][66] and its variation *Ate* pairing [99] on ordinary curve reduced the loop length to $log2(|t - 1|) / 8$, where t was the trace. An *optimal Ate* pairing [94] was able to attain the iterations of Miller Loop to its lower bound. The *R-ate* pairing [60] obtained even shorter loop length than [34] on certain pairing-friendly elliptic curves. Other efforts [4][16][86][87][88] worked on arithmetic optimization, such as denominator elimination, final exponentiation simplification, and faster variants of Miller's algorithm under the Jacobian [16] or the Edwards co-ordinate [13], efficient formulas for various curves with twists of degree 2,3,4 or 6 [23].

Antão *et al.* [2], Bernstein *et al.* [12] and Szerwinski *et al.* [93] pioneered the implementation of point multiplication on CUDA. Szerwinski *et al.* [93] straight forward ported multi-precision arithmetic on GPU. Bernstein *et al.* [12] represented a 224-bit multi-precision operand into 24 32-bit float point numbers and achieved 5895 point multiplication per second on one GTX-285 GPU. Antão *et al.* [2] implemented point multiplication under *Residue Number System* (RNS) and achieved 9827 point multiplication per second on one GTX-295 at the 112-bit security level. However, Antão *et al.* [2] did not have a comparison of computational complexity between *conventional Montgomery* and *RNS-based Montgomery* number system on GPU, and directly adopted RNS-based Montgomery number system. Longa *et al.* [63] and MIRACL [69] reported two of the best CPU-based benchmarks of point multiplication. The former was on a standard elliptic curve and the latter on a *twisted curve* over F_q^2 [63]. MIRACL [69] could run 14509 point multiplication per second on 3.0GHz AMD Ph. II X4 CPU at the 112-bit security level, and Longa *et al.* [63] 22472 point multiplication per second on 2.6GHz AMD Opteron at the 128-bit security level. Another recent work [19] also adopted RNS-based Montgomery as the number system to implement bilinear pairing on FGPA.

Two recent studies worked on GPU-based bilinear pairing solutions: Zhang *et al.* [98] implemented *Tate* pairing over a base field with a composite order, and Katoh *et al.* [54] η_T pairing in characteristic 3. Both papers serially implemented the *reduction* function [68] due to its difficulty for parallelization. As we will discuss later, a parallel version and a serial version of the reduction function are developed in our work, and we compare their performance to evaluate which solution is more efficient on the contemporary GPU architecture. Moreover, both papers ignored the parallelization of the final exponentiation (FE) step, which was almost as the same cost as Miller Loop (ML) (check Table 2 in [14] for the computing cost comparison between ML and FE). We aim to support parallelized ML, FE. Additionally, we also aim to support exponentiation over F_q^{12} for privacy-preserving protocols that may further run exponential steps on the result of bilinear pairing.

CHAPTER IV

LARGE SCALE PRIVACY-PRESERVING ED/SW PROBLEMS ON GPU

In this chapter, we present the parallel computing model to compute the large-scale privacy-preserving Edit-Distance and Smith-Waterman problems on the contemporary GPU architecture, with state-of-the-art optimization techniques such as *free-XOR* [57], *oblivious transfer extension* [48], *permute-and-encrypt* [65], *efficient lookup-table design* and *compact circuit*s [44]. We will first discuss the inter-dependency among computation steps of the SFE-based *Edit-Distance* (ED) problem and the SFE-based *Smith-Waterman* (SW) problems, on both the generator and the evaluator sides. Then, we present the low-level GPU gate garbler/de-garbler. Next, we discuss details of the pipelined computation steps for the SFE-based ED problem and the SFE-based SW problem, respectively. In the end, to verify the efficiency of the system proposed in this dissertation, we evaluate the execution result of a 5000×5000 8-bit character SFE-based ED instance and a 60×60 SFE-based SW instance.

Recalling the garbling/de-garbling process of a garble gate c=G(a,b) introduced in the subchapter 2.1, the generator generates wire labels $\{k_a^0, k_a^1\}$, $\{k_b^0, k_b^1\}$, $\{k_c^0, k_c^1\}$ for *a, b*, and *c,* runs four block-cipher operations as the garbling process of G, sends the digests of block-cipher operations to the evaluator, and then the evaluator runs the same number of block-cipher operations as the de-garbling process of G to decrypt a wire label of *c*. In this process, it is the evaluator who really computes the result of gate G. Supposing another gate e=G'=(c,d) exists and it re-uses G's output *c*, for the generator, it can simultaneously garbling G and G' if it has generated all wire labels $\{k_a^0, k_a^1\}$, $\{k_b^0, k_b^1\}$, $\{k_c^0, k_c^1\}$, $\{k_d^0, k_d^1\}$ and $\{k_e^0, k_e^1\}$. However, for the evaluator, it needs the de-garbling result of G to de-garbling G', and thus de-garbling G' must follow the de-garbling process of G.

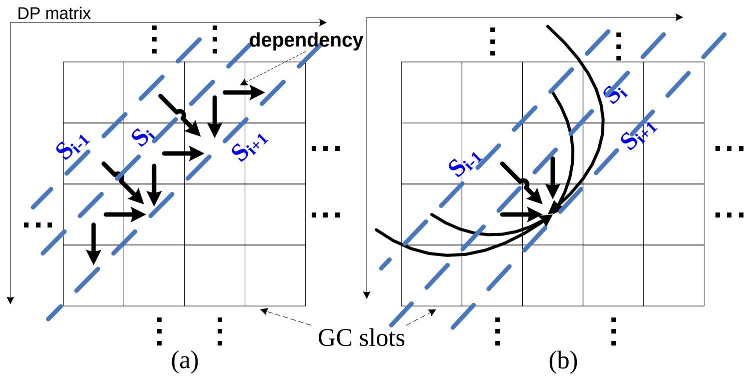


Figure 4.1 Parallel Computing Models for the (a) ED, (b) SW problems

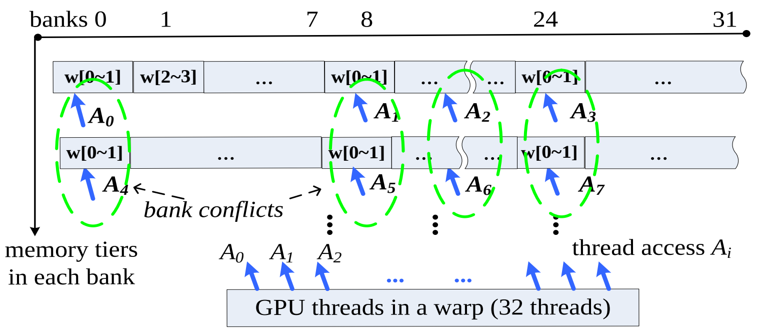
Before we discuss inter-dependency of the SFE-based ED problem and the SFE-based SW problem on the generator side and the evaluator side, we first present our divide-and-conquer design strategy for the parallel computing model. Our design strategy has two levels: the *GC level* at the bottom, and the *DP level* on the top. At the GC level, the vast number of gates are concurrently garbled on the generator side, or de-garbled on the evaluator side. At this level, we focus on maximizing the degree of parallelism for the garbling/de-garbling process with minimum idle cycles on the GPU. Satisfying the *ultra-short* security level [40] is also considered at this level. At the DP level, we focus on fully utilizing the degree of parallelism provided by the GC level, while the inter-dependency described in the previous paragraph is satisfied.

To understand the inter-dependency of the SFE-based ED problem and the SFE-based SW problem, we first analyze the inter-dependency in their plain-text counter parts. As we will show later, computing one entry of the DP matrix in the plain-text ED (SW) problem is dependent to the results of three neighbor entries on its top, left and top-left (the neighbor entries on the same column and the same row, and the neighbor entry on its top-left). The inter-dependency in the plain-text ED and SW problem are illustrated in Figure 4.1, respectively. Parallelization of the ED problem and the SW problem fits the "wavefront" pattern [62], which is proposed for tree computation where child nodes depend to their parents. The term *wavefront* describes the edge separating the executed nodes from nodes waiting for execution in the next round. In Figure 4.1, the $N \times N$ DP matrix is processed into $2N$-1 *slices*, W= $\{S_1, S_2, \ldots, S_{2N-1}\}$ and a slice S_i is a diagonal from the top right to the bottom left. And for entries of the DP matrix, called *slots*, they are independent to each other if they are on the same slice. When applying the *wavefront* pattern to the SFE-based ED and SW problems, we treat entries of the DP matrix, called *GC-slots*, as the atomic module at the DP level. Then, for the evaluator, *GC-slots* are independent when they are on the same slice. The degree of parallelism equals to the length of a slice, which increases from S_1, S_2 until S_N, and then decreases from S_N to S_{2N-1}. For de-garbling GC-slots on S_i, the pre-requisite is the de-garbled outputs on slices S_{i-2} and $S_{i-1.}$ And hence, the de-garbling process can only de-garble one slice at a time, which means S_i is mapped to GPU units after S_{i-1} is completed.

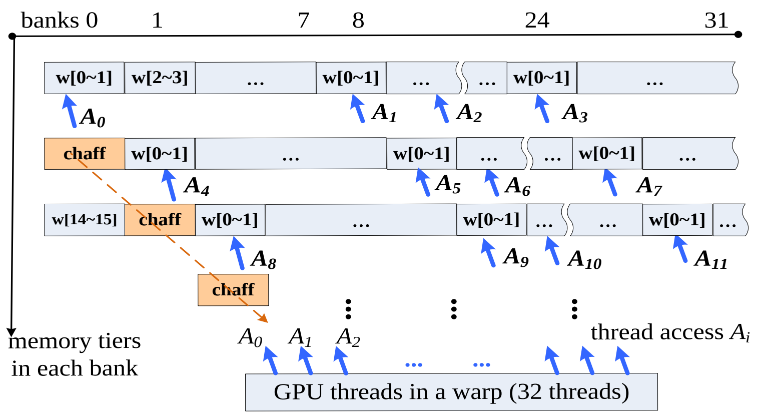
However, for the generator, according to our discussion in the previous two paragraphs, if wire labels of all GC-slots' outputs have been generated, multiple slices can be garbled simultaneously. As such, garbling the $N \times N$ matrix is transformed to a 1-D vector which is mapped to GPU units. Later in this chapter, observations in this subchapter are implemented as *cross-slice mapping policies* (CSMP) for the generator and *slice-by-slice* policies for the evaluator. In the rest of this chapter, we first present the GPU-based gate garbler, which is the implementation at the GC-level. Then we discuss the DP-level implementation for the ED problem and the SW problem, respectively.

4.1 GC-level: GPU-based Gate (De-) Garbler

Recalling the garbling/de-garbling process of a gate, an arbitrary truth table entry T_{xy} of a gate G is garbled as $Enc_{x,y}$ $(k_c^z) = \mathbf{H}(k_a^x \| k_b^y)$ XOR k_c^z, where H is the encryption function, k_a^x, k_b^y and k_c^z are wire labels, "||" is concatenation. Following Huang *et al.* [43], 80-bit wire label is adopted to meet the ultra-short security proposed in TASTY [40]. Choices of **H** are SHA-1 [43], SHA-256 [37], AES-256 supported by the AES-NI instruction set of Intel CPUs [58], or other cryptographic hash functions. In this work, we chose SHA-256 as H due to its similar cost of SHA-1 [22][80], and better secure strength than SHA-1. AES-256 is excluded because it is 3 times slower than SHA-1 on GPU [49]. As a result, $Enc_{x,y}$ (k_c^z) is in the format of SHA-256($k_a^x \| k_b^y \| i$) XOR k_c^z, where i is a 32-bit unique gate index in a garbled circuit, where $(k_a^x \| k_b^y \| i)$ is a 192-bit block, and the output of $Enc_{x,y}$ (k_c^z) a 256-bit digest. Similarly, the de-garbling function Dec($Enc_{x,y}$ (k_c^z)) is SHA-256($k_a^e \| k_b^e \| i$) XOR $Enc_{x,y}$ (k_c^z), where k_a^e, k_b^e are wire labels obtained from OT or a de-garbling process for a predecessor gate.



(a) shared memory bank conflict in the naïve placement of W[0~15]



(b). Collision avoidance shared memory access model

Figure 4.2 Optimization on Shared Memory Access

The SHA-256 code base used in this research is PolarSSL [81]. When porting the code base to GPU, following adjustments are made: our initial analysis shows that one round of SHA-256 can be further divided into four steps, and each of which produces 16 (32-bit) words *W*[0~15] based on the elements in *W* computed in the current and previous steps. Furthermore, in the end of each step, *W*[0~15] are used to update the eight 32-bit digest (cipher text). In the original code base, the four steps are computed together and thus it needs a four times larger variable *W'*[0~63]. We clearly partition the four steps and keep re-using *W*[0~15] in each step. As a result, the share memory usage per SHA256 instance is dropped to ¼ of the original version.

Overall, a total of 40 (32-bit) words of space are needed for each round. That is, when each thread runs an independent instance of SHA-256, 16 (32-bit) registers store *W*[0~15] in the current step, 8 (32-bit) registers store the digest, and one block of 16×32-bit shared memory is assigned to each thread to store *W*[0~15] produced in the past step, the storage format of such a shared memory block per thread is illustrated in Figure 4.2 (b). Here, each block (**uint32** *W*[0~15]) resides on eight shared memory bank.

It is noted that a strip of chaff spacer is inserted in the final version of memory access model. Figure 4.2 (a) illustrates the memory access pattern if the *W*[0~15] is placed into the shared memory without any optimization. In Figure 4.2 (a), it is found that, as all 32 threads in the same warp reads their own *W* with an identical offset, thread {*i*, *i*+4, *i*+8, *i*+12, *i*+16, *i*+20, *i*+24, *i*+28}(*i*=0,1,2,3) are trying to access different *tiers* (a low level GPU architecture) of the same memory bank simultaneously. When such a situation happens, GPU threads in a same warp will be stalled. This case is known as bank conflict. If *W* is placed in the share memory following the policy in Figure 4.2 (a), the actual degree of parallelism drops to 1/8 of the configuration. To eliminate the often hidden shared memory access conflicts, a strip of 64-bit *chaff* spacers is filled, one in the front of every fourth-thread's *W*[0~15]. This way, parallel memory accesses {A_i, A_{i+4}, A_{i+8},…A_{i+28}} issued by threads *i, i+4, i+8, …i+28* (*i*=0,1,2, 3) to read *W*[0~15] of the same offset in its own *W* array will access distinct memory banks with no conflict. Figure 4.2 (b) displays an example of offset *j*=0, and {A_0, A_4, A_8… A_{28}} read *W*[0].

To reduce the unnecessary off-chip memory access for reading the 192-bit input block and writing the 256-bit digest, the *coalesce memory access* scheme [36] is applied as a minor optimization. Accessing off-chip memory usually brings a 1:100 performance degradation than on-chip memory. The bandwidth of the off-chip memory interface is 256-bit in the Kepler GPU architecture. Taking the input block as an example, the 192-bit input blocks from four threads, denoted as four **uint32**[6], are interleaved in the global memory space and occupies 3×256-bit. As such, as the four threads visit an arbitrary **uint32**[*i*] of their own 192-bit blocks with the same *i*, it costs only one global memory read. Although applying the *coalesce memory access* scheme needs extra computation cost for the storage format converting on the CPU side, our initial experiment results show that this scheme can bring 5% throughput enhancement for the gate garbler.

Overall, each SHA-256 gate garbler thread uses 57 registers, where GK104 allows up to 63 registers per thread. Global memory access only occurs when the gate garbler reads wire labels or writes the digest. Each SMX has 20 warps of GPU threads, and the degree of parallelism is 5120=8 SMX $\times$ 640 threads. Each SMX has 64KB on-chip memory, partitioned as 48 KB shared memory plus 16 KB L1 cache. 41.25KB of the 48KB of shared memory is utilized to save *W*. Any attempt to assign more complete warps of thread will make the total shared memory size exceed the shared memory size boundary of GK104, it can be concluded that the gate garbler has fully utilized the shared memory resource.

As a result, the latency of computing 10000 times SHA-256 on 5120 threads is 304ms, here each SHA-256 instance reads in a block of 192 bits as the input. The throughput is 30.27Gbps. This performance result has included the GPU-CPU data exchange time, and it is comparable to the result of SHA1 on GTX-580 [49]. As reference, Intel reported that their SHA256 could obtain 11.5 cycles/byte on a single core of Intel i7 2600 in 2012 [37], equivalent to 2.47Gbps. Next, we present our DP-level design for the SFE-based Edit-Distance and Smith-Waterman problems.

4.2 DP-level: Computing SFE-based Edit-Distance

To design the parallel computing model at the DP level, the first task is analyzing the computation logic of the ED problem. The plain-text version ED problem can be described as follows:

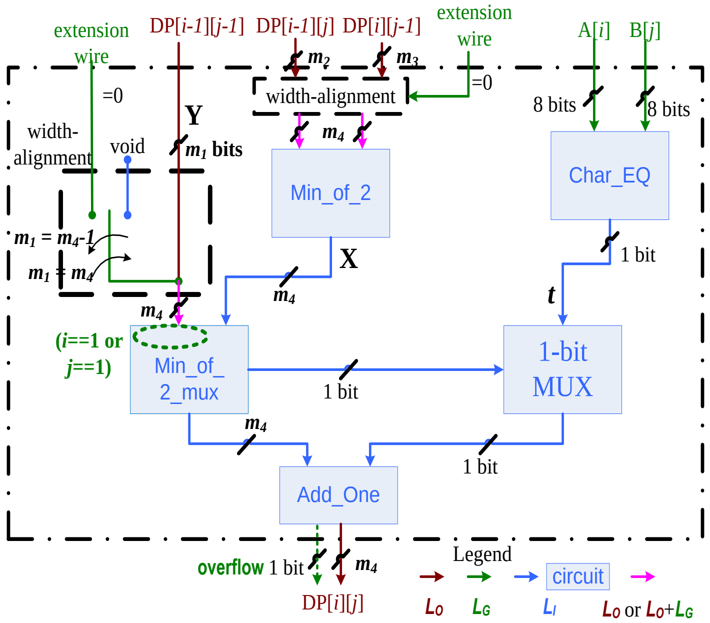


Figure 4.3 The SFE Building Block (a GC-slot at DP[*i*][*j*]) for ED

1. The two input strings A[*N*] and B[*N*] are from the generator and the evaluator respectively;

2. Solving the ED problem is essentially computing an $(N+1)\times(N+1)$ DP matrix from top-left to right-bottom. And each slot DP[i][j] ($i,j\epsilon[0,N]$) is computed as:

$$\text{DP}[i][0] = i,\ \text{DP}[0][j] = j, \text{ or}$$

$$\text{if } i,j\epsilon[1,N],\ \text{DP}[i][j] = (Y > X)\ ?\ (X+1) : (Y+t)$$

where $t = (A[i] \neq B[j])$, X=min(DP[i-1][j] , DP[i][j-1]), and Y=DP[i-1][j-1] [43].

When using the SHDL to describe the SFE-based ED problem, “DP[i][j] = (Y > X) ? (X+1) : (Y+t)” can be summarized as a combination of one “*equal*(A[i], B[j])” GC, two “*min*(x,y)” GCs, and one “*add*(x,1)” GC. Such a combination is illustrated in Figure 4.3. In Figure 4.3, the *GC-slot* represents the privacy-preserving computing logic for computing the entry DP[i][j], which is composed of two *Min_of_2* circuits (*Min_of_2* and *Min_of_2_mux*), one *Char_EQ* circuit (compute t), and one *Add_One* circuit.

[43] had already presented the optimal structure of one GC-slot, that is, GC-slots do not have a unified structure, instead, the complexity of a CG-slot is closely dependent to the actual bit-widths of inputs. However, some details for inter-connecting GCs within one GC-slot are not clear presented in their work. For example, two inputs of one *Min_of_2* GC are forced to have equal bit-width to ease the difficulty of GC design, but the *Min_of_2* GC’s inputs DP[i-1][j] and DP[i][j-1] may have different bit-width at certain slot$\{i, j\}$. In [43][44], alignment of widths of inputs for one GC was ignored.

We give the bit-width alignment scheme based on two 1-bit extension wires (see Figure 4.3) for {DP[i-1][j], DP[i][j-1]}, and {X, Y} here. Knowing that the maximum value of an arbitrary GC-slot DP[*i*][*j*] is max(*i*, *j*), the maximum possible values of inputs and intermediate results in a GC slot are listed in Table 4.1. This table is helpful to identify when the bit-width alignment scheme is necessary for the input wires of circuit *Min_of_2* and circuit *Min_of_2_mux*.

Table 4.1 Maximum Possible Values of Inputs and Intermediate Results in DP[*i*][*j*]

	DP[*i*][*j*-1] **(width = m_3)**	**DP[*i-1*][*j*]** **(width = m_2)**	***X* (*width* = m_4)**	***Y* (*width* = m_1)**
$i < j$	max(*i*,*j-1*) = *j-1*	max(*i-1*,*j*) = *j*	min(*i-1*,*j*) = *j-1*	max(*i-1*,*j-1*) = *j-1*
$i == j$	max(*i*,*j-1*) = *i*	max(*i-1*,*j*) = *j*	min(*i*,*j*) = *i*	max(*i-1*,*j-1*) = *i-1*
$i > j$	max(*i*,*j-1*) = *i*	max(*i-1*,*j*) = *i-1*	min(*i*,*i-1*) = *i-1*	max(*i-1*,*j-1*) = *i-1*

The first two columns of Table 4.1, representing DP[*i*-1][*j*] and DP[*i*][*j*-1], are values of inputs of one *Min_of_2* circuit. The difference of input value $m_3=m_2-1$($m_2=m_3-1$) is true when $i<j$ ($i>j$). As such, the extension wire is activated for DP[*i*][*j*-1] (DP[*i*-1][*j*]) when $i<j$ ($i>j$), and *j* (*i*) equals power of 2. Similarly, the 3rd and 4th columns, representing *X* and *Y*, are values of inputs of the other *Min_of_2* circuit which has one additional 1-bit “less or greater” signal output. The difference of input value $m_1= m_4-1$ is true when $i==j$. As such, the extension wire is activated for *Y* when $i==j$, and *i* is power of 2.

4.2.1 GPU Mapping Policies

Next, we present how thousands of GC-slots of the SFE-based ED problem are parallel computed on the Kepler GPU architecture. Recalling the key observation presented at the beginning of this chapter, for the generator, garbling a GC-slot on a slice S_i only needs to re-use wire labels associated output wires of GC-slots on predecessor slices S_{i-2} and S_{i-1}; And for the evaluator, de-garbling a GC-slot on a slice S_i needs to re-use decrypted results of GC-slots on predecessor slices S_{i-2} and S_{i-1}. Due to the difference of inter-dependency of computation steps for the two principals, their GPU mapping policies are designed separately.

On the generator side, the *cross slice mapping policy* (CSMP) is adopted as follows: the CSMP partitions the DP matrix into multiple *task*s, each of which aims to fill up 5120 GPU gate garbler threads to maximize the speedup factor. In one task, each GC-slot is assigned to one GPU thread. Before the current task starts, all paired wire labels for GC-slots in this task are prepared. The CSMP for the first task is illustrated in Figure 4.4.

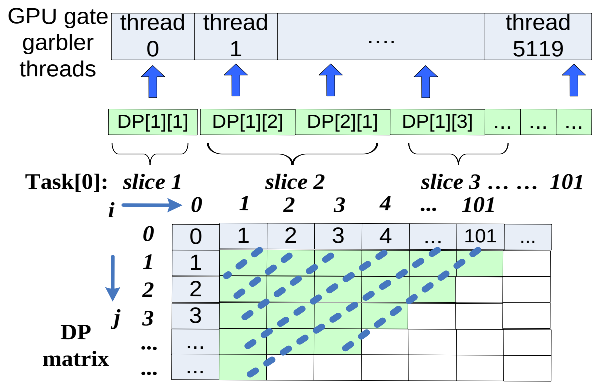


Figure 4.4 The Generator's Resource Mapping Policy: CSMP

Figure 4.4 shows that the first task *task*[0] contains 5120 GC-slots, which is from slice 1, 2, 3, … up to a fraction of slice 101. As such, *task*[0] has fully loaded the GPU gate garbler. Furthermore, to facilitate the synchronization between the two principals, the direction of counting the 5120 GC-slots is slice-by-slice, not from top-left to bottom-right. When GPU runs the 5120 GC slots in lock-step, each GPU thread garbles its corresponding GC-slot gate by gate for the entire GC-slot. In other words, the inter-dependency within one GC-slot is easily satisfied because the logic of each GC-slot is serially garbled by one GPU thread.

The challenge of CSMP is the management of wire labels. To support large problem sizes, generating wire labels for all GC-slots in the DP matrix at the beginning is inacceptable. Instead, it is preferred to generate wire labels at the beginning of each task. However, for certain wire labels, such a preference is impractical. For example, wire labels that represent output wires of GC-slots may be used in two consecutive tasks, and thus the successor task does not need to re-generate these wire labels. Another example is the wire labels that map to the input string A[*N*] and B[*N*], which are used by all tasks. As a result, it is necessary to differentiate types of wire labels according to their life-time.

Wire labels are classified into three major types L_O, L_I, and L_G. Referring to Figure 4.3, L_O represents the set of paired labels for wires of a GC-slot's outputs. L_I represents the set of paired labels for wires internal to a GC-slot and not connected to other GC-slots. L_G represents miscellaneous types of paired labels, and they are treated as a *"global"* set to simplify memory management. Classification of these three groups of wire labels is not only important to efficiently use of the GPU memory space, but also critical to synchronous accesses of wire labels by parallel GPU threads. L_O and L_I are pre-assigned at initialization of a new task, but L_G at initialization of the whole SFE system. L_O and L_I are overwritten if they are associated with an XOR gate's output [57] by a calculation result based on the XOR gate's inputs' wire labels. Even though some L_O and L_I need to be overwritten during execution, they are still pre-assigned to simplify the wire label generation function at negligible costs.

Overwriting of labels in L_O has to occur before garbling of a task. In most cases, wire labels in L_I are overwritten during garbling because no other GC-slots depend to them. However, if a wire label in L_O is dependent to some L_I, these wire labels in L_I need to be overwritten before overwriting of L_O. For instance, the output of an XOR gate G_1 is the input of another XOR gate G_2, and the output of G_2 is also the GC-slot's output. Here, wire labels associated with G_1's (G_2's) output are in $L_I(L_O)$. Both overwriting of labels for G_1 and overwriting of labels for G_2 should be done before garbling of a task, furthermore, the former overwriting needs to be done before the latter overwriting.

Next, miscellaneous cases related to L_G are listed: (1) the first case is the GC-slots on the edge of the DP-matrix (excluding the edge DP[*i*][0] and DP[0][*j*] since they are constant values). The two edges can be represented as DP[*1*][*j*], or DP[*i*][*1*], i.e., the second row and second column of the DP matrix. In these GC-slots, the *Min_of_2_mux* circuit's input DP[*i-1*][*j-1*] is a real value rather than wire labels from other GC-slots because *i* or *j*=1. Furthermore, some gates in these *Min_of_2_mux* circuits are only dependent to inputs wires of DP[*i-1*][*j-1*]. Garbling of these gates can be skipped, and these gates could be treated as the generator's inputs. As such, we directly assign paired wire labels to the outputs of these garbled gates. (2) the second case of L_G is in the *Add_one* circuit. As listed in Table 4.1, the maximum possible value of the input of the *Add_one* circuit equals *j-1* (*i-1*) if $i < j$ ($i > j$), and the maximum possible value of its output is *j* (*i*). When *j* (*i*) is power of 2, the bit-width of the output is 1-bit greater than that of the input. For this case, an overflow bit is needed for correctly representing the output value. (3) the third case is the extension wires aforementioned. (4) in the end, the forth case is wire labels mapped to the generator's input A[*N*] and the evaluator's input B[*N*]. They are global because they need to be used by multiple GC-slots, and they are generated in the system initialization phase.

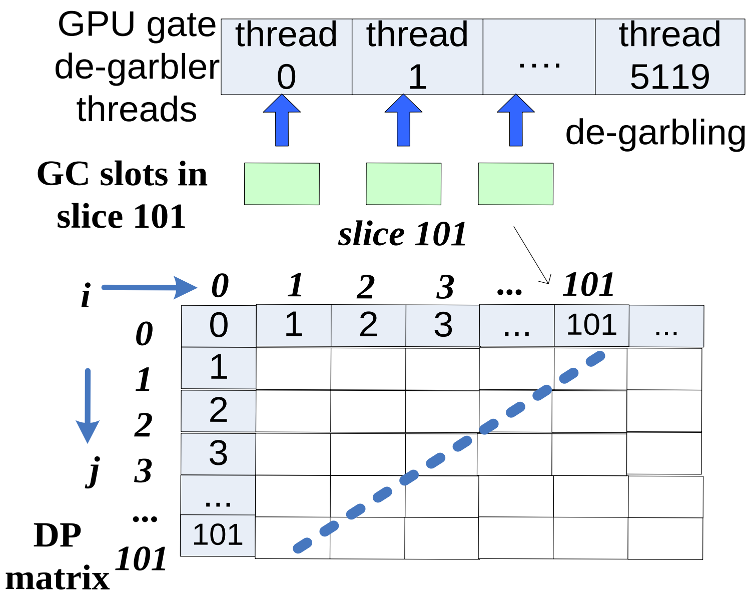


Figure 4.5 The Evaluator's GPU Resource Mapping Policy: *slice-by-slice*

On the evaluator side, a GC-slot DP[*i*][*j*] can only be de-garbled after the evaluator receives de-garbled results of slots DP[*i-1*][*j*], DP[*i*][*j-1*] and DP[*i-1*][*j-1*], which have been de-garbled in the previous two slices. As such, a *slice-by-slice* GPU resource mapping policy is proposed for the de-garbling process on the evaluator side. Figure 4.5 illustrates a snapshot of the de-garbling process of slice 101. In this figure, GC-slots in slice 101 are mapped to GPU de-garbler threads. This mapping happens after the generator side completes its tasks 0 and 1 since the generator's task 0 does not contain all GC-slots of slice 101. Same as the garbling process, each GPU thread de-garbles its corresponding GC-slot gate by gate, until all gates in its GC-slot is de-garbled. If a slice contains more than 5120 GC-slots, the mapping and de-garbling process is repeated until all GC-slots in the slice are de-garbled.

4.2.2 Memory Management and Pipelined Scheduling

While the GPU is garbling/de-garbling GC-slots, CPU is not idle. Instead, CPU are utilized to scheduling the execution of next task (slice) on the generator (evaluator) side, and maintaining memory chunks associated with next task (slice). To support the large-scale SFE-based ED problem and SW problem on commodity computers and GPUs, setting a moderate memory boundary (around 4GB on CPU, and 2GB on GPU) for our parallel computing model is necessary. To meet such a memory boundary, the *static memory management policy* is proposed as follows:

On the generator side, we observe that repetitive allocation and release of GPU memory for the L_I type wire labels are unnecessary because the host control thread can re-use one device memory chunk in multiple kernel functions. In other words, if the maximum device memory usage per task has been correctly estimated, then all tasks could repetitively utilize this memory block since they serially dispatched to the GPU device. The same memory management policy can be applied to the host memory, which only stores the encryption results of block-cipher operations as intermediate results for the generator. The memory allocation only happens in the system initialization phase, and the memory release operation only occurs in the system de-construction phase. As a result, the numbers of allocation and release operations in the host and GPU memory spaces are minimized.

Due to the slice-by-slice policy for de-garbling GC-slots on the evaluator side, the de-garbling result per slice does not need to be maintained in the GPU memory space until the whole execution ends. Instead, only de-garbling results of the latest three slices are kept in a separate memory chunk for the next slice. On the generator side, to simplify the synchronization between computation and network transferring, all computation results are copied to a separate memory block for network transferring. And then the memory space for storing computation results of the current task is ready for being re-used by the next task.

A scheduling step is a process on the CPU that sets up start-offsets of wire labels of L_I, L_O and L_G, and start-offsets of output results for each GC of each GC-slot in the current task (slice) on the generator (evaluator) side. To locate these start-offsets, it is necessary to collect information of GCs from their SHDL code. The objective of this parser is two-fold. First, it collects GC information, such as the number of L_I, L_O and L_G, the number of output entries and the dependency among gates within one GC. Second, by using collected information, it assesses the maximum memory usage of a task (slice) in the whole SFE-based protocol. Such a parsing process happens in the system initialization phase. And right now, this parsing process is hard coded in our system.

After this parsing process, there is another necessary preparation step in the initialization phase. This preparation step needs to generate static structural information for each GC-slot. It records which version of the compacted GCs is used in each GC-slot, how many wire labels of L_I, L_O and L_G are consumed in each GC-slot, and how many output entries will be generated by the block-cipher operations for each GC-slot. It also marks several flags for the utilization of wire labels L_G, for example, the overflow flag for the *Add_one* GC. With the structural information of GCs and GC-slots, computing the start-offset of wire labels of L_I, L_O, L_G, or entries of output results for each GC is simply accumulating offsets of that of the specific GC-slots, and the relative start-offsets of GC in the GC-slot.

These extension and overflow flags are critical to maintain the correctness of the logic, however, a general solution that uses conditional statement to check these flags for each GC and each gate during garbling/de-garbling may bring a large number of branch divergences on GPU. As such, this is a trade-off between the generality and performance of the system. Because this work focuses on the ED problem and the SW problem, our code is highly associated with the structure of GC-slots for the SFE-based ED problem and the SFE-based SW problem, so that a “if-else” statement for checking extension or overflow flags are triggered only when it is necessary.

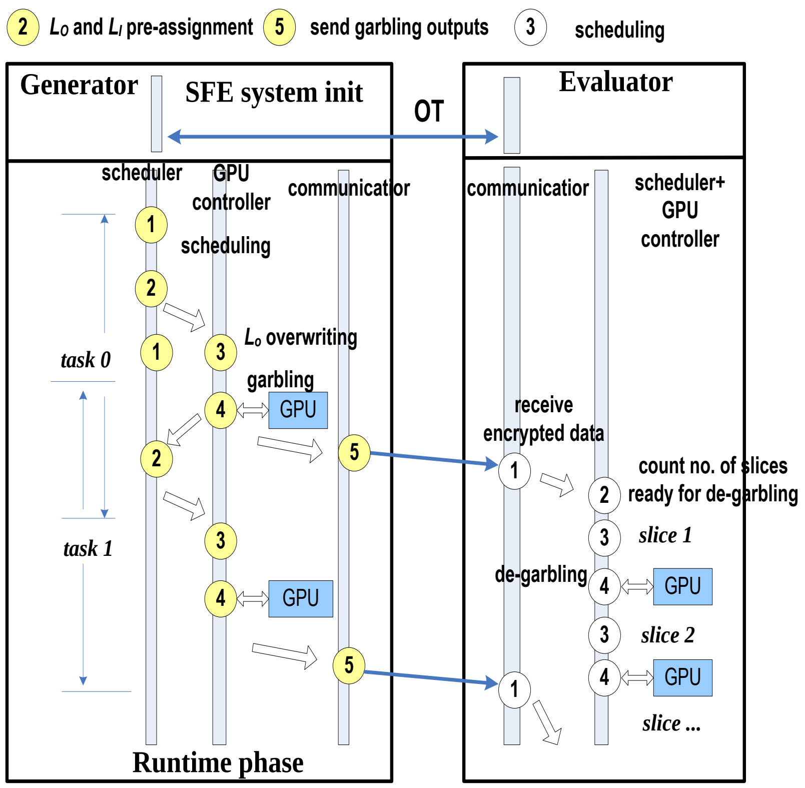


Figure 4.6 The Pipelined Garbling & De-garbling Process (ED)

Figure 4.6 illustrates the pipelined processing flows between the garbling and de-garbling processes for the ED problem. On the generator side, the three CPU threads are a *scheduler*, a *GPU controller* and a *communicator*. The generator's step 1 (the evaluator's step 3), named as "scheduling" are the scheduling step presented in last several paragraphs. When the scheduling work for the current task is completed, the scheduler pushes the current task to a queue shared with the GPU controller. Then the GPU controller revises slots' output wire label pairs for XOR gates as the step 3 (of the generator), and runs gate garbling on GPU. Meanwhile, the *scheduler* starts the scheduling step for the next task. Once the garbling process completes, the communicator transfers encrypted truth tables, encrypted permute-and-encrypt bits [65], and wire labels for extension wires (only one of a pair that associates with value 0) used in the current task, to the evaluator.

On the evaluator side, the two CPU threads are a *communicator* and a *scheduler+GPU controller*. Its *scheduler+GPU controller* first determines how many slices are ready for de-garbling after receiving the encryption data of the latest task. Figure 4.6 illustrates an example that the most recent received task, task 0, contains multiple slices. As such, the scheduling step and the GPU-based de-garbling step are invoked multiple times, each time for one slice in the task 0.

Table 4.2 Pipeline Execution Time Break Down (ED)

Exec Time	**Generator**	**Evaluator**
1. SFE System initialization	6.92s	2.94s
2. Scheduling	6.06s	23.04s
3. GPU garbling/de-garbling (without GPU-CPU data copy)	1062.95s (0.218 s/task)	136.55s (0.014s /slice)
4. CPU-GPU data copy, resource mgmt	99.13s	50.21s
5. Total computing latency	1520s	345.3s

4.2.3 Experiment Result and Analysis

Following the previous study [43], we select the test case 5000×5000 8-bit alphabet ED problem, which is composed of 1.88 billion non-free gates, as the test case to evaluate the performance of our parallel computing model on a CPU-GPU system. The generator runs 4883 tasks on a Xeon E5504 CPU at 2.00GHz with 16GB memory plus a GTX 680, and the evaluator runs 9999 slices on an Intel i7-3770K CPU at 3.5GHz with 8GB memory plus a GTX680 GPU.

Table 4.2 lists the break down of execution times for major pipeline steps of the test case. Row (2) matches the generator's step 1 (in Figure 4.6), and the evaluator's step 2+3. Row (3) matches the generator's step 2+3+4, and the evaluator's step 4. Row (5) lists the total computing latencies (1520s, 345.3s), which does not include networking transmission latencies, nor the system initialization time. There is a difference between the total computing latencies (row 5), and the sum of rows 2, 3, and 4. Such a difference is mainly spent in a compaction process of the garbling outputs. In this compaction process, encrypted truth table entries are compressed to eliminate bubbles caused by our static memory management policy. This step is necessary to reduce network transferring cost. And on the evaluator side, the time difference is spent in a reverse process of the generator's compaction process, which normalizes lengths of the garbling outputs.

In Table 4.2, it is also shown that the time spent in garbling is much longer than that for de-garbling. It fits the expectation since the cost of de-garbling a gate is reduced to 12.5% (for a 3-bit in 1-bit out gate) or 25% (for a 2-bit in 1-bit out gate) of garbling a gate when the permute-and-encrypt technique [65] is applied. Another reason would be that, in most cases, the computing logic of slots in one task is much more diverse (uses more different versions of the *Min_of_2*, *Min_of_2_mux* or *Add_One* circuits) than that in one slice. Such diversity results in greater synchronization cost under the SIMT architecture.

The computation between the two principals is also pipelined. In this test case, the generator usually completes its total computing tasks when the evaluator completes 93% of de-garbling slices. The overall running time, excluding networking delays, to compute the 5000×5000 test case is 1555 seconds, which translates to a throughput of 1.209×10^6 gates per second. Compared with the computing speed of 96000 gates per second [43], the acceleration rate is 12.5 folds.

Table 4.3 Major Memory Utilization on the Generator Side (ED)

	Generator side memory usage
L_G for DP[*1*][*j*] and DP[*i*][*1*]	1.1MB (host & GPU)
L_G for overflow	0.3MB (host & GPU)
circuits structural info of DP[*i*][*j*]	286MB (host)
relative start addresses of L_O, L_I, or L_G for each GC-slot in one task	0.3MB (host & GPU)
L_I for each GC-slot in one task	12.5 MB (GPU)
garbling output of one task	80MB (host & GPU)
L_O of GC-slots in latest three slices in the previous task	3.8MB(GPU)
network transferring queue	3.2GB (host)

Table 4.3 illustrates the major host & GPU memory consumption on the generator side. As shown in Table 4.3, the major memory utilization is storing circuit structural information for the 5000×5000 GC-slots. Comparing with that, the computation-related memory utilization for one task is negligible. In sum, the overall memory utilization excluding network transferring-related part is around 400MB. Due to the bursty of garbling outputs pushing into the network transferring queue as we observed, an empirical memory upper bound 3.2GB is set for the network transferring queue to prevent memory exhaustion. After counting in the network transferring memory cost, the total memory usage is around 3.6GB, which is acceptable for personal computers or small servers.

4.3 DP-level: Computing SFE-based Smith-Waterman

In this subchapter, we present the DP-level design for the SFE-based Smith-Waterman (SW) problem, especially the part different from that for the SFE-based ED problem. At the beginning of this subchapter, we present the structure of a GC-slot in the DP matrix of the SW problem. Our revised SW algorithm is displayed as Figure 4.7. The algorithm inputs are two genome sequences α and β from the generator and the evaluator respectively, a function gap(x) = $a + b\,x$ (where a and b are public co-efficients) and a 2-dimensional score matrix. Our selection of gap(x) and score matrix follows [65], that is, gap(x) = -12-7x and the score matrix BLOSUM62 [41]. There are 20 types of genome enumerated in BLOSUM62, and thus the bit-width of each symbol in α and β is 5.

```
Smith-Waterman(α, β, gap, score):
1: for i from 0 to α.length:
2:     DP[i][0] = 0;
3: for j from 0 to β.length:
4:    DP[0][j] = 0; (j=[0, β.length])
5: for i from 1 to α.length:
6:     for j from 1 to β.length:
7:         signed tmp = DP[i-1][j-1] + score[α[i]][β[j]];
8:         m = 0;
9:         for o from 1 to i, and then 1 to j:
10:             m = max(m, signed (DP[x][y]-|gap(o)|) ),
               here {x,y}={i-o,j}or{i,j-o}, DP[x][y] >= |gap(o)|
11:      DP[i][j] = max(m, signed tmp);
```

Figure 4.7 The Revised Smith-Waterman Algorithm

For convenience of parallelization, we make several small revisions to further partition the SW algorithm into steps (lines 7 and 11) of O(N^2) time complexity and steps (lines 8-10) of O(N^3) time complexity. Line 7 includes a lookup function that generates a score from the score matrix, and an addition function that sums results of DP[i-1][j-1] and the newly generated score. It is noticeable that the score would be a negative value, and thus the sum may also be negative. Therefore, the addition function needs to export a 1-bit sign flag as part of its output. For lines 8-10, the original version of SW algorithm differentiates the dependency among GC-slots in the same column and the same row, and separates the logic into two for loops (one for column, the other for row). Here, we consider them as the homogeneous inter-dependency with different sources of wire labels. This loop can be further translated as a sequence of {*signed_Subtraction*, *Max*} circuits. And such a sequence in an arbitrary GC-slot is denoted as *SEQ*. Line 11 only includes one *Max* circuit which compares the result of

SEQ and that of line 7. Noting that the value of an arbitrary DP[*i*][*j*] should always greater or equal to 0, this final *Max* circuit does need to export a sign flag.

At runtime, there is one opportunity for further optimization. Line 9 in Figure 4.7 indicates that the original length of SEQ is *i*+*j*. However, one node (DP[*x*][*y*] – gap(*o*)) in the SEQ can be skipped if it is negative for sure. Checking whether (DP[*x*][*y*] – gap(*o*)) of SEQ is negative does not break the privacy. According to the computation nature of the plain-text version of SW, the maximum possible value of DP[*x*][*y*] is min(*x*,*y*) × *SMAX*, where *SMAX* is the maximum positive value in BLOSUM62. The other operand of the subtraction, gap(*x*) = -12-7*x*, is also public. As such, if the maximum possible value of DP[*x*][*y*] ≤ gap(*o*), a node (DP[*x*][*y*] – gap(*o*)) is skipped.

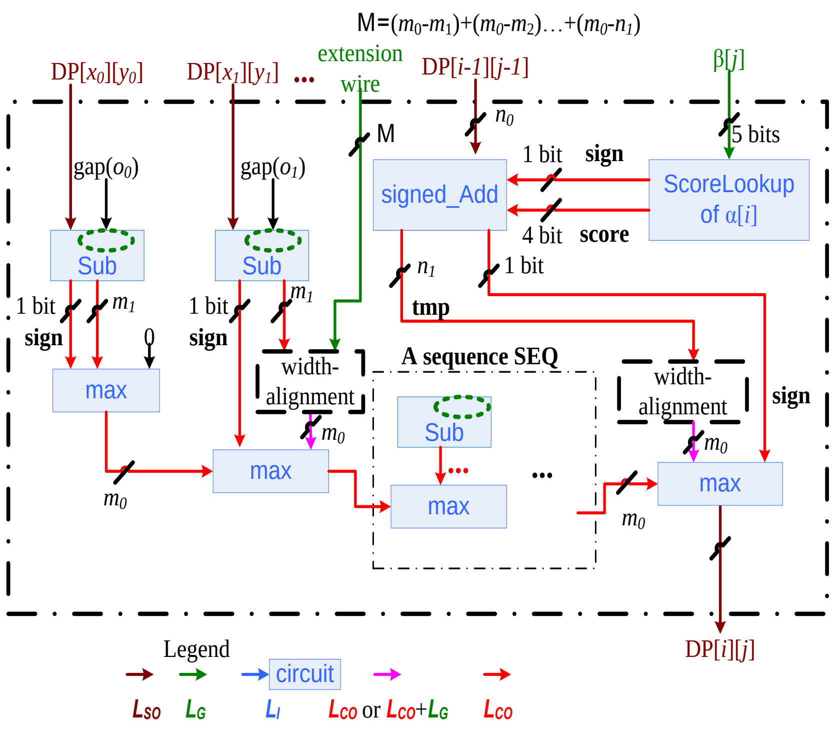


Figure 4.8 The SFE Building Block (a GC-slot at DP[*i*][*j*]) for SW

The structure of a GC-slot DP[*i*][*j*] is shown in Figure 4.8. It includes a *scoreLookup* circuit, a *signed_Addition* circuit, a sequence of *unsigned_Max*(*m*, *signed_Subtraction*(DP[*x*][*y*], |gap(*o*)|)) circuits, and an *unsigned_Max*(*m*, *tmp*) circuit. The gap(*o*) is treated as the generator's input wires since it is independent with the evaluator.

In Figure 4.8, colors for of L_I, L_O and L_G follow the color usage in last subchapter, but L_O is further divided to two different types of wire labels, denoted by L_{SO} and L_{CO}: L_{SO} represents the set of paired labels for wires of GC-slots' outputs, L_{CO} the set of paired labels for wires of circuits' outputs within GC-slots. Separating L_{CO} and L_{SO} is necessary to construct a more fine-grained parallel computing model for SW. L_I represents the set of paired labels for wires within circuits. Furthermore, L_I also include labels for extension wires for *Max* in SEQ, since the number of these wires for all GC-slots in the entire DP matrix is too large to be kept as *global* wire labels. Based on the same reason, if a garbled gate in *signed_Subtraction* does not accept any input from the evaluator, it are treated as a wire in L_I. L_G includes sets of wires labels for the overflow bits of the *signed_Addition* circuit of all GC-slots in the DP matrix, and the evaluator's input β[*N*]. Special cases are GC-slots DP[1][*j*] and DP[*i*][1]. Their GC-slot structure can be simplified as one *scoreLookup* circuit because DP[*i*-1][*j*-1]=0, and the outputs of *Max* circuits are always 0.

4.3.1 GPU Mapping Policies

The GPU resource mapping policies for the SFE-based SW problem is similar to that for the SFE-based ED problem. That is, the task partition of the *cross-slice-mapping* (CSMP) policy for the generator, and the partition of slice-by-slice policy for the evaluator are the same as that for the ED problem. For example, the task[0] on the generator side also contains 5120 GC-slots. And on the evaluator side, GC-slots are de-garbled slice by slice. However, the policies for the SFE-based SW problem are more fined-grained than that for the SFE-based ED problem because the former problem has much more complicated computation structures of its GC-slots.

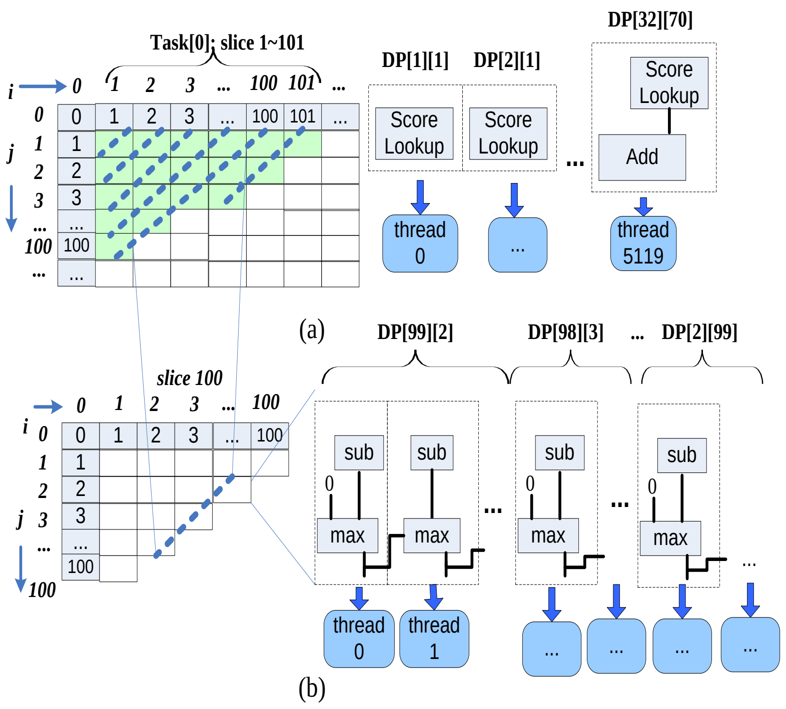


Figure 4.9 The Generator's CSMP (SW) for (a) Line 7 and (b) Lines 8-11 in Figure 4.7

Figure 4.9 (a) illustrates a snapshot of task[0]. It is clear that only {*scoreLookup*, *signed_Addition*} circuits for each GC-slot in task[0] are contained in Figure 4.9 (a). This part has O(N^2) time complexity, and therefore, each GC-slot has only one pair of {*scoreLookup*, *signed_Addition*} circuits. Next step is garbling the the O(N^3) time complexity part, in a manner of slice-by-slice and within the scope of one task. Taking the task[0] as an example, for each slice in current task, we calculate the number of paired {*signed_Subtraction*, *Max*} circuits of all GC-slots per slice. Then, each pair of {*signed_Subtraction*, *Max*} circuits of the current slice is mapped to one GPU gate garbler thread. Figure 4.9 (b) uses the 100th slice in task[0] as an example and illustrates the mapping policy for the O(N^3) time complexity part (lines 8-10) and line 11.

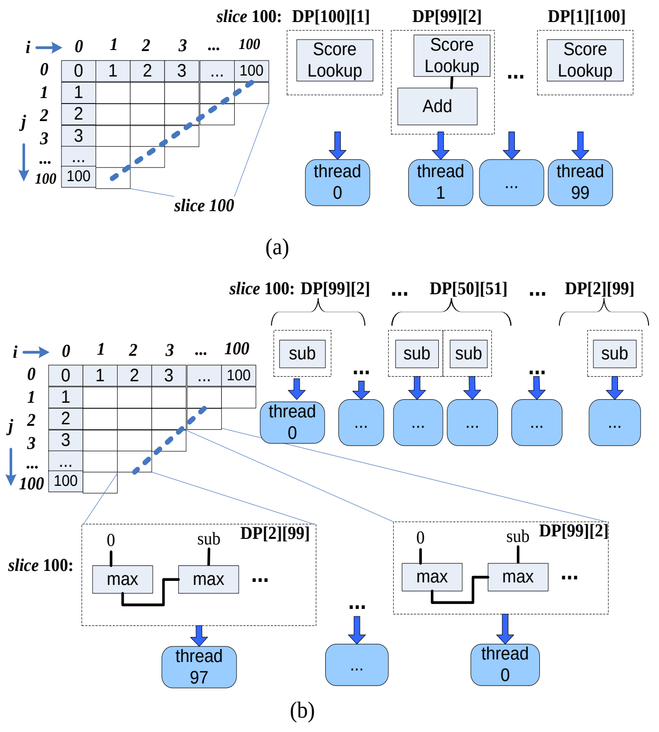


Figure 4.10 The Evaluator's slice-by-slice Mapping Policy (SW) for (a) Line 7 and (b) Lines 8-11 in Figure 4.7

The evaluator also has a fine-grained slice-by-slice resource mapping policy, which does not only separate the de-garbling processes of the O(N^2) time complexity part and the O(N^3) time complexity part, but also separates the de-garbling processes of *signed_Subtraction* circuits and *Max* circuits in all GC-slots of a slice. Figure 4.10 (a) illustrates the slice-by-slice mapping policy of 100 {*scoreLookup, signed_Addition* (if applicable)} circuits in the 100th slice to 100 GPU threads. Then, as shown in Figure 4.10 (b), for all *signed_Subtraction* circuits of all GC-slots in the 100th slice, each circuit is mapped to one GPU thread because they are independent of each other. Later, in the same slice, all *Max* circuits of one GC-slot are mapped to one GPU thread to enforce serial de-garbling of *Max* circuits.

4.3.2 Pipelined Scheduling

The assignment of wire labels for the SFE-based SW problem is also slightly different from that for the SFE-based ED problem. L_{CO} and L_I are pre-assigned at initialization of a new task, but L_{SO} and L_G at initialization of the SFE system. L_{SO} is treated as global variables during the entire privacy-preserving computing because their dependency crosses the DP matrix. The static memory allocation for L_{SO}, L_{CO}, L_G and L_I is similar to its counterpart in the ED problem. The only difference is the assessment of the maximum memory usage for saving L_{CO} and L_I for line 7 (in Figure 4.7) is per task, and L_{CO} and L_I for line 7-11 (in Figure 4.7) per slice.

Comparing with the SFE-based ED problem, the structure of GC-slots for the SFE-based SW problem contains richer flags: first, because all *unsigned_Max* circuits within one GC slot need to have the same I/O bit width, extension-wire flag is necessary for each circuit in the sequence of {*unsigned_Max*} circuits. Second, it saves start-offsets of L_{CO} associated with *unsigned_Max*, *signed_Subtraction* and *signed_Addition* circuits' output wires are contained. Third, each *unsigned_Max* circuit saves two offsets, one points to its predecessor *unsigned_Max* circuit in the SEQ, the other points to the *signed_Subtraction* circuit that passes its output to the *unsigned_Max* circuit.

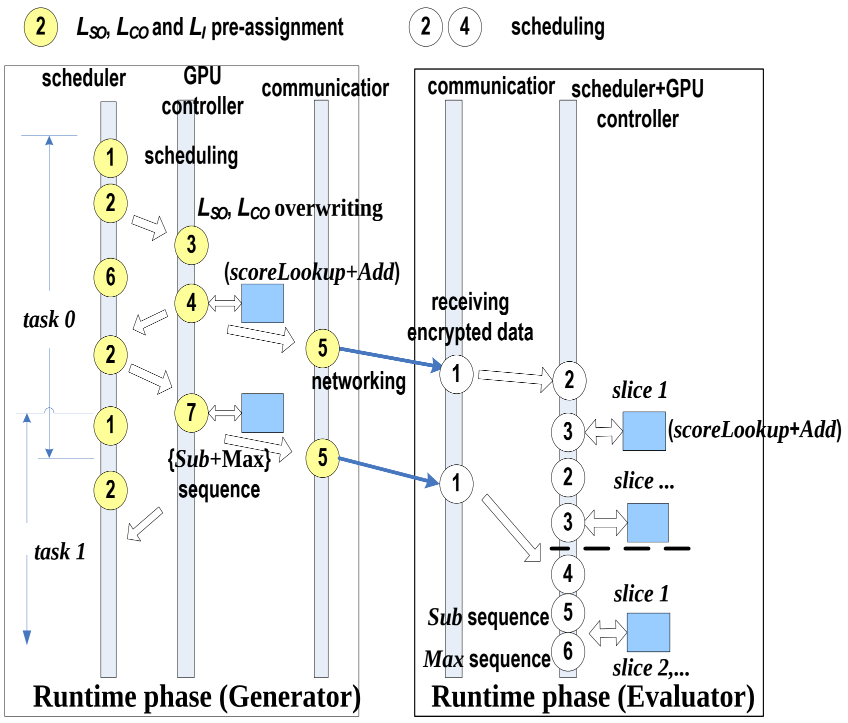


Figure 4.11 The Pipelined Garbling & De-garbling Process (SW)

Figure 4.11 illustrates the runtime pipeline of the garbling and de-garbling process for the SW problem. On the generator side, the step 3 overwrites paired wire label that map to outputs of *signed_Addition* circuits. Then the step 4 garbles *scoreLookup* plus *signed_Addition circuits* in the task[0]. Meanwhile, the step 6 schedules the garbling of the {*signed_Subtraction, unsigned_Max*} circuit sequence SEQ for all slices in the task[0]. Then, step 7 garbles the SEQ slice-by-slice.

On the other hand, the evaluator's step 2 schedules the de-garbling process for {*scoreLookup*, *signed_Addition*} slice by slice, and step 4 for the {*unsigned_Max*, *signed_Subtraction*} sequence. Step 3, 5, 6 de-garble {*scoreLookup* plus *signed_Addition*}, the *signed_Subtraction* circuits sequence, and the *unsigned_Max* circuits sequences respectively. In Figure 4.11, the benefit of pipelined computing line 7 and line 8-11 (in Figure 4.7) can be easily high-lighted. That is, part of the latency for transferring garbled result of *SEQ* of all GC-slots in one task is covered by the slice-by-slice de-garbling {*scoreLookup, signed_Addition*}.

4.3.3 Experiment Results and Analysis

Follow previous studies [43][52], we select the 60×60 SW problem as the test case to verify the efficiency of our system. Table 4.4 lists the break down of execution time on the generator and the evaluator side. In Table 4.4, it shows that a very large proportion of execution time is spent in the garbling (de-garbling) process of *Max* circuits on the generator (evaluator) side. It meets the expectation because the amount of *Max* circuits is huge, and all *Max* circuits within one GC-slot have to be de-garbled sequentially. Garbling and de-garbling process of *signed_Subtract* circuits takes much less execution time due to the independency of *signed_Subtract* within one GC-slot. In this test case, the execution times of garbling and de-garbling the time complexity O(N^2) part are trivial.

Table 4.4 Execution Time Break Down (SW)

Exec Time	**Generator**	**Evaluator**
SFE system initialization	2.6s	4.51s
Scheduling	0.0176	0.014s
garbling & de-garbling(*scoreLookup*)	0.02s	0.0018s
garbling & de-garbling(*signed_Addition*)	0.044s	0.037s
garbling & de-garbling(*signed_Subtract*)	0.45s	0.091s
garbling & de-garbling(*Max*)	2.7s	8.5s
Total computing latency	5.6s	8.64s

The time latency from the generator's task 0 to the evaluator's de-garbling of slice 119 is 9.69 seconds, and the total computing latency (two initialization phases + 9.69, excluding networking cost) is 16.79 seconds. This result represents a 24.7x acceleration factor over the computing time (415 seconds) for the same 60×60 SW problem reported in Huang *et al.* [43]. In terms of the memory usage, for the studied case, it took about 40MB to store encrypted truth table entries and permute-and-encrypt bits. The statically allocated memory for saving all paired wire labels of GC-slots' and circuits' outputs is less than 4MB.

4.4 Summary

According to the experimental result reported in subchapter 4.2 and 4.3, it can be concluded that the GPU-based parallel computing model proposed in this dissertation can effectively accelerate the SFE-based ED and the SFE-based SW problems. Comparing with the CPU-based SHA-256 version reported on Intel i7 CPU [37], our

GPU-based SHA-256 version in this work has roughly 3 folds throughput enhancement. Comparing with the CPU-version benchmark reported in [43], which parallel computed the SFE-based ED and SW problems on two computes with Intel Duo E8400 3GHz CPUs, our 10+ folds speed up rates can be due to reasons as follows: first, when mapping the computing structure of the SFE-based ED problem and SFE-based SW problem to the GPU-based gate garbler, the degree of parallelism provided by the gate garbler is fully utilized by the generator, and is maximally utilized by the evaluator while it needs to satisfy the inter-dependency among slices. Second, the pipeline mechanism causes few idle cycles on the CPU-GPU architecture. Third, the static memory management policy eliminates unnecessary memory allocation and release in both host and GPU memory spaces.

The CSMP and slice-by-slice GPU resource mapping policies are general policies for an arbitrary problem size N of SFE-based DP problems which satisfy the *wavefront* parallel patterns. Our experiment results further show that, if the time complexity reaches O(N^3), fine-grained mapping policies that partition time complexity O(N^3) part and O(N^2) part of a DP instance have better chance than coarse-grained counterparts to fully utilize the degree of parallelism provided by the GPU-based gate-garbler. Knowing the “general purpose” computing nature of the SFE protocol, our design experiences is also helpful for system design of other SFE-based problems.

A tool chain for an automatic execution process of SFE-based DP problems on GPU is the future purpose. Currently, the runtime execution part is automatic, but the offline part is still manual. To support an automatic offline process, a new language is needed to define the inter-dependency among GCs within one GC-slot of the DP matrix, and a new parser is needed to convert both SHDL-based GC files and the inter-dependency among GC-slots to structural information in memory.

CHAPTER V

PARALLEL ECC ALGORITHMS ON GPU

In this chapter, we present the parallel computing models of two ECC algorithms, point multiplication and bilinear pairing, on the contemporary GPU architecture. To study the computing requirement of point multiplication and bilinear pairing, we first study how to utilize ECC-based PSI and SH in the health-care cloud service SAPPHIRE [77] to protect the health records of patients (privacy information) from the cloud service provider.

Four roles of SAPPHIRE are formally defined as follows: the patient(s) *Bob*, the clinic(s) *Alice*, the cloud service provider (CSP), and the request routing server (RRS). An instance of the SH protocol is called when Bob authenticates a request from Alice, or the RRS authenticates a request from Bob or Alice. As a 1-to-N server setting, the RRS may need to authenticate a large number of clients (Alice or Bob) in a short time interval. A PSI instance is called when Alice queries Bob's health record from the CSP. When such a query occurs, the CSP first sends the RRS a large number of encrypted health records, in which only one of the records is Bob's. And then Alice runs a PSI instance with the RRS to receive Bob's health record without telling the RRS which record it is interested in. In one PSI instance, each encrypted health records on the RRS side triggers a number of point multiplication on both the RRS side and the Alice side. Therefore, when the RRS meets bursts of urgent PSI or SH transactions in an emergency response situation, it requires a solution that is capable to handle a large volume of PSI and SH transactions timely.

For security considerations, it is assumed that the server can be compromised, and it is unclear that GPU forbids other host processes to read the GPU memory. Therefore, when designing utilization scenario for GPU-based point multiplication and bilinear pairing, only the *public* computation are considered. On the RRS, the two inputs of bilinear pairing are the CSP/customer/clinic's public key and a group secret (a large integer) encrypted via ECC point multiplication. Both inputs are public data and therefore they can be safely computed on a server without privacy concern.

In the rest of this chapter, we first analyze the PSI and SH protocols used in SAPPHIRE. Next, we present the parallel computing model of multi-precision arithmetic operations, which are fundamental functions for both ECC algorithms. Then, we discuss optimization techniques applied for high-level arithmetic operations in both ECC algorithms. And then, the experimental results are presented. In the end, the major bottlenecks of parallelized ECC algorithms on contemporary GPU architecture are analyzed.

5.1 Computation Requirements in PSI and SH

Figure 5.1 illustrates the authentication process between the RRS and Alice. For simplicity, Figure 5.1 only illustrates part of the authentication process which involves bilinear pairing:

As a prior process occurred before launching an SH session between Alice and the RRS, Alice had negotiated a group secret ss_a with the RRS when it was registered in this cloud service, and the RRS returned $sR = [ss_a]\ P_R$ to Alice as the registration result. In this prior process, the RRS did not directly return ss_a to prevent Alice from guessing how a group secret is generated.

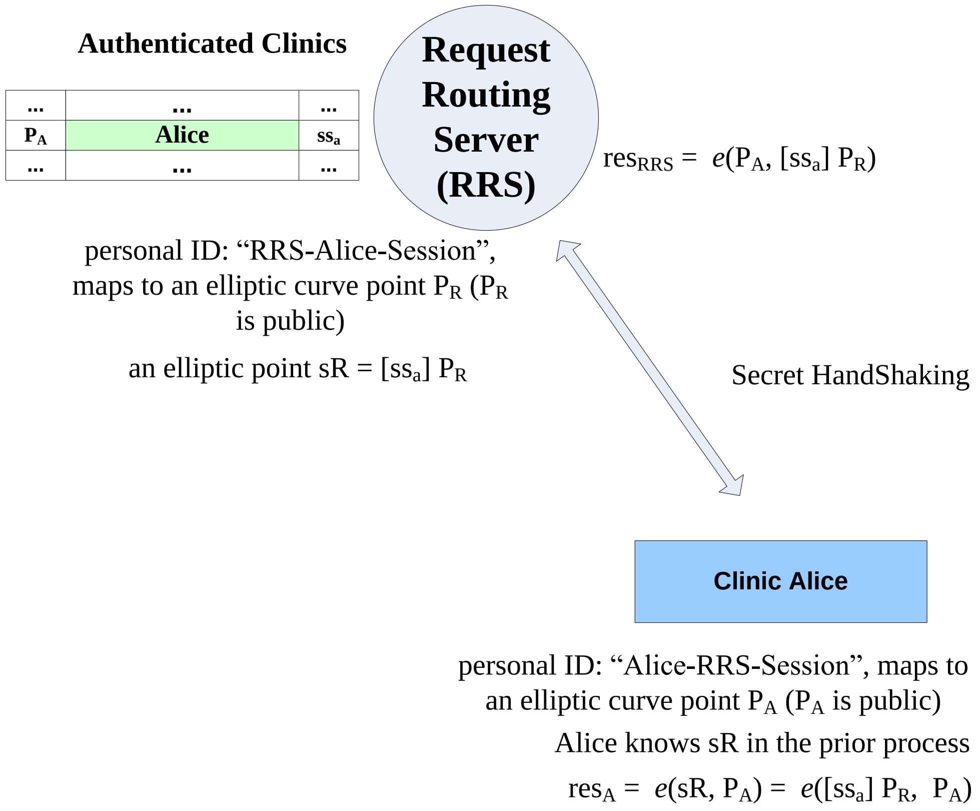
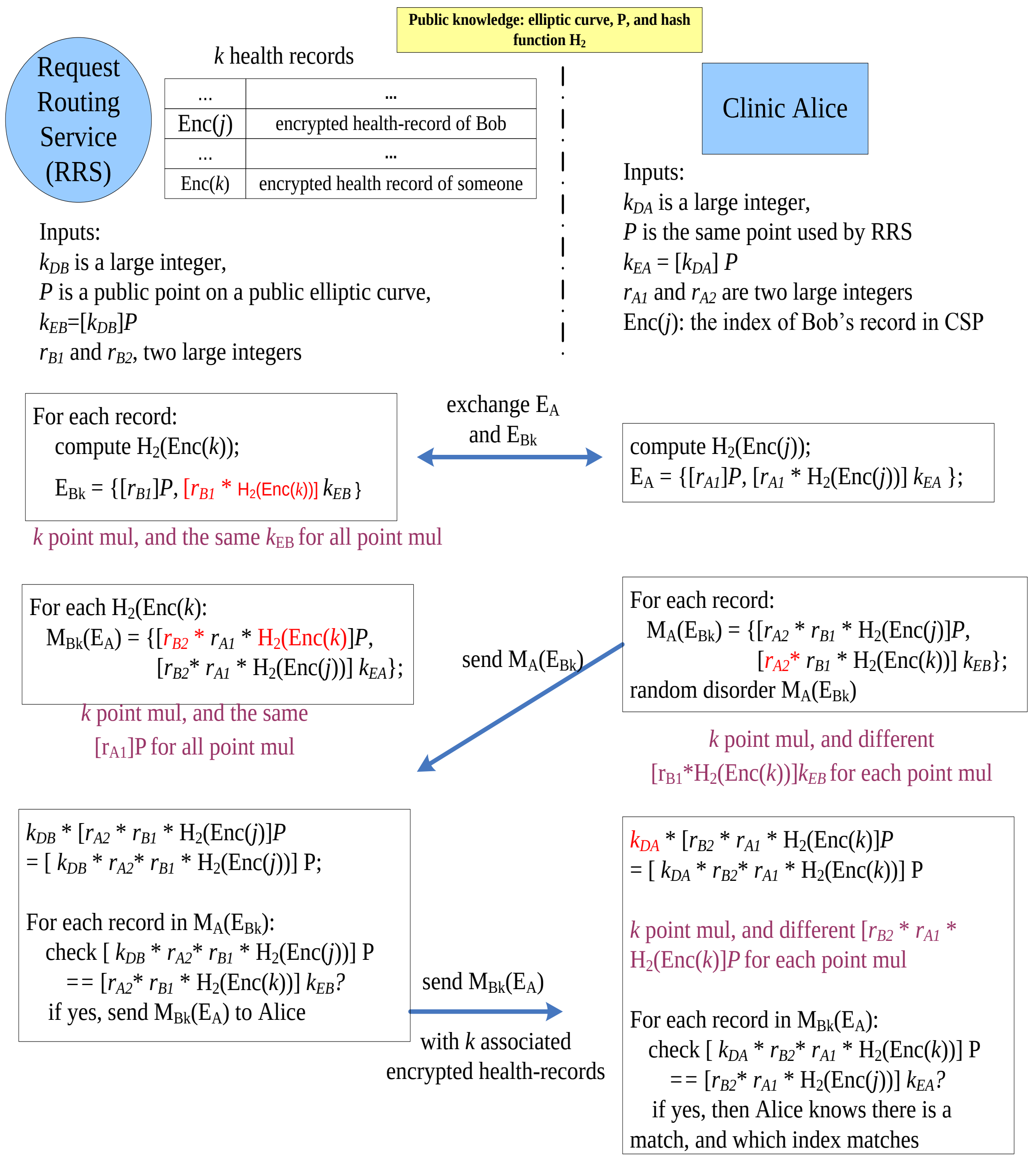

Authenticated Clinics
...
P_A
Alice
ss_a
Request Routing Server (RRS)
$res_{RRS} = e(P_A, [ss_a]\ P_R)$
personal ID: "RRS-Alice-Session", maps to an elliptic curve point P_R (P_R is public)
an elliptic point $sR = [ss_a]\ P_R$
Secret HandShaking
Clinic Alice
personal ID: "Alice-RRS-Session", maps to an elliptic curve point P_A (P_A is public)
Alice knows sR in the prior process
$res_A = e(sR, P_A) = e([ss_a]\ P_R, P_A)$

Figure 5.1 The Secret Handshake between Alice and the RRS

Then, as shown in Figure 5.1, an SH session starts between Alice and the RRS. In Figure 5.1, P_R is an elliptic curve point mapped from the RRS's personal ID. Similarly, Alice also has a point P_A associated with its ID. These personal ID can be arbitrary strings, and hence do not necessarily associate with their identities such as the full name, the driver license No., or the MAC address. In an SH session, Alice computes a pairing function $e(sR, \mathrm{P}_{\mathrm{A}}) = e([\mathrm{ss}_{\mathrm{a}}]\ \mathrm{P}_{\mathrm{R}}, \mathrm{P}_{\mathrm{A}})$; on the other hand, the RRS computes another pairing function $e(\mathrm{P}_{\mathrm{A}}, [\mathrm{ss}_{\mathrm{a}}]\ \mathrm{P}_{\mathrm{R}})$. According to the bilinear property $e([a]\ \mathrm{P}, \mathrm{Q}) = e(\mathrm{P}, [a]\ \mathrm{Q})$, if and only if the two principals share a common group secret ss_{a}, results of the two pairing functions can be the same.

For the case that the RRS authenticates different clinics in a short time interval, several observations are drawn as follows. First, these authentication requests are independent to each other. Second, on the RRS side, the code path of the pairing function is determined by the order of the select elliptic curve, not the inputs P_{A} and $[\mathrm{ss}_{\mathrm{a}}]\ \mathrm{P}_{\mathrm{R}}$. And hence, when all players in the system use the same elliptic curve for SH, different bilinear pairing instances on the RRS have the same code path, and hence fit the restriction of “single instruction” in the SIMT architecture.

Next, we present the utilization of PSI in SAPPHAIRE. A PSI session is called when the RRS returns the patient Bob’s health record to the clinic Alice. The purpose of using PSI is protecting Bob’s health record from a carious RRS who may assess Bob’s health status based on clinics’ requests, or an unauthorized Alice who wants to guess the storage structure of health records on the CSP. In order to protect the privacy of Bob, the CSP will return k health records to the RRS, and then Alice runs a PSI session with the RRS to select one health record. No matter the result of the PSI session, Alice does not know any information of other health records, and the RRS does not know which record is matched by the Alice. If Alice is authorized to access Bob’s record, then the PSI session will return it Bob's record, and both Alice and the RRS know a unique match occurs; otherwise, the PSI session indicates that nothing is matched and the RRS rejects Alice's query.



Public knowledge: elliptic curve, P, and hash function H_2
Request Routing Service (RRS)
k health records
… | …
Enc(j) | encrypted health-record of Bob
… | …
Enc(k) | encrypted health record of someone
Clinic Alice
Inputs:
k_{DB} is a large integer,
P is a public point on a public elliptic curve,
$k_{EB}=[k_{DB}]P$
r_{B1} and r_{B2}, two large integers
Inputs:
k_{DA} is a large integer,
P is the same point used by RRS
$k_{EA} = [k_{DA}] P$
r_{A1} and r_{A2} are two large integers
Enc(j): the index of Bob's record in CSP
For each record:
compute $H_2(\mathrm{Enc}(k))$;
$E_{Bk} = \{[r_{B1}]P, [r_{B1} * H_2(\mathrm{Enc}(k))]\, k_{EB}\}$
k point mul, and the same k_{EB} for all point mul
exchange E_A and E_{Bk}
compute $H_2(\mathrm{Enc}(j))$;
$E_A = \{[r_{A1}]P, [r_{A1} * H_2(\mathrm{Enc}(j))]\, k_{EA}\}$;
For each $H_2(\mathrm{Enc}(k)$:
$M_{Bk}(E_A) = \{[r_{B2} * r_{A1} * H_2(\mathrm{Enc}(k)]P, [r_{B2} * r_{A1} * H_2(\mathrm{Enc}(j))]\, k_{EA}\}$;
k point mul, and the same $[r_{A1}]P$ for all point mul
send $M_A(E_{Bk})$
For each record:
$M_A(E_{Bk}) = \{[r_{A2} * r_{B1} * H_2(\mathrm{Enc}(j)]P, [r_{A2} * r_{B1} * H_2(\mathrm{Enc}(k))]\, k_{EB}\}$;
random disorder $M_A(E_{Bk})$
k point mul, and different $[r_{B1} * H_2(\mathrm{Enc}(k))]k_{EB}$ for each point mul
$k_{DB} * [r_{A2} * r_{B1} * H_2(\mathrm{Enc}(j)]P = [\, k_{DB} * r_{A2} * r_{B1} * H_2(\mathrm{Enc}(j))]\, P$;
For each record in $M_A(E_{Bk})$:
check $[\, k_{DB} * r_{A2} * r_{B1} * H_2(\mathrm{Enc}(j))]\, P == [r_{A2} * r_{B1} * H_2(\mathrm{Enc}(k))]\, k_{EB}$?
if yes, send $M_{Bk}(E_A)$ to Alice
send $M_{Bk}(E_A)$
with k associated encrypted health-records
$k_{DA} * [r_{B2} * r_{A1} * H_2(\mathrm{Enc}(k)]P = [\, k_{DA} * r_{B2} * r_{A1} * H_2(\mathrm{Enc}(k))]\, P$
k point mul, and different $[r_{B2} * r_{A1} * H_2(\mathrm{Enc}(k)]P$ for each point mul
For each record in $M_{Bk}(E_A)$:
check $[\, k_{DA} * r_{B2} * r_{A1} * H_2(\mathrm{Enc}(k))]\, P == [r_{B2} * r_{A1} * H_2(\mathrm{Enc}(j))]\, k_{EA}$?
if yes, then Alice knows there is a match, and which index matches

Figure 5.2 Private Set Intersection between Alice and the RRS

Figure 5.2 illustrates such a PSI session between Alice and the RRS. Before the PSI session launches, the CSP sends the RRS k encrypted health records associated with the cipher-text of their storage indices in the CSP. The encrypted storage indices are denoted by Enc(k), and that associates with Bob's health record is denoted by Enc(j), which is generated by the CSP when Bob's health record is submitted. If Alice is authorized by Bob to access its health record, Alice has received Enc(j) from Bob.

In one PSI session, the RRS holds k encrypted health records, but only one of them belongs to Bob, which is Alice's query target. If Alice has been authorized by Bob, the RRS can find a match in the end of the PSI process. Due to the randomly disordering step of Alice, RRS does not know which entry in the original set is matched. As such, the RRS cannot assess Bob's health status according to the medical specialty of Alice. If no match had found by the RRS, the RRS rejects to send any health records to Alice.

In one PSI session, the possibility of the RRS to correctly guess out which entry Alice is looking for is $1/k$. Larger k means better protection for Bob's health status, and therefore larger k is preferred. To complete one PSI session, Figure 5.2 shows that there are $2\times k$ point multiplications on each side. On the RRS side, the point multiplication computations are in the format of $[I_k]\ P$, where the point P is the same in the k instances. However, on Alice side, the format is $[I]\ P_k$, where points P_k are different and the scalar I is the same in the k instances. As we will present in subchapter 5.3, comparing with than $[I_k]$ P, $[I]$ $\mathrm{P_k}$ requires much more computation cost for generating the pre-calculated tables for $\mathrm{P_k}$ during the computing process of point multiplication.

5.2 Computing Model of Multi-precision Arithmetic

The fundamental arithmetic functions of point multiplication and bilinear pairing are multi-precision arithmetic such as add/sub/mul/div/exp, whose operands have $n \times 32$ bit length and $n \geq 2$. Our first task for exploring the design and the performance evaluation of point multiplication and bilinear pairing on the contemporary GPU architecture is designing the low level parallel computing model for the multi-precision arithmetic and the storage format of arithmetic operands, and selecting a suitable number system for GPU.

Before presenting the formal design of the parallel computing model of ECC algorithms, we first evaluate the throughput of INT32/SPF instructions on GTX-680. The initial result is in Table 5.1, which shows that single addition and multiplication in SPF is roughly 3-folds faster than its INT version.

Table 5.1 Execution Time of INT32 and SPF arithmetic on GTX-680

Instruction	INT32 A+B	SPF A+B	INT32 A×B	SPF A×B	INT32 A×B+C	SPF A×B+C
Exec times (ms)	0.0308	0.012	0.0326	0.011	0.0114	0.0117

As we know, earlier generation GPUs have only implemented fused multiplication plus addition (FMA) as one instruction for SPF. However, Table 5.1 indicates that such a FMA for INT32, or a similar mechanism, has been supported by GK104 architecture. This may be the reason why SPF and INT32 have similar throughputs of MUL+ADD. As such, using SPF may be a good choice, and using INT32 is also acceptable. Eventually, INT32 is adopted because of the reasons as follows.

Given the parameters of elliptic curve and security strength select in this dissertation, each operand of the low-level multi-precision arithmetic is an 8×32 bit = 256 bit integer (n = 8). Two previous studies [12][70] used SPF/DPF to represent the integer operands of low-level multi-precision arithmetic. In [12], the bit length of their operands is 224-bit, and they saved one 224-bit integer into an array of 24 floating points, each floating point saved 10 bits of the original integer. Such a representation triples (and more) the memory consumption, which is one of the reasons result in a relative small degree of parallelism of their computing model. Instead, we adopted the SPF type but represented a 256-bit integer by a float32[8] array. And then the round up problem is met during iteratively MUL+ADD of floating point values. Converting intermediate floating point results to integer after several MUL+ADD might avoid rounding up effect, but there is no good way to ensure an optimal insertion method. [70] represented a 256-bit integer as in the polynomial format with 12 double-precision floating point coefficients. Knowing that the throughput of Double-Precision-Floating (DPF) is 1/24 of that of SPF on GTX-680, such a DPF-based data representing is not advisable on Kepler Architecture.

The next problem is selecting a suitable number system for GPU. Previous studies on parallelization of ECC algorithms were mainly based on either conventional Montgomery [68] or Residue Number System (RNS) Montgomery system. In the conventional Montgomery system, the modular multiplication $c = a \times b \bmod q$ is implemented by one multiplication step T = $a \times b$ following by one reduction step *reduct*(*T*). The *reduct*(*T*) avoids to call the expensive division operation. On the other hand, based on the *Chinese Reminder Theorem* (CRT), RNS decomposes a modulus *M* to *n* co-prime integers m_1, m_2, …, m_n, then an arbitrary integer $X < M$ can be uniquely represented as $x_i = X \bmod m_i$, $1 < i < n$, $M = \prod_{i=1}^{n} m_i$. Computation in mod m_i is independent with each other, so that RNS is well suited for the SIMT architecture. However, the RNS system cannot be directly used in a prime field since the modulus *M* is not a prime, unless two extra Base Extension (BE) steps [8] are inserted in the reduction step. As such, a multi-precision modular multiplication $a[n] \times b[n] \bmod q$ in RNS needs $2n^2+5n$ 32-bit MUL by using four threads, while $2n^2+n$ MUL in the conventional Montgomery by using one thread. The other two extra costs are from the synchronization for the complicated comparison under RNS in each reduction and modular subtraction, and the potential branch divergence in each modular subtraction. Such synchronization cost grows as more threads are set to compute one instance.

Although the computing cost of one modular multiplication in the parallelized RNS is greater than that in the (serial) conventional Montgomery, as concluded in [19], the parallelized RNS is more efficient when it is applied to a long addition sequence of modular multiplication $(a \times b + c \times d + e \times f + \ldots) \bmod q$. Such sequences frequently appear in the bilinear pairing algorithm, and lengths of such sequences are closely dependent to which extension fields the *lazy reduction* technique [3] is applied to. Therefore, comparison of RNS and conventional Montgomery will be discussed together with the evaluation of general lazy reduction in subchapter 5.4. In this subchapter, we quantitatively evaluate the performance of multi-precision arithmetic in the conventional Montgomery system.

In terms of the suitability of parallelizing multi-precision arithmetic in the conventional Montgomery system, we note that the multi-precision multiplication T=a×b is consisted of multiple independent sequences of MUL+ADD operations, mapping them to multiple threads is similar to mapping computing of multiple sub-residues in RNS to multiple threads. Second, the reduction step *reduct*(T) includes two multiplications and one addition, most of which can be easily parallelized.

In this research, because $a \times b$ mod q in the base field F_q is not only the most expensive operation in F_q, but also the most frequently invoked low-level arithmetic operation in point multiplication and bilinear pairing, it is selected as the representative code segment to evaluate several parallel computing model options. As aforementioned, in the conventional Montgomery system, $c = a \times b$ mod q is composed of (1) $T = a \times b$ and (2) c = *reduction*(T). In both point multiplication and bilinear pairing algorithms, the modulus q, elements a, b, c needs one **uint32[8]** array, and the intermediate result T one **uint32[16]** array. Four parallel computing models are worth evaluating, they can be formally named as *CI*-1/2/4/8*thread* models, where each *computing instance* (CI) is performed by 1/2/4/8 co-operative thread(s). Another common computing model, known as the *bit-slice* model proposed in [54], is ignored in our work because the reduction in this model shows highly sequential nature.

In the computation of (1) $T = a \times b$ and (2) c = *reduction*(T), the access pattern of a and b in the shared memory space can affect the access speed and thus affect the overall performance. In the shared memory, operands *a* and *b* are defined as "**__shared__ uint32[8× BLK_CI_SIZE]**", where **BLK_CI_SIZE** is the number of CIs per block. Taking the CI-2thread model as an example, each two threads access the same **uint32**[8*i*+0,8*i*+7], 0 < *i* < **BLK_CI_SIZE**. As such, threads {0,16}, {2,18}, ..., {12,28}, {14,30} will concurrently read the eight **uint32**[0], which is a typical access pattern in the multiplication addition/subtraction/multiplication. However, knowing that GK-104 has 32 64-bit shared memory banks, the real concurrency of such an access pattern is only 1/2 of the expectation because threads 0 and 16 are access different ties (a low-level GPU memory architecture) of bank 0 and thus they compete the memory interface. Because each SMX in GK-104 has 32 LD/ST units, analyzing this type of bank conflict is limited within a warp of threads.

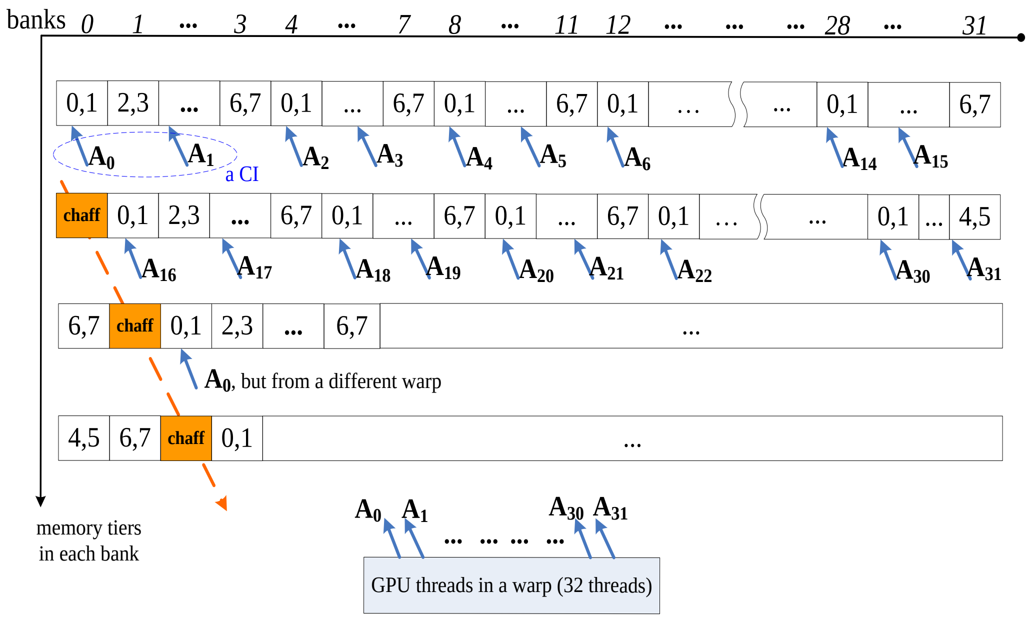

banks
0 1 ... 3 4 ... 7 8 ... 11 12 ... ... 28 ... 31
0,1 2,3 ... 6,7 0,1 ... 6,7 0,1 ... 6,7 0,1 ... ... 0,1 ... 6,7
A0 A1 a CI A2 A3 A4 A5 A6 A14 A15
chaff 0,1 2,3 ... 6,7 0,1 ... 6,7 0,1 ... 6,7 0,1 ... ... 0,1 ... 4,5
A16 A17 A18 A19 A20 A21 A22 A30 A31
6,7 chaff 0,1 2,3 ... 6,7 ...
A0, but from a different warp
4,5 6,7 chaff 0,1 ...
memory tiers in each bank
A0 A1 A30 A31
GPU threads in a warp (32 threads)

Figure 5.3 Collision-free Memory Access of a 256-bit Variable (*CI-2thread*)

Taking the CI-2thread model as an example, to remove such bank conflicts, a strip of 64-bit *chaff spacer* in the front of every eighth-**uint32[8]**. Figure 5.3 illustrates such an insertion scheme for saving $a[8\times$**BLK_CI_SIZE**$]$ in the shared memory. Furthermore, the insertion of the chaff spacer in the 3rd row of Figure 5.3 is not necessary, but it is placed there to simplify locating the start addresses of each a. After applying this insertion scheme, threads 0 and 16 read bank 0 and 1 respectively when they are reading the first element of **uint32[8]** associated with their CIs. As such, this type of bank conflict is removed. We further insert spacer to ensure the address of each variable in the shared memory always starts from bank 0.

Next, the workload balance of $T = a \times b$ is analyzed. The workload of $T[16]=a[8] \times b[8]$ can be considered as eight sequences of $T[x]=a[i] \times \mathrm{b}[j=0\sim7]$, $0\leq i\leq7$, and $x=i + j$ respectively. Considering the overflow effect of $a[i] \times \mathrm{b}[j]$, each sequence would rather be $T[x]$ += (low 32-bit) $a[i] \times b[\mathrm{j}=0,1,...,7]$, and $T[x+1]$ = (high 32-bit) $a[i] \times b[\mathrm{j}=0,1,...,7]$. Therefore, the inter-dependency among sequences are the R/W order of $T[x]$ and $T[x+1]$. If each $T[0\text{-}15]$ vector in the shared memory is partitioned into multiple segments with a constant size, each segment is mapped to one thread, and R/W address of each thread has a constant offset which is large enough, our observation is that there would be no race condition on R/W $T[x]$ and $T[x+1]$. Considering the bank width on GK104 is 64-bit, meaning $T[x]$ and $\mathrm{T}[x+1]$ are in the same bank (if x is even), the size of the segment should be at least one bank width. It also implies the infeasibility of the *CI-8thread* model where two neighboring threads in one CI would simultaneously read and write the same bank respectively and thus causes the inter-CI bank conflict. This observation indicates that the multi-precision multiplication $T[2n]=a[n] \times b[n]$ under the conventional Montgomery system shows a similar parallel workload balance result to that under RNS. As a result, the parallelized $T[16]=a[8] \times b[8]$ in the *CI-2/4thread* models are designed as shown in Figure 5.4.

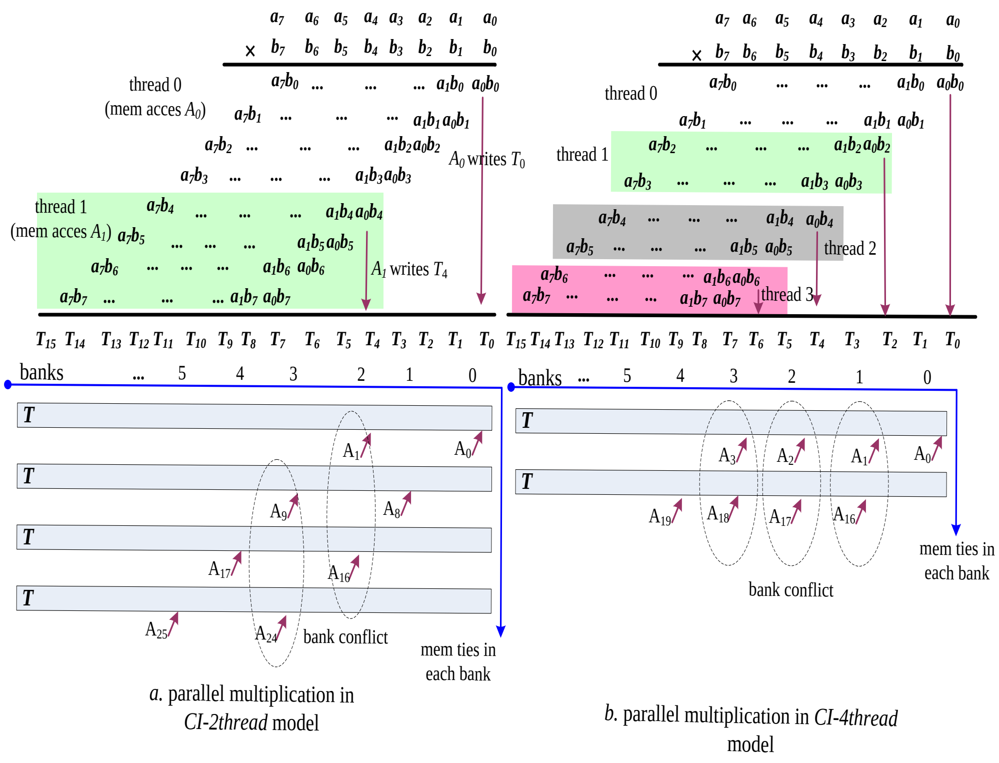
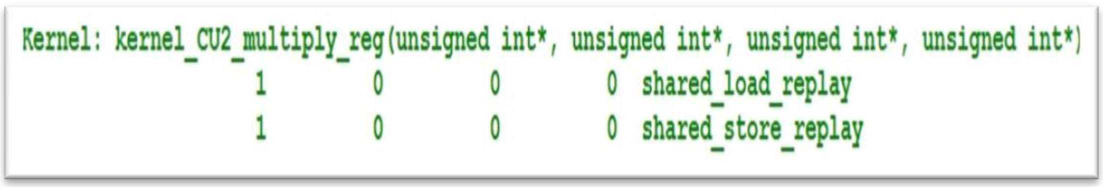


Figure 5.4 Parallel Multi-precision Multiplication $T = a \times b$

We prove that there is no bank conflict within each CI and across multiple CIs as follows. For the *CI-2thread* model, Taking the first CI thread {0,1} as an example, as shown in Figure 5.4.a, thread 0 computes $a_i \times \{b_0, b_1, b_2, b_3\}$, and thread 1 computes $a_i \times \{b_4, b_5, b_6, b_7\}$. Here, operands b_j read by thread 0 are on bank 0 or 1, and that read by thread 1 are on bank 1 or 2. When thread 0 updates $\{T_x, T_{x+1}\}$, thread 1 is updating $\{T_{x+4},$

T_{x+5}}, which are two banks away from {T_x, T_{x+1}}. For the *CI-4thread* model, as shown in Figure 5.4.b, thread 0 computes $a_i \times \{b_0, b_1\}$, thread 1 $a_i \times \{b_2, b_3\}$, thread 2 $a_i \times \{b_4, b_5\}$, and thread 3 $a_i \times \{b_6, b_7\}$. Here, operands b_j read by thread 0/1/2/3 are on bank 0/1/2/3 respectively. When thread 0 updates {T_x, T_{x+1}}, thread 1/2/3 is updating {$T_{x+2/\,x+4/\,x+6}$, $T_{x+3/\,x+5/\,x+7}$} respectively. The writing address of each thread is one bank away from each other. It is evident that threads in the same CI, or across different CIs do not meet any bank conflict during computing $a[i] \times b[j=0\sim7]$ in current execution order.

However, a new type of bank conflict across-CIs on GK104 is clearly shown in Figure 5.4. Such bank conflicts occur when threads sum the multiplication results and write back to *T*. Comparing with earlier GPU architectures, the bank width of GK104 grows from 32-bit to 64-bit, that is, *T*[0-15] in one tier are stored denser than in earlier GPUs. As such, two segments of *T* with different start offsets and associated with different CIs may be placed on the same bank. One example is shown in Figure 5.4, in the *CI-2thread* model, threads {0,1}, {8,9}, {16,17}, {24,25} for four CIs are writing result $T_0 = a_0 \times b_0$ and $T_4 = a_0 \times b_4$ back to shared memory, the banks of T_0 and T_4 for the four CIs are {0,2}, {1,3}, {2,4} and {3,5}. Noting that these eight threads are in the same warp, and thread 1 and thread 16 are competing for bank 2. To eliminate this type of bank conflict, 12 registers are used to temporarily save the multiplication results of T_0-T_{11} or T_4-T_{15} for each thread, and then a serial step accumulates results of two threads. In the end, the NVIDIA profiling tool *nvprof* is utilized to validate the bank conflict elimination schemes discussed in this subchapter, and the profiling result of the number of bank conflict of $T=a\times b$ in the CI-2thread model is shown in Figure 5.5.

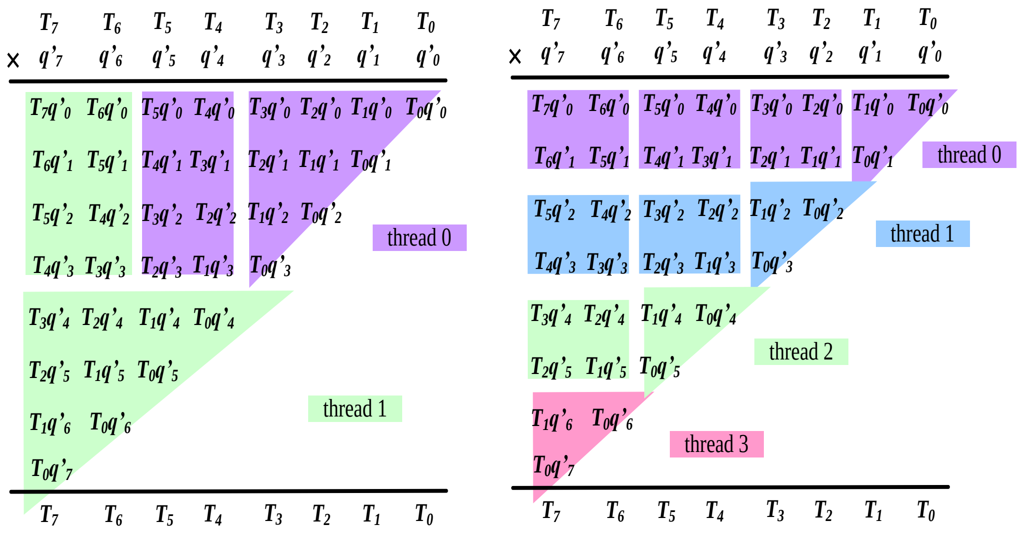


```
Kernel: kernel_CU2_multiply_reg(unsigned int*, unsigned int*, unsigned int*, unsigned int*)
                  1        0        0        0  shared_load_replay
                  1        0        0        0  shared_store_replay
```

Figure 5.5 Profiling Result of Bank Conflict for $T = a \times b$ (CI-2thread)

Theoretically, $a[n] \times b[n]$ (n=8) in the *CI-2thread* model cost $n^2/2$ MUL, which is the same as computing it in RNS when using 2 threads per instance. Taking into count of the synchronization overhead for bank conflict elimination, the actual cost of $a[n] \times b[n]$ is slightly above $n^2/2$ MUL. On the other hand, elimination of the type of bank conflict presented in the previous paragraph is also necessary under RNS if $T[2n]$ is consecutively placed in shared memory, and thus switching to RNS would not bring obvious extra gain for multi-precision multiplication. (An alternative, which saves $T[2n]$ as n separate variables in shared memory, would avoid this bank conflict. But this option will grossly complicate the code structure of low level multi-precision arithmetic functions, especially the division function).

According to [68], the multi-precision version of (2) $c = reduction(T=a \times b)$ is serially optimized as an iterative loop, where the dependency across iterations impedes the parallelization [54][98]. For parallelization, the *CI-2thread* model and the *CI-4thread* model adopt the single-precision version of reduction, which includes two parallelized multiplication (1) $m = (T \bmod R)\, q' \bmod R$, and (2) $m \times q$, and one parallel addition $T+mq$, where $R \times R^{-1} - q \times q' = 1$ and R = 2^{256}. Since (1) $m = (T \bmod R)\, q' \bmod R$ computes the low-256 bit half of T, this step costs only 56% of MUL+ADD of a full multi-precision multiplication. The parallelization solution for this step is similar to that for a full multi-precision multiplication, which is shown in Figure 5.6. It is shown that the workload can be equally balanced in the *CI-2thread* model, but in the *CI-4thread* model, explicit synchronization is necessary since the workloads per thread are different.



a. (T mod R) * q' mod R in *CI-2thread*

b. (T mod R) * q' mod R in *CI-4thread*

Figure 5.6 Implementation of *Reduction*(T)

To evaluate which computing model option results in the optimal throughput, we compare the combination of serial/parallel multiplication (1) $T = a \times b$ plus (2) serial or parallel version of *reduction*(T) (S-/P-*reduct*) in the *CI-1/2/4thread* model on GTX-680. The degree of parallelism of GPU in the *CI-1/2/4thread* models are 160/352/738 GPU threads per block, equivalent to 160/176/184 CIs per block. Furthermore, when computing $T + m \times q$, the parallel version of *reduction*(T) (denoted by P-*reduct*) reads T from global memory. These configuration parameters are optimized settings, borrowed from the shared memory usage analysis of bilinear pairing, as will be discussed later.

11938 times of multi-precision multiplication and 8312 times of reduction are repeated, which are an estimation of these two functions in one complete *R-ate* pairing. Here, the number of multi-precision multiplication and reduction is not 1:1 matched because the lazy reduction scheme has been applied.

Table 5.2 Performance of 11938 *mul* + 8321 *reduct* in Various Models

models	$T=a\times b$	S-*reduct*	P-*reduct*	best mul+reduct	thread per SMX	throughput (/sec)
CI-1thread	57.87ms	39.95ms	N/A	97.82ms	160	13085.3
CI-2thread	39.40ms	36.71ms	48.73ms	76.11ms	352	18499.5
CI-4thread	28.58ms	51.64ms	83.59ms	80.22ms	738	18349.5

The execution time of multiplication and *reduction*(T) in the *CI-1/2/4thread* models are listed in Table 5.2. The last column of Table 5.2 presents the throughput (per second). According to Table 5.2, several important conclusions can be drawn:

(1) The parallelization of multi-precision multiplication $T = a \times b$ works. However, the gain from parallelization shrinks as the thread count per CI increases. A possible reason of this shrinking effect is, as the thread count per CI grows, more synchronization is needed for summing $T[i]$ in registers;

(2) If the shared memory usage per CI is bisected into two threads, the increase of thread number per SMX is usually greater than doubling. Due to the limit of placing complete warps into SMX, a large shared memory usage per THREAD usually results in an insufficient shared memory utilization rate. As each thread consumes less shared memory resource, it is possible to put more complete warps

into SMX to approach the shared memory limit. As a result, we observe a maximum degree of parallelism per SMX in the *CI-4thread* model in Table 5.2. The positive effect of greater thread count per instance on the shared memory utilization rate is one of the reasons that result in a higher throughput in the *CI-2thread* model than in the *CI-1thread* model;

(3) The serial version of *reduction* (S-*reduct*) in the *CI-2thread* model is slightly faster than that in the *CI-1thread* model, possibly due to an outlier of the micro-architecture;

(4) In the *CI-2thread* model, the parallel version of *reduction* is much slower than its serial counterpart. The breakdown of execution time shows that computing $m = (T \bmod R)\ q' \bmod R$ and $m \times q$ took 43% execution time of *reduction*, the rest time is spent in the addition $T + m\ q$ because the T in this addition is a copy in the global memory. Because performing the addition instructions in a multi-precision addition is very cheap, 48.73ms $\times$ 43% = 21ms would be the true execution time of 8312 parallel *reduction* if all variables are in shared memory. Furthermore, a quantitative understanding of cost to run multi-precision addition with one of its operands in the global memory is obtained: its overhead is close to 0.76 serial version of *reduction* in the *CI-2thread* model (calculated from (48.73ms $\times$ 0.57) / 36.71ms); most of it contributed by the global memory access. Such a quantitative understanding will be critical for us to design the general lazy reduction scheme on GPU. In brief, it can be concluded

that one multi-precision ADD/SUB operation in the global memory space is roughly costs 76% execution time of one *reduction* in the shared memory.

(5) Overall, Table 5.2 shows that the *CI-2thread* model with a serial version of *reduction* illustrates the best performance among three options. And the *CI-2thread* model is used as the low-level parallel computing model in both point multiplication and bilinear pairing implementations.

5.3 Sliding Window–based Point Multiplication

The sliding window-based algorithms [56] are the most widely used implementation methods for point multiplication. Among these sliding window-based algorithms, the algorithm with dynamic window sizes and both positive and negative window values is commonly recognized as the most efficient serial implementation [56], because it needs a minimum number of point addition for one point multiplication. This implementation can be described as follows: The scalar k in a point multiplication instance $[k]$ P is viewed as a stream of binary bits, given a point table which includes pre-calculated $[-L]$ P, $[-L+2]$ P, …, $[L-2]$ P, $[L]$ P (here L is a positive odd integer), a window slides k from its *most significant bit* (MSB) to its *least significant bit* (LSB), then a formula $(2^{\alpha} + a) \times 2^{\beta} = \lambda$ is calculated, where λ = the binary value of the segment of k in the current window, a is a value satisfy $-L \leq a \leq L$, if $a \neq 0$, one point addition/subtraction is triggered if $a > 0$ / $a < 0$, and α and β are the numbers of point doubling before and after this point addition. Such a window sliding process continues until the window goes through all bits of k. The arithmetic operations used in this algorithm include elliptic curve point addition, point subtraction and point doubling operations, which rely on arithmetic operations in the base field F_q.

Although the sliding window-based algorithm with dynamic window sizes is efficient on CPU, it is not GPU-friendly algorithm. For example, assuming k_1 and k_2 are two scalars in two parallel point multiplication instances, and k_2 has a higher hamming weight than k_1. When these two point multiplication instances simultaneously execute in the SIMT architecture, a possible snapshot could be: for the instance that runs $[k_1]$ P, it is executing α times of point doubling, at the same time, for the other instance that runs $[k_2]$ P, a much higher hamming weight of scalar indicates more frequent invocation of the point addition, and therefore this instance may have done the point doubling part and want to run a point addition operation. In SIMT, the latter instance has to wait for the former.

Comparing with sliding window-based algorithms with dynamic window sizes, the variant with a fixed window size is a more SIMT-friendly option for point multiplication. In this variant of point multiplication, no matter the values of k_1 and k_2, the instances using k_1 and k_2 will computes the same number of point doubling followed by a point addition/subtraction. Knowing that the bit length of scalars is selected as 256-bit to achieve the 128-bit security level, if the window size equals N, then the number of point addition is $256/N$ (or $256/N + 1$ if 256 cannot be divided by N). When adopting the variant with a fixed window size, a larger size of the window is preferred since it results in fewer point addition operations.

Assuming that the window size is set as N, the pre-calculated table for point P has 2^N-*2* entries {[2] P, …, [2^N-*1*] P}, and its memory size is (2^N-*2*) $\times$ 64Bytes. Because the pre-calculated table is read only during the point multiplication, it is preferred to utilize the 48KB (per SMX) read-only data cache provided by GK104 GPUs to store it. However, when N becomes too large, the pre-calculated table cannot be fully stored in the read-only data cache, some of its entries need to be stored in the global memory space.

To utilize a large N with few access of global memory, we adopt a large N and store the most frequently visited entries of the pre-calculated table in the read-only data cache, and store the rest entries in the global memory. N = 9, 10, 11 are evaluated in this work. In these three cases, the sizes of pre-calculated table will be slightly greater than 48KB (the size of data cache). When N = 9 / 10 / 11, storing the whole table needs 32 / 64 / 128 KB, and 29 / 26 / 24 point addition is needed to run one point multiplication operation. N = 9 is not a good choice since the data cache is not fully utilized. When N = 11, the size of table becomes too large, so that the majority of the pre-calculated table needs to be stored in the global memory space. Furthermore, comparing with the case N=10, using N=11 can only reduce two 2 point addition operations. As such, N=10 is adopted, and nearly 75% entries of the table is saved in the read-only data cache, which have almost fully occupied the 48KB read-only data cache.

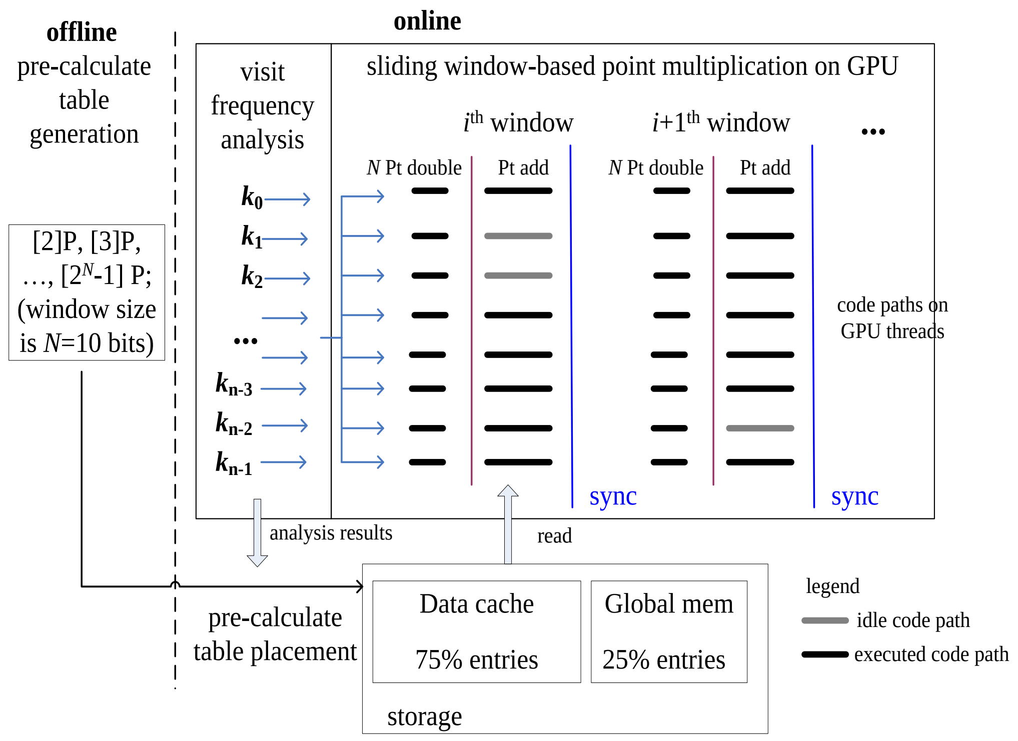
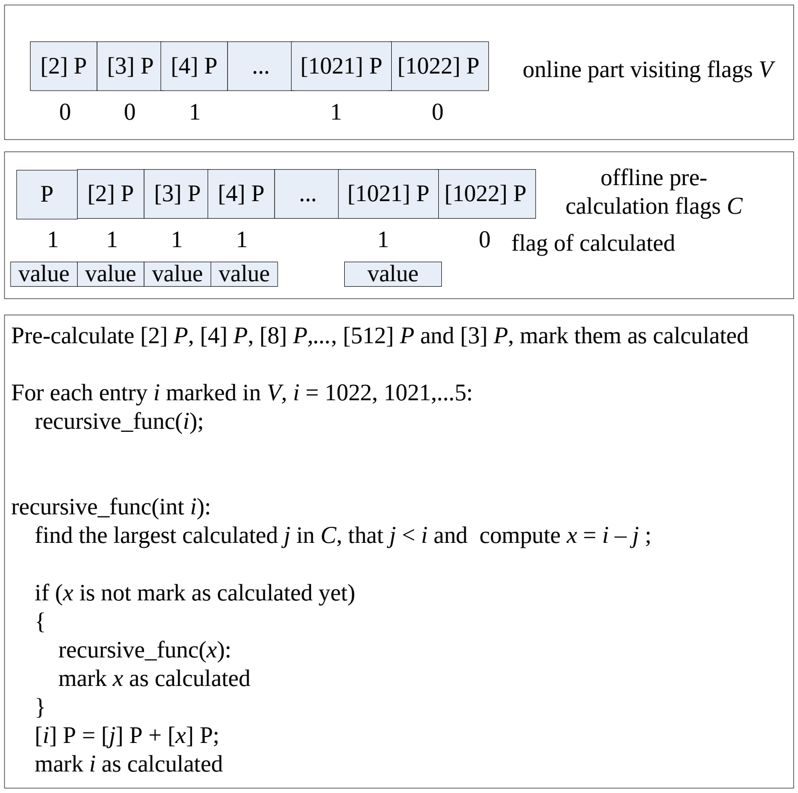


Figure 5.7 GPU-based Point Multiplication (the Same *P* among Instances)

Figure 5.7 illustrates the execution effect of sliding window-based point multiplication on GPU. Because all instances use the same public point *P*, the generation of table {[2] *P*, [3] *P*, … [2^N-1] P} can be constructed before the point multiplication starts. 26 windows slide are needed to go through the 256-bit scalars. In each window (except the last window), 10 point doubling operations are invoked, followed by a point addition if the value in the current window interval of the scalar k_i's bit sequence is non-zero. In sum, the overall cost for online computation part of point multiplication can be predicted. That is, 256 point doubling operations, 26 point addition operations, 26 data cache / global memory read of elliptic curve points, 26 synchronization instructions.



```
Pre-calculate [2] P, [4] P, [8] P,..., [512] P and [3] P, mark them as calculated

For each entry i marked in V, i = 1022, 1021,...5:
    recursive_func(i);


recursive_func(int i):
    find the largest calculated j in C, that j < i and  compute x = i – j ;

    if (x is not mark as calculated yet)
    {
        recursive_func(x):
        mark x as calculated
    }
    [i] P = [j] P + [x] P;
    mark i as calculated
```

Figure 5.8 Online Generation of Pre-calculated Table (Various Points among Instances)

Next, we discuss another utilization case of the parallelized point multiplication (shown in Figure 5.2), which sets the same scalar and different points as the input for each instance. Its (on-line) sliding-window algorithm is skipped since it is similar to the case with different scalars and the same point, which have been presented in last paragraph. For this utilization case, the generation of pre-calculated table is more complicated since each point multiplication instance needs to run a pre-calculated table generation process for its point P_i. Because at most 26 of 1022 entries of the pre-calculated table will be visited by the sliding window algorithm, constructing a complete pre-calculated table {[2] P_i, [3] P_i, … [2^N-1] P_i} for point P_i is over complicated. An more efficient way is only calculating entries that will be visited by the sliding window algorithm. Based on this observation, a pre-calculated table generation algorithm for point P_i is proposed and displayed in Figure 5.8. The vector V flags all entries visited by the sliding window algorithm, and the vector C flags all entries need to be calculated to calculate V. By default, it is assumed as all $[2^n]$ P_i need to be calculated in C. Then, as shown in Figure 5.8, a recursive process is invoked to calculate V.

At the first glance, the recursive processes of generating the pre-calculated table for different points P_i are not suitable for GPU architecture due to their extensive branch divergences. However, the code path of these recursive processes are determined by the scalar of the point multiplication. Because all point multiplication instances are using the same scalar, their recursive processes of generating the pre-calculated tables for P_i have the same code path, and thus fit the SIMT architecture. As such, we implement the GPU-based parallel pre-calculated table generation process for P_i. We will present the experiment result of this part in the sub-chapter 5.5.

5.4 Optimization of Arithmetic in Extension Fields

Comparing with point multiplication, which only needs arithmetic operation in the base field F_q, bilinear pairing needs arithmetic operations in higher extension fields such as F_q^2, F_q^4, F_q^6, F_q^{12}. An arithmetic operation in higher extension fields can be represented as a tower of arithmetic in lower extension fields [87]. Taking the field F_q^2 as an example, an element in F_q^2 can be represented as a polynomial $a+bx$, where a and b are elements in F_q, and x is the root of the irreducible polynomial $x^2 + \beta$. That is, x^2 can be replaced by $-\beta$. By doing so, the arithmetic in F_q^2 becomes the analogy with arithmetic of complex numbers, with imaginary square root of $-\beta$. For example, the multiplication in F_q^2 is in the format as: $(a + xb)\ (c + xd) = (ac - bd) + x\ (bc + ad)$. In this computation, 4 modular multiplications in F_q are called.

The lazy reduction scheme can be applied to the multiplication in F_q^2. First, the computation process of $(a + xb)\ (c + xd)$ can be optimized as $(ac - bd) + x\ [(a+b)(c+d) - ac - bd]$, which reduces the number of modular multiplication in F_q to three. Second, instead of invoking modular multiplication for ac, bd, and $(a+b)(c+d)$, multiplication is invoked. The reduction step is moved after the accumulation of $(ac - bd)$, and $[(a+b)(c+d) - ac - bd]$, so that only two reduction is necessary, one for $(ac - bd)$, the other for the result of $[(a+b)(c+d) - ac - bd]$. Furthermore, a general lazy reduction scheme was first proposed in [3], which can be applied to F_q^2 or higher extension fields such as F_q^6 and F_q^{12}. When the general lazy reduction scheme is applied in F_q^{12}, it can reduce the number of *reduction* in a modular multiplication in F_q^{12} to 12. The experiment result in [3] showed that the general lazy reduction scheme could significantly reduce the computational complexity of modular multiplication and squaring in F_q^{12}, and thus greatly increase the throughput of bilinear pairing on CPU.

On the other hand, the lazy reduction scheme has a side effect. That is, delaying *reduction* operations to higher extension fields means that each variable in lower extension fields has to occupy the doubled memory space before the reduction occurs. To apply the lazy reduction scheme on GPU, the increase of memory space usually means a vast of number of temporary variables need to be stored in the global memory. One design option to reduce the global memory visit is assigning more shared memory to each CI. However, such a design option also decreases the degree of parallelism. An alternative option is keeping the shared memory usage per CI while using more global memory for each CI. With this design option, the degree of parallelism is kept, but more global memory access will occur.

Table 5.3 Performance of 1000 Modular Multiplication in F_q^4

Optimization choices	Execution time	Threads per SMX	Shared mem per CI	Throughput (/sec)
lazy reduct in F_q^2	265.4ms	352	256 bytes	5313×10^3
lazy reduct in F_q^4	301.8ms	352	256 bytes	4662×10^3
prefetch + lazy reduct in F_q^4	304.7ms	352	256 bytes	4617×10^3
lazy reduct in F_q^4	233.7ms	224	320 bytes	3829×10^3
lazy reduct in F_q^4	225.2ms	224	384 bytes	3982×10^3

We first examine the throughput of the modular multiplication in F_q^4 with the general lazy reduction scheme applied in F_q^2 and F_q^4. Table 5.3 lists the performances of

1000 modular multiplication with several different design options. The difference between row 1 and 2 of Table 5.3 is where the lazy reduction is applied. It is found that the latter case returns a worse performance outcome, even though it calls fewer reductions. It shows that the increase of slow global memory access has dominated the benefit of fewer *reduction* operations. Such a result is opposite to the observations of applying lazy reduction on CPUs. We further investigated the software-based pre-fetching scheme [59]. In this scheme, before reading a variable of the current warp from the global memory, variable that will be read by threads in the next warp is pre-fetched to L2 cache. Row 3 illustrates the performance of applying the pre-fetching scheme with the lazy reduction to F_q^4, it shows that no noticeable performance gain was obtained by pre-fetching. Such a result is not surprising since it is commonly known that pre-fetching may not always accelerate the computing process, and sometimes pre-fetching can even trigger some hurtful memory accesses. In this experiment, because the runtime scheduling of warps on SMX is transparent to programmer, the *pre-fetching for next warp* policy without hardware support cannot guarantee to make a positive hit at run time.

Rows 2, 4, and 5 in Table 5.3 illustrate some marginal improvement of execution time when more shared memory is allocated to each CI. The execution gain is only marginal because the memory usage of EAGL is spatially and timely optimized. And therefore, less benefit can be further gained as more shared memory is assigned to each CI. On the other hand, assigned more shared memory for each CI led to significant drop of throughput due to the reduced degree of parallelism.

Based on the results of applying lazy reduction in F_q^2 and F_q^4, it is the time to answer whether it is suitable to apply lazy reduction to field F_q^{12} on GPU. A quick analysis of computational complexity for applying the lazy reduction to F_q^{12} suggested a sharp increase of the global memory access than that in the case of applying the lazy reduction in F_q^4: when applying the lazy reduction to F_q^{12}, the memory size of all temporary variables in a modular multiplication in F_q^{12} doubles, and most computation steps have to either fully reside in the global memory, or frequently invokes data swapping between shared memory and global memory. As a result, on the contemporary GPU architecture, when applying the lazy reduction to F_q^{12}, its overhead of global memory access was deemed to be too high to make this technique useful. In summary, despite applying lazy reduction in higher extension fields F_q^4 or F_q^{12} has further reduced the computational complexities, applying lazy reduction in extension fields F_q^2, which triggers much fewer global memory accesses, is best suited for contemporary GPU architecture.

As discussed in subchapter 5.2, although the RNS and conventional Montgomery system only determine the implementation of low-level multi-precision multiplication and reduction, the efficiency of these number system is closely related to which extension field the lazy reduction policy is used. Assuming that the same lazy reduction policy applied, it is concluded that our conventional Montgomery-based *CI-2thread model* needs slightly less computational complexity to run a modular multiplication in F_q^{12} than the RNS-based computing models in [2][19]. We assert this conclusion as follows. First, when comparing the multi-precision multiplication under RNS with 2 threads and that in our *CI-2thread* model, as aforementioned, they have similar computational complexity – $n^2/2$ MUL plus some synchronization cost. Second, in terms of the computational complexity of the *reduction* function, [19] showed that a serial *reduction* function in RNS costs $2n^2+3n$ MUL, where $2n^2$ was spent in two matrix operations in the Base Extension (BE). It is known that the workload of a matrix operation can be equally balanced to two threads, and thus the cost of a reduction in their RNS-based model would be n^2+3n MUL when two threads are used. On the other hand, the reduction in the *CI-2thread* model costs n^2+n MUL. Therefore, the computational complexity of one reduction function in the RNS-based model is slightly higher than that in the *CI-2thread* model. The last factor is the computational complexity reduction by applying the lazy reduction technique. A modular multiplication in F_q^{12} can be considered as multiple addition sequences of modular multiplication, each sequence is in a format as ($a \times b + c \times d + e \times f + \ldots$) mod q. When the lazy reduction is applied to higher extension field, the average length of these sequences becomes larger, and the number of reduction becomes less. In other words, the lazy reduction technique determines the numbers of multi-precision multiplication and reduction operations, and the selection of a number system determines the computational cost of the multi-precision multiplication ($a \times b$, $c \times d$, or $e \times f$) and the reduction (implied in "mod q"). As such, the RNS-based computing model and our *CI-2thread* model get the same decrease of computational cost from the lazy reduction technique.

Other state-of-the-art optimization techniques for arithmetic in extension fields F_q^2, F_q^4, F_q^{12} applied in our work include: (1) type D sextic twist [89] of the Barreto-

Naehrig (BN) curve [10] over F_q^2 for input points; (2) a low hamming weight BN curve [24]; (3) denominator elimination and lazy reduction in [87] and final exponentiation optimization for BN curves in [24] and [88]; (4) Karatsuba multiplication for multiplication in F_q^{12}, Chung-Hasan SQR_3 [21] for squaring in F_q^{12}; (5) for unitary elements of F_q^{12} generated during the final exponentiation, fast squaring of elements in F_q^4 [92] and Granger-Scott fast squaring [35] in F_q^{12}. We acknowledge that the new implementation-friendly curve family [78] (a subclass of BN curve) for the *optimal Ate* pairing [94] is also important, but it is not considered in this work because of the extensive additional efforts to rewrite its lazy reduction-related parts for the cross verification of EAGL.

Next, we analyze the trade-off between the pipelining effect on GPU and the overhead for memory access by adjusting the shared memory (smem) usage per CI. Less shared memory per CI means more threads per block and thus better pipeline utilization, while it also means more slow memory visits. Without a clear guidance for an optimal configuration rule, we gradually tune the shared memory usage per CI in an ascend order to find the peak point for the throughput of bilinear pairing. By summarizing the memory usage of point and field arithmetic functions in the bilinear pairing, it is found that the most frequently accessed cache could be reduced up to four elements in F_q^2 per CI, quantitatively 256 bytes, while the overhead for chaff spacers and shared memory bank alignment is negligible. Therefore, we select the 256-byte as the pivot point, tune the shared memory (smem) usage per CI from 192-byte to 384-byte, stepped by one element in F_q^2 (64 bytes), and observe the fluctuation of throughput.

Table 5.4 Throughput Fluctuation as Shared Memory Usage per CI Changes

smem usage per CI (bytes)	192	256	320	320 (P-*reduct*)	384
gpu thread per block	480	352	224	224	224
smem utilization rate	93.6%	91.7%	79.1%	79.1%	87.5%
throughput (pairing/sec)	2926	3350.9	2077	2564.6	2861

Throughputs of bilinear pairing with different configuration parameters are listed in columns 2-6 of Table 5.4. It is noticeable that when the smem usage per CI equals 320-byte, the parallel version of reduction (P-*reduct*) can move the T in the of $T+mq$ step from the global memory space (in Table 5.2) from the shared memory space. The result of this version is shown in the 5th column of Table 5.4. It is found that the peak throughput occurs when each CI caches four elements of field F_q^2 in the shared memory space (the 3rd column of Table 5.4). According to Table 5.4, when the shared memory usage per CI equals 320-byte, the shared memory usage utilization rate is fairly poor due to the limitation of assigning complete warps of threads to the SMXs. When this parameter grows to 384-byte, the negative effect of worse pipeline utilization begins to negate the benefits of more fast memory hits. In sum, the lazy reduction in F_q^2 is adopted, and 256-byte as the shared memory (smem) usage per CI in EAGL.

5.5 Experiment Results of Point Multiplication

In this sub-chapter, we first report the experiment result of our GPU-based point multiplication with the same point P and different scalars in each instance. The GPU platform is a GTX-680 card. Parameters of the degree of parallelism are 8 SMX $\times$ 384 threads, equivalent to 8×192=1536 concurrent instances. 1536 256-bit random generated scalars are inputs of this experiment. As a result, the execution time, including pre-processing of scalars on CPU and point multiplication on GPU, is 32.68ms, equivalent to 47000 point multiplication per second, where the CPU-based pre-processing process for analyzing the visit frequency of entries of the pre-calculated table takes less than 1ms.

Table 5.5 Comparison of Throughputs of Point Multiplication among GPU Implementations

Implementations	Key size	Throughput (/sec)	Device	Device peak GFLOPS
[12]	224 bit	5895	GTX 285	1062
[2]	224 bit	9827	GTX 295	1788
EAGL	256 bit	47000	GTX 680	3090

Table 5.5 compares the throughputs of point multiplication between our work and two recent GPU-based implementations. Before comparing the throughput enhancement of EAGL with existing benchmarks which obtained on earlier generations of GPUs, throughputs are normalize by the difference of peak device GFLOPS. Although the difference on peak GFLOPS can roughly reflect the difference of their processing strength, we understand that it is not a perfect method to measure the processing strength since the architectural changes across generations of GPU are usually coupled with different memory bandwidth and frequency, and new features to facilitate off-chip or on-chip memory access. After normalizing the throughputs by the difference of peak device GFLOPS, it is shown that EAGL has roughly 2.76 times

higher throughput than that in [2]. Noting that the peak GFLOPS scale up between GTX-680 and GTX-295 is 1.72 folds, which is smaller than the throughput grows between EAGL and [2], it indicates that the *CI-2thread* model in EAGL efficiently utilize the on-chip resources of GK-104 architecture.

Next, EAGL is compared with several recent CPU-based point multiplication implementations in Table 5.6. For more benchmarks of point multiplication on CPU or other platforms such as FPGA, PS3, we refer readers to [2]. It is found that EAGL can provide 2.1 times higher throughput than that in [63]. Furthermore, it is noticeable that EAGL and [69] have similar code paths; and EAGL and [63] are based on different types of elliptic curves, where EAGL is based on a standardized elliptic curve, and [63] a twisted curve over $F_q{}^2$.

Table 5.6 CPU vs. GPU-based Point Multiplication Implementations

Implementations	Key size	Elliptic curve	Throughput (/sec)	Device
Optimized GLS method [63]	256 bit	twisted Edward curves	22472	2.6GHz AMD Opteron
MIRACL [69]	224 bit	standardized	14509	3.0GHz AMD Phenom II
EAGL	256 bit	standardized	47000	GTX 680

In the end, we present the experiment result of point multiplication with different points and the same scalar. Because this research is the first work to discuss this type of parallel point multiplication, no existing experiment result are found for comparison. In this experiment, 1000 random generated scalars are tested, and the average number of point addition/doubling to generate the pre-calculated table is 39.5, much less than 1022 point addition/doubling needed by the MIRACL's pre-calculated table generation method. On GTX-680, the execution time of the pre-calculated table generation (in Figure 5.8) is 8.52ms. The rest part of point multiplication with different points and the same scalar has a similar computational cost to point multiplication with different scalars and the same point. As a result, the point multiplication with different points and the same scalar needs roughly 25% execution time than the point multiplication with the same point and different scalars.

5.6 Experiment Results of Bilinear Pairing

In this subchapter, we present the experiment result of bilinear pairing and some further analysis based on the result. First, EAGL is compared with GPU-based implementations in the literature [54][98]. As illustrated in Table 5.7. The configuration parameter of the degree of parallelism is 8 SMX $\times$ 352 threads ($8\times176=1408$ instances). On a GTX-680, the execution time is 420.19ms for 1408 instances, equivalent to 3350.9 pairings/sec. In comparison, the throughput of η_T pairing under 128-bit AES security strength is 254 pairings/sec on one Tesla C2050 card [54]. The peak GFLOPS of GTX-680 is roughly three times larger than M2050/C2050, after taking into account the difference of computational strength, the throughput of EAGL is roughly 4.4 times greater than that in [54]. Furthermore, this performance comparison does not even count in the factor that as the FE step in [54] was not parallelized. According to such a comparison, it could be concluded that following the traditional parallel computing mapping policy, the *CI-2thread* computing model fits the SIMT architecture better than the bit-slice model, which greatly complicates the variable manipulation in multi-precision integer arithmetic operations under SIMT architecture.

In the end of Table 5.7, we also include the results of [98] which achieved 23.8ms/pairing for the composite-order pairing as a reference. However, making quantified comparison between EAGL and that in [98] is difficult because of the significant difference on computational complexity between the prime-order and composite-order pairing.

Table 5.7 Comparison of Execution Time Among GPU Implementations

Implementations	Algorithm	Curve Type	Security	Exec time (ms)	Device
EAGL	*R-ate*, prime order	ordinary	128-bit AES	0.298	GTX-680
[54]	η_T, prime order	ordinary	128-bit AES	3.94	C2050
[98]	*Tate*, composite order	super-singular	80-bit AES	23.8	M2050

Next, EAGL is compared with existing CPU-based pairing solutions [3][14][70], where all the performance results were based on a single CPU core by their authors. Our objective is two-fold: First, we would like to evaluate the performance of EAGL by comparing with the benchmarks on contemporary commodity CPUs. Furthermore, our purpose is obtaining some in-depth understanding on the bottlenecks of different system architectures. Lack of actual experimental results, a perfect acceleration model is adopted for CPU cases where the speed up is proportional to the number of available processor cores. The performance figures of studied cases are summarized in Table 5.8. One can see that EAGL on one GTX-680 board has about half of the throughput that could be achieved in [3] based on the perfect acceleration model for multi-core CPUs. Comparing with the acceleration rate of EAGL on point multiplication, we found a much worse acceleration rate on bilinear pairing on the contemporary GPU architecture.

Table 5.8 Comparison of Throughput, EAGL vs. CPU-based Solutions

Implementations	Algorithm	Device	Core clock	Throughput
EAGL	*R-ate* pairing	GTX-680	1006MHz	3350.9
[70]	*Ate* pairing	Intel Q6600	2.4GHz	4×669 (est.)
[14]	*Ate* pairing	Intel i7 860	2.8GHz	4×1202 (est.)
[3]	*Ate* pairing	Intel i5 760	2.8GHz	4×1661 (est.)

However, GPU-based bilinear pairing solution supported by EAGL is not meaningless. It is noted that the CPU host thread that runs on CPU has negligible performance cost. As such, the CPU host processor(s) can utilize GPU as a pairing co-processor, while the host processor(s) can run other business logic such as database management, high-throughput networking or file I/O. For an application requiring small to moderate throughput of bilinear pairing, EAGL can be a supplement for CPU-based solutions. Furthermore, EAGL is a scalable solution to provide excessive throughput of bilinear pairing.

5.7 Analysis of System Bottleneck

Computing performance is affected by computational complexity of the algorithm, programming techniques, and the underlying architectures. All three factors need to be seamlessly integrated to achieve top performance. When the acceleration rates of EAGL on point multiplication and bilinear pairing are cross compared, it is found that EAGL shows different performance relationships when it compares with CPU-based implementations. When comparing the best CPU-based benchmarks reported in [63] (for point multiplication) and [3] (for bilinear pairing), EAGL's throughput enhancement rate is around 2.1 folds for point multiplication, but roughly 50% for bilinear pairing. It is interesting to investigate reasons of such a phenomenon.

The first question is whether the *CI-2thread* computing model, which is based on the conventional Montgomery number system, fits contemporary GPU architecture. The compilation result of CUDA compiler shows that almost all shared memory are occupied under current parallelism configuration, and the register count per thread reaches 63, which is the upper-bound on GK104 GPUs. As such, the *CI-2thread* model has fully utilized the on-chip resources of GTX-680.

Then, the next question is whether the *CI-2thread* model is better than the RNS-based parallel computing model. Because no previous studies implemented RNS-based bilinear pairing on GPU platforms, to evaluate whether the *CI-2thread* model is better than the RNS-based parallel computing model, the comparison target has to switched to point multiplication since both algorithms share the same low-level computing model. It is reported in [2] that 9827 224-bit point multiplication/sec can be achieved on a GTX-

295 (1788 peak GLOPS). On the other hand, EAGL on a GTX-680 offers 47000 256-bit point multiplication/sec on a GTX-680 (3090 peak GLOPS). EAGL's performance on point multiplication is 2.76 times higher than [2] after normalizing the difference of peak GFLOPS, even at higher security strength. Moreover, the RNS-based computing model needs extra memory space to save the matrixes in the BE step in the *reduction* function. Although it is very hard to compare the computational strength utilization rate across different generations of GPUs, a higher growth of throughput (2.76) than that of GFLOPS (1.72) at least proves that the *CI-2thread* model is not worse than RNS on contemporary GPUs.

A comparison of sizes of memory usage per CI between point multiplication and bilinear pairing gives us some clues on why EAGL obtains different performance acceleration effect on point multiplication and bilinear pairing, vs. state-of-the-art CPU-based solutions. The change of memory usage sizes is illustrated in Figure 5.9. It is shown that, in both point multiplication and bilinear pairing, the sizes of fast cache (in the shared memory space) are stable. However, in bilinear pairing, the size of slow cache (in the global memory space) fluctuates, and only in the line function calculation step, its size of slow cache drops under that in point multiplication. Figure 5.9 clearly suggests that one bilinear pairing instance needs much more global memory resources than one point multiplication instance.

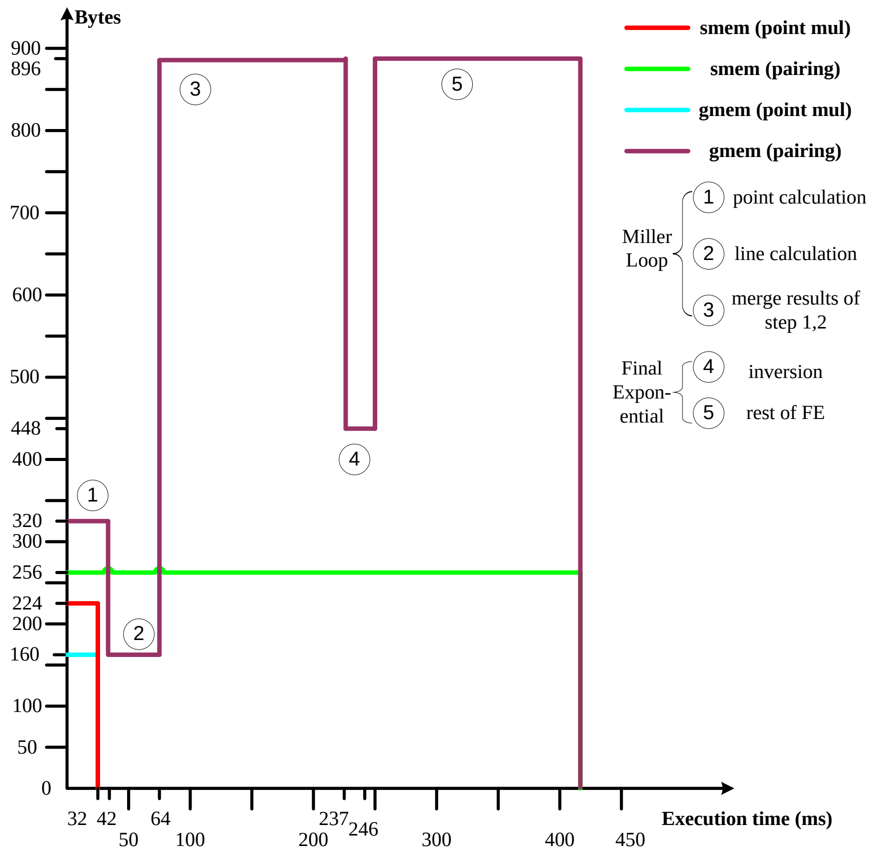


Figure 5.9 Shared/Global Memory (s/g mem) Size of Memory Usage per CI in EAGL's Point Multiplication of and Bilinear Pairing

To gain some in-depth understanding about the bottleneck, the computation steps of bilinear pairing are broken down. One pairing computation consists of 11938 multi-precision multiplication, 8312 reduction, plus over 20k inexpensive multi-precision addition/subtraction operations. Although EAGL supposes most low-level multi-precision arithmetic operations occurs only in the shared memory and registers, computations in higher extension fields have to swap variables between available shared memory and global memory. As such, some variable swapping occurs as prior steps or post steps for multi-precision arithmetic operations. If assuming data swapping between shared memory and global memory has negligible cost, then the computation cost of a bilinear pairing operation is equivalent to the sum of computing 11938 multi-precision multiplication, 8312 reduction, plus 20k multi-precision addition/subtraction operations. In those arithmetic operations, the multi-precision addition/subtraction operations take less than less than 15ms if all operations occur in the shared memory space. According to the result shown in Table 5.2, running 11938 multiplication plus 8312 reduction operations in the shared memory space and registers takes 76ms. The overall estimated execution time is close to 91ms, which is much less than the actual execution time (420ms).

Even though the hardware level profiling tools is not available for precise measurement, it is asserted that a large proportion of execution time (420ms) is spent in variables swapping between shared memory and global memory spaces as follows: further breakdown of execution time shows that each powering of arbitrary x in F_q^{12} in the Final Exponential (FE) step takes 47ms. On the other hand, NVIDIA's profiling tool *nvprof* shows that one concurrent global memory copy of elements in F_q^{12} takes 35μs. Because a global memory access usually takes hundreds of cycles, and GTX-680 has 1006MHz processor unit, such an execution time for one concurrent global memory copy of elements in F_q^{12} is reasonable. Furthermore, *nvprof* shows that one powering of x in F_q^{12} triggers nearly 500 times more global memory hits than a copy in F_q^{12}. Such a ratio indicates that global memory hits in one powering of arbitrary x in F_q^{12} takes almost 17ms, equivalent to 35% of the execution time. And this estimated percentage has not counted in extra synchronization and branch divergent cost associated with global memory hits, which occurs if one global memory hit is embedded by an if-else statement. One example is the borrow bit calculation in the multi-precision subtraction operation. In sum, it can be concluded that a major proportion of execution time is spent in swapping variable between the shared memory space and the global memory space.

CHAPTER VI

CONCLUSION AND FUTURE WORK

This dissertation explores design issues for parallelization of SFE-based secure *Edit distance* (ED) and *Smith-Waterman* (SW) algorithms and the ECC-based *Private Set Intersection* (PSI) and *Secret Handshake* (SH) protocols on the Kepler GPU architecture.

A parallel computing model for SFE-based ED and SW algorithms are proposed. It includes a high-throughput GPU-based gate (de-)garbler, a *static* memory management strategy, pipelined design, and general GPU resource mapping policies for DP problems which is parallelized based on the *wavefront* parallel computing pattern. This dissertation shows that, with very little waste of processing cycles or memory space, the Kepler GPU architecture can be fully utilized to run billions of garbled gates to implement SFE-based ED and SW algorithms.

Second, this dissertation shows that the conventional Montgomery-based number system is friendlier to the Kepler GPU architecture than the RNS–based Montgomery number system is, based on the comparison of throughputs in this work vs. those reported in [2]. Furthermore, on Kepler GPU architecture, the *lazy reduction* in the quadratic extension field obtains better throughput results than that in the quad or the twelfth extension field, which is contrary to the results reported on CPU architectures [3]. The Elliptic curve Arithmetic GPU Library (EAGL) is implemented, which can run 3350.9 *R-ate* (bilinear) pairing/sec, or 47000 point multiplication/sec at the 128-bit security level. Although this dissertation does not study other bilinear-pairing-based secure protocols such as key agreement [29][97], identity-based encryption [18][95], identity-based signatures [71][90], short signature schemes [15][47][64], EAGL can be applied in the construction of these protocols in a straight-forward fashion.

Third, this dissertation illustrates that simple ECC-based computations, such as point multiplication or field arithmetic in the quadratic extension field can be effectively supported by the Kepler GPU architecture. It is identified that lacking of advanced memory management functions in the contemporary GPU architecture impose some limitations on bilinear pairing operations. Substantial performance gain can be expected when the on-chip memory size and/or more advanced memory prefetching mechanisms are supported in future generations of GPUs.

With respect to the modular structure and the tool chain for automation of SFE-based computing problems, three new challenges shall be solved for the future generation of the GPU-based parallel computing model. Firstly, unlike [65] [40] and [43], where wire label are generated when a gate is garbled, the computing model proposed in this dissertation pre-assigned wire labels before a task is dispatched to GPU for garbling. As a result, the parallel computing model proposed in this dissertation needs a fine-grained categorization of wire labels (L_o, L_G and L_i), while [65] [40] and [43] do not. A new description language needs to be invented, so that the tool chain can extract wire label categorization information for the SHDL execution engine. Second, because the parallel computing model proposed in this dissertation needs to utilize the maximum usage for each type of wire label, an offline parser needs to be invented to assess maximum usage of wire labels, encrypted results for each GC-slot of the SFE-based DP matrix. The offline parser should accept arbitrary sizes of the DP matrix. Third, Because SHDL cannot cover the inter-dependency specification among garbled circuit, and among GC-slots, a new specification language needs to be design as a supplement of SHDL. This new description language should not only be able to describe inter-connections among GCs, but also describe wire label types of these inter-connections to facilitate the pre-assigning at runtime. Then, implementing a comprehensive tool chain becomes practical.

With GPU being a major branch of parallel architectures to support massively fine-grained parallelism, how to match computing needs of privacy-preserving protocols with the GPU architecture is an open research challenge for now, and the future. To understand what revisions are needed for EAGL when expanding it to future generation GPUs, the specification changes between Fermi (an elder generation of Kepler) and Kepler are studied, and three major library factors are empirically identified as follows:

The first factor is revising the insertion offset of the chaff spacer in the collision-free shared memory access model for arithmetic in the base field. Due to the change of bank width (from 32-bit to 64-bit as Fermi evolves to Kepler), one 256-bit operand is placed in 4 consecutive banks, instead of 8. A chaffer with the same width of the bank width needs to be inserted before the first complete operand in a tier. When expanding EAGL to next generation GPUs, revising the insertion offset is necessary when the width or the number of banks is changed. The second factor is the degree of parallelism for a peak benchmark, especially in the implementation of bilinear pairing. Because

EAGL relies on a heuristic, which gradually increases the size of fast memory per instance, to find out the peak performance in the trade-off between memory access speed and the degree of parallelism, expanding to next generation GPUs needs to repeat such a heuristic to determine the new peak performance and associated fast memory usage per instance. Since EAGL is locality aware for storage of temporary variables in low level arithmetic functions, only the top level function which specifies fast memory usage per instance needs to be modified. The last most important factor is the efficiency of lazy reduction may change due to the new global memory accessing speed on next generation GPUs. It is possible that lazy reduction in F_q^4 or F_q^{12} will be more effective. EAGL has implemented lazy reduction in F_q^4 and thus only lazy reduction in F_q^{12} needs to be done if lazy reduction in F_q^4 shows positive acceleration effect.

REFERENCES

[1].O. Ahmadi, D. Hankerson, and A. Menezes, “Software Implementation of arithmetic in GF(3^m)”. In WAIFI’07, 2007.

[2].S. Antão, J.C. Bajard, L. Sousa. “RNS-based Elliptic Curve Point Multiplication for Massive Parallel Architectures”. The Computer Journal 2011, vol. 55 issue 5, pp 629-647, 2012.

[3].D.F. Aranha, K. Karabina, P. Longa, C.H. Gebotys, J. Lopez, “Faster Explicit Formulas for Computing Pairing over Ordinary Curves”. In Eurocrypt 2011, pp 48-68, 2011.

[4].C. Arene, T. Lange, M. Naehrig, C. Ritzenthaler, “Faster Computation of the Tate Pairing”. Journal of Number Theory, vol.131, issue 5, pp 842-857, 2011.

[5].K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. “The Landscape of Parallel Computing Research: A View from Berkeley”. Tech report UCB/EECS-2006-183, 2006.

[6].G. Ateniese and M. Blanton, “Secret Handshakes with Dynamic and Fuzzy Matching,” In NDSS’07, 2007.

[7].R. Babich, M.A. Clark, B. Joo, G. Shi, R.C. Brower and S. Gottlieb. “Scaling Lattice QCD beyond 100 GPUs”. In SC-2011, 2011.

[8].J.C. Bajard, L.S. Didier, and P. Kornerup. “An RNS Montgomery Modular Multiplication Algorithm”. Computers, IEEE Transaction on 47(7):766-776, 1998.

[9].D. Balfanz, G. Durfee, N. Shankar, D. Smetters, J. Staddon, and H. Wong, "Secret Handshakes from Pairing-based Key Agreements," In S&P'03, 2003.

[10].P.S.L.M. Barreto and M. Naehrig, "Pairing-Friendly Elliptic Curves of Prime Order". In SAC 2005, pp 319-331, 2005.

[11].P.S.L.M. Barreto, S.Galbraith, C.O. hEigeartaigh and M. Scott. "Efficient Pairing Computation on Supersingular Abelian Varieties". Designs, Codes and Cryptography, vol.42, Num.3, pp 239-271, 2007.

[12].D. J. Bernstein, T.R. Chen, C.M. Cheng, T. Lange, and B.Y. Yang, "ECM on Graphics Cards". In Eurocrypt 2009, pp 483-501, 2009.

[13].D. J. Bernstein and T. Lange. "Faster Addition and Doubling on Elliptic Curves". In Asiacrypt'07, 2007.

[14].J.L. Beuchat, J.E.G. Diaz, S. Mitsunari, E. Okamoto, F. Rodriguez-Henriquez, and T. Teruya "High-Speed Software Implementation of the Optimal Ate Pairing over Barreto-Naehrig Curves". In Pairing 2010, pp 21-39. 2010.

[15].D. Boneh, B. Lynn, H. Shacham, "Short Signatures from the Weil Pairing". In Asiacrypt 2001, pp 514-532, 2001.

[16].J. Boxall, N.E. Mrabet, F. Laguillaumie, D.P. Le. "A Variant of Millers Formula and Algorithm". In Pairing'10, 2010.

[17].C. Cachin. "Efficient Private Bidding and Auctions with Oblivious Third Party". In CCS-99, 1999.

[18].A.D. Caro, V. Iovino, G. Persiano, "Fully Secure Anonymous HIBE and Secret-Key Anonymous IBE with Short Ciphertexts". In Pairing 2010, pp 347-366, 2010.

[19].R.C.C. Cheung, S. Duquesne, J. Fan, N. Guillermin. “FPGA Implementation of Pairings using Residue Number System and Lazy Reduction”. In CHES’11, 2011.

[20].B. Chor, O. Goldreich, E. Kushilevtiz, M. Sudan. “Private Information Retrieval”. In 36th IEEE Conference on the Foundations of Computer Science (FOCS), pp 41-50, 1997.

[21].J. Chung and M.A. Hasan. “Asymmetric squaring formulae”. In ARITH 2007, pp 113-122, 2007.

[22].Crypto++5.6.0 benchmarks on Intel Pentium 4 CPU. Available at http://www.cryptopp.com/benchmarks-p4.html, last modified in 2009.

[23].C. Costello, T. Lange, M. Naehrig. “Faster Pairing Computations on Curves with High-Degree Twists”. In PKC’10, 2010.

[24].A. J. Devegili, M. Scott, and R. Dahab, “Implementing Cryptographic Pairings over Barreto-Naehrig Curves”. In Pairing 2007, pp 197-207, 2007.

[25].W. Diffie, M. Hellman. “New directions in cryptography”. IEEE Transactions on Information Theory 22 (6): pp 644–654. 1976.

[26].P. Duan. “Oblivious Handshakes and Computing of Shared Secrets: Pairwise Privacy-preserving Protocols for Internet Applications”. Ph.D. thesis available at: http://repository.tamu.edu/bitstream/handle/1969.1/ETD-TAMU-2011-05-9445/DUAN-DISSERTATION.pdf, 2011.

[27].I. Duursma, H.S. Lee, “Tate Pairing Implementation for Hyperelliptic Curves $y^2 = x^p - x + d$”. In Asiacrypt’03, 2003.

[28].Z. Erkin, M. Franz. J. Guajardo, S. Katzenbeisser, I. Lagendijk, and T. Toft. “Privacy-preserving Face Recognition”. In PET 2009.

[29].D. Fiore, R. Gennaro, and N.P. Smart, “Constructing Certificateless Encryption and ID-Based Encryption from ID-Based Key Agreement”. In Pairing 2010, pp 167-186, 2010.

[30].T.K. Frederiksen, J.B. Nielsen. “Fast and Malicious Secure Two-Party Computation Using the GPU”. http://eprint.iacr.org/2013/046.pdf, 2013.

[31].M. Freedman, K. Nissim, and B. Pinkas, “Efficient Private Matching and Set Intersection”, In Eurocrypto’04, 2004.

[32].D. Freeman, M. Scott, and E. Teske, “A Taxonomy of Pairing-friendly Elliptic Curves”. Journal of Cryptology, vol. 23, pp 224-280, 2010.

[33].C. Gentry. “A Fully Homomorphic Encryption Scheme”. Ph.D. dissertation. http://cs.au.dk/~stm/local-cache/gentry-thesis.pdf, 2009.

[34].R. Granger, F. Hess, R. Oyono, N. Theriault and F. Vercauteren, “Ate pairing on hyperellitpic curves”. In Eurocrypt’07, 2007.

[35].R. Granger, M, Scott. "Faster Squaring in the Cyclotomic Subgroup of Sixth Degree Extensions". In PKC 2010, pp. 209-223, 2010.

[36].GTX-680 white papers. Available at http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf, 2012.

[37].J. Guilford, K. Yap, V. Gopal. “Fast SHA-256 Implementations on Intel Architecture Processors”. Available at http://download.intel.com/embedded/processor/whitepaper/327457.pdf, 2012.

[38].N. Gura, A. Patel, A. Wander, H. Eberle, S. C. Shantz. “Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs”. In CHES’04, 2004.

[39].C. Hazay and Y. Lindell, “Efficient Protocols for Set Intersection and Pattern Matching with Security against Malicious and Covert Adversaries”, In TCC’08, 2008.

[40].W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. “TASTY: Tool for Automating Secure Two-Party Computations”. In CCS 2010.

[41].S. Henikoff and J.G. Henikoff. “Amino Acid Substitution Matrices from Protein Blocks”. In the National Academy of Sciences of the United States of America, 1992.

[42].F. Hess, N.P. Smart and F. Vercauteren, “The Eta Pairing Revisited”, IEEE Trans. on Inform Theory, vol.52, pp 4595-4602, 2006.

[43].Y. Huang, D. Evans, J. Katz, L. Malka. “Faster Secure Two-Party Computation Using Garbled Circuits”. In 20th USENIX Security Symposium, 2011.

[44].Y. Huang, L. Malka, D. Evans, and J. Katz. “Efficient Privacy-preserving Biometric Identification”. In NDSS 2011.

[45].Y. Huang, D. Evans, J. Katz. “Private Set Intersection: Are Garbled Circuits Better than Custom Protocols?”. In NDSS 2012.

[46].Y. Huang, J. Katz, D. Evans. “Quid Pro Quo-tocols: Strengthening Semi-Honest Protocols with Dual Execution”, In S&P 2012.

[47].Q. Huang, D.S. Wong, W. Susilo, “A New Construction of Designated Confirmer Signature and Its Application to Optimistic Fair Exchange”. In Pairing 2010, pp 41-61, 2010.

[48].Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. “Extending Oblivious Transfers Efficiently”. In Advances in Cryptology – Crytpo, 2003.

[49].K. Jang, S.Han, S. Han, S.Moon, K.S. Park. “SSLShader: Cheap SSL Acceleration with Commodity Processors”. In NSDI 2011.

[50].S. Jarecki and X. Liu, “Unlinkable Secret Handshakes and Key-private Group Key Management Schemes,” In Applied Cryptography and Network Security, 2007.

[51].S. Jarecki and X. Liu, “Efficient Oblivious Pseudorandom Function with Applications to Adaptive OT and Secure Computation of Set Intersection”, In TCC’09, 2009.

[52].S. Jha, L. Kruger, and V. Shmatikov. “Towards Practical Privacy for Genomic Computation”. In S&P 2008.

[53].G. S. Kaminsky, M. Freedman, M. J. Karp, B. Mazieres, and H. Yu. “RE: Reliable Email”, In NSDI’06, 2006.

[54].Y. Katoh, Y.J. Huang, C.M. Cheng, T. Takagi, "Efficient Implementation of the eta Pairing on GPU". Cryptology ePrint Archive, http://eprint.iacr.org/2011/540.pdf, 2011.

[55].L. Kissner and D. Song, “Privacy-preserving Set Operations”, In Crypto’05, 2005.

[56].N. Koblitz, “Elliptic curve cryptosystems”. Mathematics of Computation, Vol.48, No.5, pp 203-209, 1987.

[57].V. Kolesnikov and T. Schneider. “Improved Garbled Circuit: Free XOR Gates and Applications”. In ICALP 2008.

[58].B. Kreuter, A. Shelat, C.h. Shen, “Billion-Gate Secure Computation with Malicious Adversaries”, In 21th USENIX Security Symposium,2012.

[59].J.Lee, N.B. Lakshminarayana, H. Kim, R. Vuduc. “Many-Thread Aware Prefetching Mechanisms for GPGPU Applications”. In MICRO 2010, pp 213-224, 2010.

[60].E.J. Lee, H.S. Lee and C.M. Park, “Efficient and Generalized Pairing Computation on Abelian Varieties”. IEEE Transactions on Information Theory, Vol.55, Issue.4, pp 1793-1803, 2009.

[61].Levenshtein VI. "Binary codes capable of correcting deletions, insertions, and reversals". Soviet Physics Doklady 10, 1966.

[62].Y. Liu, et al. “CUDASW+2.0: Enhanced Smith-Waterman Protein Database Search on CUDA-enabled GPUs based SIMT and Virtualized SIMD Abstractions”. BMC Research Notes 3(1) 93, 2010.

[63].P. Longa and C. Gebotys. “Analysis of Efficient Techniques for Fast Elliptic Curve Cryptography on x86-64 based Processors”. IACR Cryptology ePrint Archive, 335, 1–34, 2010.

[64].S. Lu, R. Ostrovsky, A. Sahai, “Perfect Non-interactive Zero Knowledge for NP”. In Eurocrypt’06, pp 339-358, 2006.

[65].D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. “Fairplay – a Secure Two-party Computation System”. In 13th USENIX Security Symposium, vol. 13, pp 20-20, 2004.

[66].S. Matsuda, N. Kanayama, F. Hess, E. Okamoto, “Optimized Versions of the Ate and Twisted Ate Pairings”. In IMACC’07, 2007.

[67].V. Miller, “Use of elliptic curves in cryptography.” In Crypto’85, 1986.

[68].P. L. Montgomery, “Modular multiplication without trial division”. Mathematics of Computation 44 (1985), pp 519-521, 1985.

[69].Multi-precision Integer and Rational Arithmetic C/C++ Library, MIRACL. Available at http://www.certivox.com/miracl/, 2013.

[70].M. Naehrig, R. Niederhagen, and P. Schwabe, “New Software Speed Records for Cryptographic Pairings”. In Latincrypt 2010, pp 109-123, 2010.

[71].T. Nakanishi, Y. Hira, and N. Funabiki, “Forward-Secure Group Signatures from Pairings”. In Paring 2009, pp 171-186, 2009.

[72].M. Naor, B. Pinkas, and R. Sumner, “Privacy-preserving Auctions and Mechanism Design”,In ACM Conference on Electronic Commerce,1999.

[73].National Institute of Standards and Technology, “Secure Hash Standard”, FIPS 186, US Department of Commerce, Nov. 2001.

[74].National Institute of Standards and Technology, “Advanced Encryption Standard”, FIPS 197, US Department of Commerce, Jan. 1992.

[75].National Security Agency. “The Case for Elliptic Curve Cryptography”. Available at http://www.nsa.gov/business/programs/elliptic_curve.shtml. 2009.

[76].M. Osadchy, B. Pinkas, A. Jarrous, and B. Moskovich. “SCiFI: a System for Secure Face Identification”. In S&P 2010.

[77].J. Pecarina, S. Pu, J.C. Liu, “SAPPHIRE: Anonymity for Enhanced Control and Private Collaboration in Healthcare Clouds”. In CloundCom 2012, pp 99-106, 2012.

[78].G.C.C.F. Pereira, M.A. Jr. Simpl ́cio, M. Naehrig, P.S.L.M. Barreto, “A Family of Implementation-Friendly BN Elliptic Curves”. Journal of Systems and Software, vol.84, issue 8, pp 1319-1326, 2011.

[79].B. Pinkas, T. Schneider, N. Smart and S. Williams. “Secure Two-party Computation is Practical”. In Advances in Cryptology –Asiacrypt, 2009.

[80].Performance Comparison: Security Design Choices. http://msdn.microsoft.com/en-us/library/ms978415.aspx. 2002.

[81].PolarSSL’s SHA-256 code. https://polarssl.org/sha-256-source-code. 2013

[82].M.O. Rabin. “How to Exchange Secrets with Oblivious Transfer”. Technical Report 81, Harvard University, 1981.

[83].M. Raya and J.P. Hubaux, “Securing Vehicular Ad Hoc Networks”. Journal of Computer Security, vol.15, pp 39-68, 2007.

[84].R. Rivest, A. Shamir, L. Adleman. “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems”. Communications of the ACM 21 (2): pp 120–126, 1978.

[85].A.R. Sadeghi, T. Schneider, and I. Wehrenberg. “Efficient Privacy-preserving Face Recognition”. In ICISC 2009.

[86].M. Scott, "Faster Pairings using an Elliptic Curve with an Efficient Endomorphism". Indocrypt 2005, pp 258-269, 2005.

[87].M. Scott, "Implementing Cryptographic Pairings". *Pairing* 2007, pp 177-195, 2007.

[88].M. Scott, N. Benger, M. Charlemagne, L. J. D. Perez and E.J. Kachisa, "On the Final Exponentiation for Calculating Pairings on Ordinary Elliptic Curves". In Pairing 2009, pp 78-88, 2009.

[89].M. Scott. "A note on twists for pairing friendly curves". Available at ftp://ftp.computing.dcu.ie/pub/crypto/twists.pdf. 2009

[90].N.P. Smart, and B. Warinschi, "Identity Based Group Signatures from Hierarchical Identity-Based Encryption". In Pairing 2009, pp 150-170, 2009.

[91].T. F. Smith, and M. S. Waterman. "Identification of Common Molecular Subsequences". Journal of Molecular Biology 147: pp 195–197. 1981.

[92].M. Stam, A.K. Lenstra, "Efficient Subgroup Exponentiation in Quadratic and Sixth Degree Extensions", In CHES 2002, pp 318-332, 2002.

[93].R. Szerwinski, and T. Guneysu, "Exploiting the Power of GPUs for Asymmetric Cryptology". In CHES 2008, pp 79-99, 2008.

[94].F. Vercauteren, "Optimal Pairings". IEEE transaction of Information Theory, vol.56, issue 1. pp 455-461. 2010.

[95].L. Wang, L. Wang, M. Mambo, E. Okamoto, "New Identity-Based Proxy Re-encryption Schemes to Prevent Collusion Attacks". In Pairing 2010, pp 327-346, 2010.

[96].A. Yao. “Protocols for Secure Computations”. FOCS 23rd Annual Symposium on Foundations of Computer Science, pp 160-164, 1982.

[97].K. Yoneyama, “Strongly Secure Two-Pass Attribute-Based Authenticated Key Exchange”. In Pairing 2010, pp 147-166, 2010.

[98].Y. Zhang, C.J. Xue, D.S. Wong, N. Mamoulis, S.M. Yiu, “Acceleration of Composite Order Bilinear Pairing on Graphics Hardware”. Cryptology ePrint Archive, available at http://eprint.iacr.org/2011/196.pdf, 2011.

[99].C.A. Zhao, F. Zhang and J. Huang, “A Note on the Ate Pairing”. International Journal of Information Security, Vol.6, No.7, pp 379-382, 2008.