

Fastplay: A Parallelization Model and Implementation of SMC on CUDA Based GPU Cluster Architecture

Shi Pu, Pu Duan, Jyh-Charn Liu
Department of Computer Science and Engineering,
University of Texas A&M University,
College Station, TX, United States
shipu, dp1979, liu@cse.tamu.edu

Abstract— We propose a four-tiered parallelization model for acceleration of the secure multiparty computation (SMC) on the CUDA based Graphic Processing Unit (GPU) cluster architecture. Specification layer is the top layer, which adopts the SFDL of Fairplay for specification of secure computations. The SHDL file generated by the SFDL compiler of Fairplay is used as inputs to the function layer, for which we developed both multi-core and GPU based control functions for garbling of various types of Boolean gates, and ECC-based 1-out-of-2 Oblivious Transfer (OT). These high level control functions invoke computation of 3-DGG (3-DES gate garbling), EGG (ECC based gate garbling), and ECC based OT that run at the secure protocol layer. An *ECC Arithmetic GPU Library* (EAGL), which co-run on the GPU cluster and its host, manages utilization of GPUs in parallel computing of ECC arithmetic. Experimental results show highly linear acceleration of ECC related computations when the system is not overloaded; When running on a GPU cluster consisted of 6 Tesla C870 devices, with GPU devices fully loaded with over 3000 execution threads, Fastplay achieved 35–40 times of acceleration over a serial implementation running on a 2.53GHz duo core CPU and 4GB memory. When the execution thread count exceeds this number, the speed up factor remains fairly constant, yet slightly increased.

Keywords- secure multiparty computation (SMC), Oblivious Transfer (OT), ECC, Graphic Processing Unit (GPU)

I. INTRODUCTION

Since publication of the seminal work of the Millionaire’s problem [8] two decades ago, the *secure multiparty computation* (SMC) paradigm has been widely recognized as a general framework for design of privacy-preserving protocols. In the Millionaires problem, two principals Alice and Bob want to know who has more wealth without telling each other their own amounts. A generalization of this problem, called *secure function evaluation* (SFE), was further proposed in [6]. In SFE, two or more parties perform a joint *computing function*, e.g., ordering of inputs, matching of inputs, addition, etc., so that specific inputs and outputs are unveiled or protected under controlled conditions. The integration of the *oblivious transfer* (OT) protocol [18] and *garbled circuit* is a time proven architecture to implement different SMC based computing functions.

SMC provides strong privacy protection but it incurs significant computations and bandwidth costs. To meet the performance needs of real world applications, there is a strong need to accelerate SMC computations for large scale

privacy preserving applications, e.g., DNA based biometrics [27][28], information queries [29], auctions [30], network security alert correlation [31], etc. Some systems directly customized the secure functions to fit certain specialized functionality, for example, auction or edit distance computation [28, 30, 32]. On the other hand, Fairplay [33] and its variations proposed in [27, 34, 35, 37] emerge as a general framework for specification of secure functions, based on the *secure function definition language* (SFDL). A compiler parses an application specified in SFDL to generate a net-list of *gates* and their interconnection topology based on the *secure hardware definition language* (SHDL). Fairplay uses Java crypto packages for run-time execution.

In SMC based computations, OT has the highest cost, both for computations and communications, while other crypto related functions also incur non-trivial computing costs. Noting that because all data bits can be processed independently, based on identical computing steps, we observe that they are ideal applications for acceleration based on the SIMD parallel architectures.

Based on the above observation, the main objective of this paper is aimed at exploring system design issues related to parallelization of large scale SMC applications. Our system, called *Fastplay*, has its foundation on the Fairplay for specification of applications and compilation of an SFDL program into SHDL. For ECC arithmetic operations on GPU, we reengineered a portion of MIRACL [5] to fit them into the restricted resource environments on GPU. Furthermore, we developed a full suite of GPU resource management functions, CPU-GPU co-run controls, and various Boolean gates used in SMC.

For performance benchmarking, we dissected the complete execution process of an SMC circuit into several computing parts and compare the execution time of each part with a serial version. We also benchmarked the bandwidth for EGG, 3-DGG, and ECC-based 1-out-of-2 OT protocols respectively. When running on a GPU cluster consisted of 6 Tesla C870 devices, with GPU devices fully loaded with over 3000 execution threads, Fastplay achieved 35–40 times of acceleration over a serial implementation running on a 2.53GHz duo core CPU and 4GB memory. When the execution thread count exceeds this number, the speed up factor remains fairly constant, yet slightly increased.

Main designs to achieve the performance goal are summarized as follows:

- [1]. A parallel implementation for ECC-based 1-out-of-2 OT protocol on the GPU architecture. Both ECC-based and 3-DES based gate garbling protocols are

supported. A caching technique was developed. The ECC based protocol is well suited for GPU cluster architecture because most of its point multiplications can share the unique and constant public point P . For case studies, we tested “equality checking” and “minimum of two” circuits.

- [2]. The *ECC Arithmetic GPU Library* (EAGL) is designed to minimize the context switching overhead of GPU execution and full utilization of GPU devices. We streamlined synchronization and execution control threads between GPUs and the host CPU.

The rest of this paper is organized as follows: section 2 presents the foundation work of Fastplay. Section 3 discusses the secure protocol layer. Section 4 extends the prototype of gate garbling protocol to various types of gate garbling functions and circuit connection issues. The ECC Arithmetic GPU Library is presented in section 5. Experimental results and evaluation are illustrated in section 6. Section 7 discusses related work. Conclusions are drawn in section 8.

II. FOUNDATIONAL WORK

The objective of Fastplay is to replicate the single-bit Boolean secure function evaluation [9] onto the massive number of execution threads available on a GPU cluster to achieve the goal of acceleration for large scale applications. Fastplay is an ECC-based implementation for the Boolean secure function evaluation. For ECC-based arithmetic computations, we made extensive modifications to some portion of the MIRACL code base for optimal execution on the GPU architecture. Moreover, we adopt the Fairplay [33] as the programming model to achieve compatibility with existing serial solutions.

In the Boolean secure function evaluation problem a secure function $f(x, y)$ has 1-bit input x from a *sender*, 1-bit input y from a *chooser*, and one 4-entry truth table. The sender and the chooser are not allowed to learn about each other’s input during the protocol execution. To achieve this goal, sender generates two random N -bit integers k_s^0 and k_s^1 mapping to x ’s potential values 0 and 1, k_c^0 and k_c^1 mapping to y ’s potential values, k_o^0 and k_o^1 mapping to potential output values 0 and 1. Sender holds the plain truth table, which saves plain output values 0 or 1, indexed by its corresponding input values (00, 01, 10, 11). For each plain truth table entry T_{ij} that saves output value z , sender encrypts it as $E(T_{ij}) = E_{k_s^i}(E_{k_c^j}(k_o^z))$, where $E_{k_s^i}$ means encryption using key k_s^i that represent sender’s input value i . After encrypting all four truth table entries ($ij = 00, 01, 10$ and 11), sender transfers message $\varepsilon = \{E(T_{ij}), k_s^x\}$ that maps x ’s real value, pre-defined unique characteristic of k_o^0 and k_o^1 to Bob, and delivers k_c^y to chooser via OT. Finally, chooser can successfully decrypt one of four $E(T_{ij}) = k_o^z$ that maps the output value $z = f(x, y)$.

Fairplay system is the first general software suite that supports implementation of the general SMC scheme based on SFDL and its compiler tools. The programming model of Fairplay system is illustrated in Fig. 1. The programming model can be modeled into four layers: *specification layer*, *function layer*, *secure protocol layer*, and *arithmetic layer*.

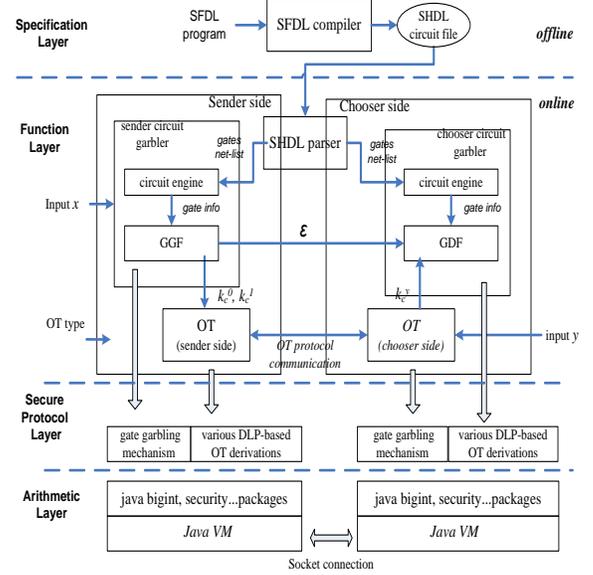


Figure 1. The programming model of Fairplay

We adopted the SFDL compiler in specification layer for Fastplay, but we developed our own implementations for remaining layers.

At the specification layer, SFDL supports specification of a general secure function $f(x, y)$ as a *circuit* that includes customized multi-bit I/O from two parties, and the function body describing the functionality of the circuit. An offline SFDL compiler is employed to convert a *circuit* program into a gates-level hierarchy file written in SHDL. Function layer includes an online SHDL parser that generates a net-list of gates based on the SHDL file, a circuit engine that collects each gate’s information and invokes a pair-wise gate garbling/de-garbling function (GGF/GDF) on each side respectively, and a pair-wise sender/chooser 1-out-of-2 OT instances that are invoked by GGF/GDF. The gate garbling schemes and various DLP-based OT implementation derivations can be summarized as the secure protocol layer. The arithmetic layer is composed of interfaces in java large integer, security and associated packages. At runtime, the sender/chooser side independently runs on a Java virtual machine, and connects with the other side via socket.

MIRACL [5] contains comprehensive elliptic curve point arithmetic routines, e.g., point multiplication, point addition, subtraction, doubling, normalization and so on. Our initial attempts to port MIRACL directly to the CUDA environment proved to be virtually impossible due to numerous challenges in buffer management, memory access model, and execution control, etc.

Fig. 2 depicts the high-level programming model of Fastplay. In function layer, the circuit engine of Fastplay employs a *gate garbling task dispatcher* to organize the gate-level and circuit-level connections, and identify the type of GGFs/GDFs needed to invoke. Similar to solutions discussed in [27, 37], a large application can be decomposed into multiple circuits. This dispatcher also supports the multi-level circuit hierarchy by re-using predecessor circuits’ keys saved by the sender in successor circuit’s gate garbling. It

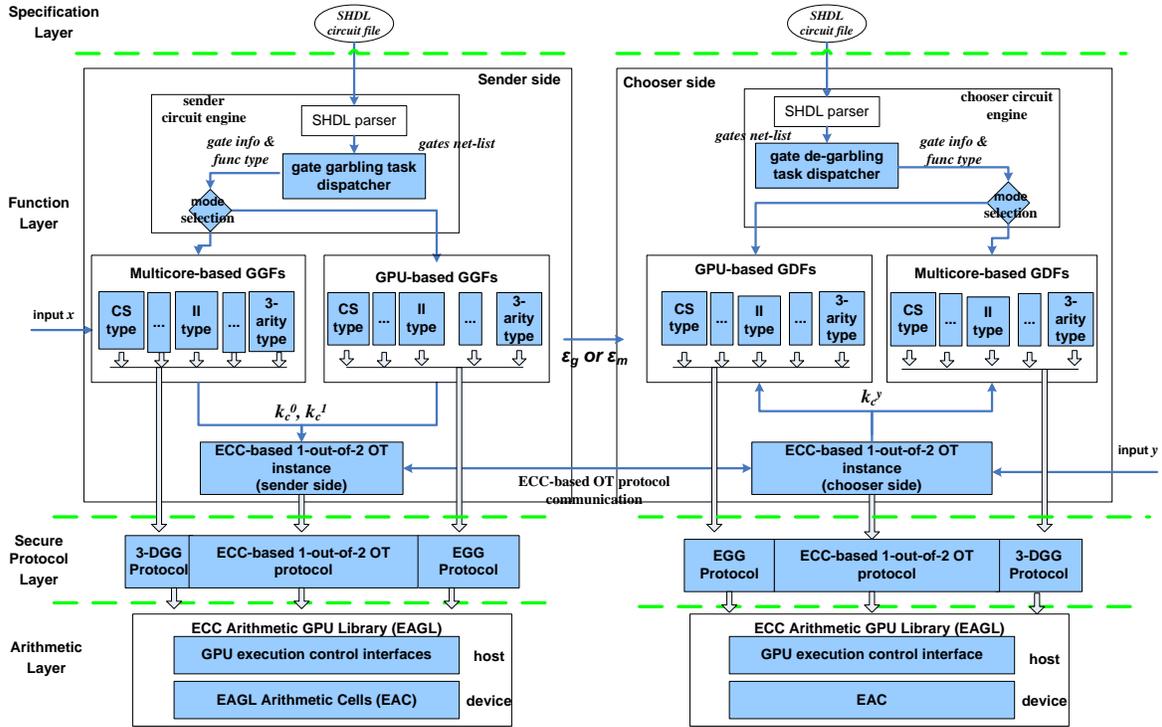


Figure 2. The high-level programming model of Fastplay

allows the chooser to decrypt successor circuit based on the predecessor circuit's SMC result.

Fastplay supports both multicore-based and GPU-based GGFs/GDFs. Triple-DES is used for gate garbling protocol (3-DGG protocol) in the multicore implementation, and the encryption messages $\varepsilon_m = \varepsilon$. On the other hand, an ECC-based gate garbling protocol (EGG protocol) was implemented for the GPU-based GGFs/GDFs. The 3rd component of their encryption messages ε_g uses $\{k_c^0, k_c^1\}$ directly, without leaking any sensitive data. We further note that the gate's arity and types of gate inputs can determine details of a garbling function, such as whether or not OT needs to be involved. As a result, we implemented one type of GGF/GDF for each type of gate so that the synchronization mechanism can be greatly simplified.

For the secure protocol layer, Fastplay consists of three secure protocols: *ECC-based gate garbling protocol* (EGG protocol), *triple-DES based gate garbling protocol* (3-DGG protocol), and *ECC-based 1-out-of-2 OT protocol*. Both EGG and 3-DGG protocols need to use the ECC-based 1-out-of-2 OT protocol for the secure transferring of k_c^0 (or k_c^1) that will be needed for decryption of the truth table.

The arithmetic layer of Fastplay, the *ECC Arithmetic GPU Library (EAGL)*, consists of two major modules for control and interfaces between the host and GPU devices. The *GPU execution control interface* runs on host and manages synchronization of GPU-host threads, their data movements. It is also responsible for configuration, initialization and destroying of GPU context, and exception handling. A massive number of instances can be dispatched to GPU threads for parallel executions. Major reengineering

efforts include (1) elimination of unused branches, concurrent write conflicts, one-time use variables, and (2) optimization of data structures for more efficient memory usage.

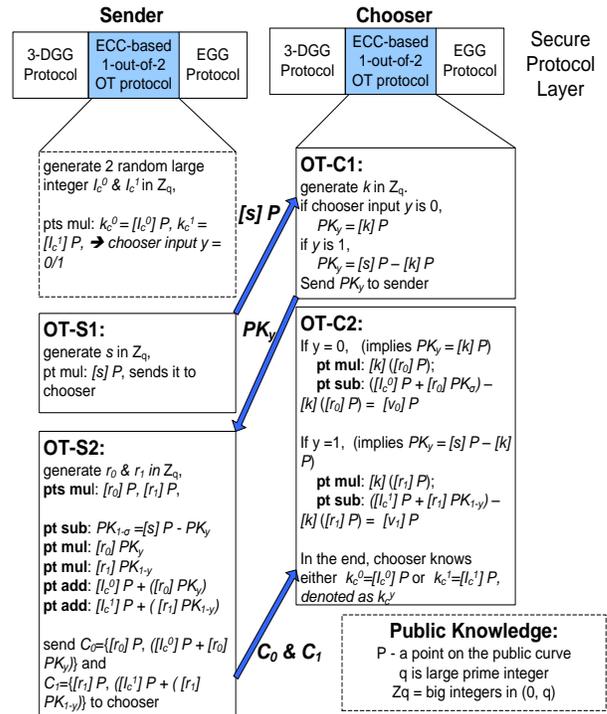


Figure 3. ECC-based 1-out-of-2 Oblivious Transfer Protocol

III. SECURE PROTOCOL LAYER

A. ECC-based 1-out-of-2 OT protocol

In the literature, *oblivious transfer* (OT) [18] can be conceptually understood as *sender* controls the garbling parameters I_0 and I_1 for the plaintext message of M_0 and M_1 at each transaction, and *chooser* can choose between one of the two input values 0 or 1. M_0 and M_1 can be published after they are encrypted by a well known encryption scheme using $[I_0]P$ and $[I_1]P$ as their keys, respectively. Generation of the M_0 and M_1 is domain dependent, and therefore not considered as a part of the OT. In the context of the EGG and 3-DGG protocol, the output value 0 and 1 of a gate's plain truth table entries are M_0 and M_1 , whose decryption keys I_c^0 and I_c^1 plays the role of garbling parameters I_0 and I_1 in the general OT paradigm. For consistency, we use I_c^0 and I_c^1 in the following descriptions of our ECC-based 1-out-of-2 OT protocol.

The ECC-based 1-out-of-2 OT protocol is illustrated in Fig. 3. At first, with no knowledge of the chooser's privacy y , the sender generates two keys $k_c^0 = [I_c^0]P$ and $k_c^1 = [I_c^1]P$, where I_c^0 and I_c^1 are two large integers and P is a base ECC point of a chosen elliptic curve. Here $[I_c^0]P$ and $[I_c^1]P$ are standard point multiplication method in ECC. Then, the sender starts transferring a randomized point $[s]P$ to the chooser. After receiving $[s]P$, chooser generates another random point $PK_y = [k]P$ if the chooser's input value $y=0$, otherwise $PK_y = ([s]P - [k]P)$. In this step, the chooser associates its privacy y with PK_y and transfers PK_y back to sender. Then the sender generates two random points $[r_0]P$ and $[r_1]P$, and seals the garbling parameters I_c^0 and I_c^1 through point multiplication and point addition as the homomorphic operation. The encrypted messages are denoted by C_0 and C_1 . Finally, the chooser can only correctly decrypt either C_0 or C_1 according to his original privacy y . As a result, chooser will hold either $[I_c^0]P$ (if $y=0$) or $k_c^1 = [I_c^1]P$ (if $y=1$). In the whole process, generating $k_c^0 = [I_c^0]P$, $[I_c^1]P$, $[s]P$, $[k]P$, $[r_0]P$ and $[r_1]P$ are usually viewed as sender's *initialization phase* [42], so is the $[k]P$ for the chooser side.

In terms of computing cost, the sender needs five point multiplications in his initialization phase. In his online execution phase, he needs two point multiplications and three point additions/subtractions in step *OT-S2*. For the chooser, there is one point multiplication in his initialization phase, two point multiplications, and two or three point subtractions in his online execution phase.

As seen later, the size of an elliptic curve point is 320 bits. In terms of bandwidth, the communications cost from sender to chooser is 320 bits in pre-computing phase (*OT-S1*), and $4 * \text{sizeof}(\text{point}) = 4 * 320 \text{ bits} = 1280 \text{ bits}$ in the computation phase (*OT-S2*), and that from chooser to sender is 320bits.

The security of the ECC-based 1-out-of-2 OT protocol is based on the well-known ECDLP. For *outside adversaries*, an eavesdropper *Eve* cannot use P , $[s]P$ and PK_y to find s and k to discover I_c^0 and I_c^1 if he cannot calculate discrete logs on an elliptic curve. Similarly, *Eve* cannot use $[r_0]P$, $[r_1]P$, $[r_0]PK_y$ and $[r_1]PK_{1-y}$ to find k to discover I_c^0 and I_c^1 if

she cannot calculate discrete logs. In other words, *Eve* cannot obtain the key $k_c^0 = [I_c^0]P$ or $k_c^1 = [I_c^1]P$ that must be uniquely acquired by the chooser to decrypt a certain entry of a truth table later in the EGG protocol and the 3-DGG protocol. For *inside adversaries*, we use $[r_0]PK_y + [I_c^0]P/[r_1]PK_{1-y} + [I_c^1]P$ to protect I_c^0/I_c^1 . Since r_0 and r_1 are two randomly chosen large integers, a malicious chooser cannot use $[r_0]P/[r_1]P$ and $[r_0]PK_y/[r_1]PK_{1-y}$ to derive PK_y/PK_{1-y} in $\{[r_0]P, [I_c^0]P + [r_0]PK_y\}$ and $\{[r_1]P, [I_c^1]P + [r_1]PK_{1-y}\}$ so that I_c^0 and I_c^1 are protected unless he can solve ECDLP. In other words, the chooser cannot have both k_c^0 and k_c^1 simultaneously so he cannot to decrypt entries of the truth table that are not indexed by his real input value. Similarly, a malicious sender cannot know the results of the chooser's selection (I_c^0 or I_c^1) since k cannot be derived from PK_y and P .

B. ECC-based Gate Garbling Protocol (EGG Protocol)

In a general SMC protocol, a sender keeps the computation logic and its inputs in secret. A chooser provides his inputs and eventually obtains the computation results without knowing sender's inputs, while the sender also does not learn the chooser's inputs. Usually any complicated computation logic can be represented as the combination of multi-level Boolean gates. Although the garbling/de-garbling behaviors are slightly changed when the gate's arity and inputs' type are different, the truth table encryption/decryption scheme does not change. In this section, the discussion of the EGG protocol is based on a prototype of the 2-arity gate with two inputs from a chooser and a sender, respectively. This gate is of the CS (chooser-sender) type. The security properties of EGG protocol under the semi-honest model are:

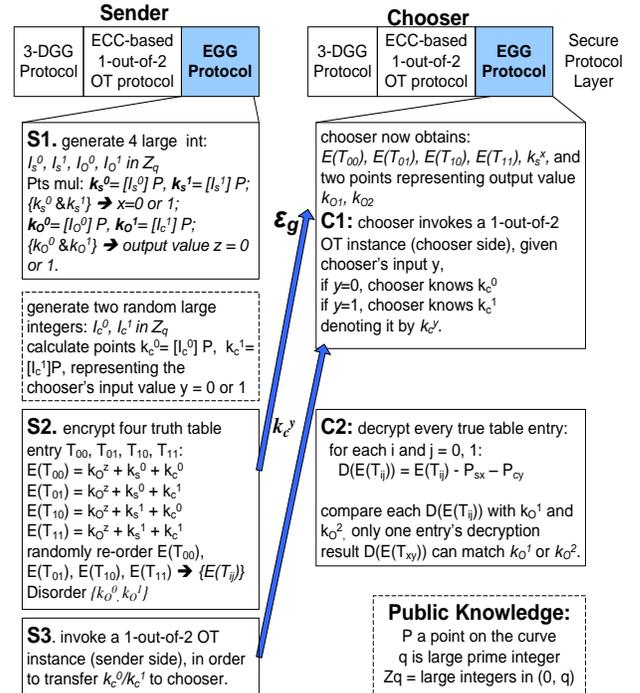


Figure 4. Prototype of EGG protocol

- [1]. The chooser is able to decrypt only one of the four entries, which is indexed by the inputs from both sides;
- [2]. Neither the chooser nor the sender knows the other's input value;
- [3]. Any third-party adversary cannot decrypt the truth table entry;

To achieve Property 1, garbling/de-garbling needs to utilize the standard ECC point addition and subtraction (i.e. given point P and Q, P+Q-Q=P). To achieve Property 2 and 3, one 1-out-of-2 OT instance is invoked to transfer the key determined by the chooser's input. Fig. 4 illustrates the encryption/decryption scheme. The EGG protocol can be described as follows:

(1)Step *SI*: the sender generates two 160-bit large integers, I_s^0 and I_s^1 , and another two 160-bit large integers I_o^0 and I_o^1 . Then the sender calculates ECC points k_s^0, k_s^1, k_o^0 and k_o^1 following the formula that $k_s^i = [I_s^i]P$, $k_o^z = [I_o^z]P$, here $i, z = 0$ or 1 , and P is one public point on the pre-selected curve. Points k_s^0 and k_s^1 are used to map the possible value 0 and 1 of sender's 1-bit input x . Similarly, k_o^0 and k_o^1 map to the possible value 0, 1 of output z .

Preparation for 1-out-of-2 OT: Although we mentioned the generation of k_c^0 and k_c^1 in ECC-based 1-out-of-2 OT protocol, they were actually generated k_c^0 and k_c^1 . Sender generates k_c^0 and k_c^1 that map to the possible values 0 and 1 of the chooser's 1-bit input y . As we will see later, generation of k_c^0 and k_c^1 is counted as the execution of the 1-out-of-2 OT instance in benchmarking.

(2)Step *S2*: we denote the four entries of plain truth table as T_{00}, T_{01}, T_{10} , and T_{11} respectively, where the subscripts of T are actually specified by the gate description in the circuit list file. For convenience, T 's 1st subscript represents the sender's input bit, and T 's 2nd subscript representing the chooser's input bit. For each truth table entry, the sender encrypts it as: $E(T_{ij}) = k_o^z + k_s^i + k_c^j$. Here, k_o^z is the point that maps to the output value z as the result of corresponding truth table entry T_{ij} . By denoting the sender's real input value as x , $x = 0$ or 1 , the sender transfers $\varepsilon_g = \{\text{randomly disordered } \{E(T_{ij})\}, k_s^x, \text{ and randomly disordered } \{k_o^0, k_o^1\}\}$ to the chooser.

(3)Step *S3*: The sender invokes an ECC-based 1-out-of-2 OT instance (sender side) so that the chooser can select to know either k_c^0 or k_c^1 based on the value of his 1-bit input y . The key the chooser finally obtained is denoted by k_c^y .

(4)Step *CI*: The chooser invokes an ECC-based 1-out-of-2 OT instance (chooser side) and gets k_c^y . The chooser also receives ε_g .

(5)Step *C2*: For each encrypted truth table entry $E(T_{ij})$, the chooser tries to decrypt it as: $D(E(T_{ij})) = E(T_{ij}) - k_s^x - k_c^y$. Although he gets four different points as computation results, only one of them matches the k_o^z generated by sender. The chooser takes k_o^z as the computing results.

The size of the data transferred from the sender to the chooser is $2^{\text{arity}} \cdot 320 + 320 \cdot 2 + 320$. Given that a gate's arity is 2, the total size is 2240bit. Since our protocol follows the standard SMC architecture [6] and elliptic curve discrete logarithm problem (ECDLP) [1], the proofs of all the three security properties are obvious and omitted here.

In terms of computing cost, the sender needs four scalar point multiplications in its key generation phase ($[I_s^i]P$, $[I_o^z]P$, $i, z=0,1$), and $2^{\text{arity}} \cdot 2$ point additions for online encryption. The chooser only needs $2^{\text{arity}} \cdot 2$ point subtractions for online decryption, and $(2^{\text{arity}} + 2)$ point normalizations to normalize the decryption results $D(E(T_{ij}))$ and $\{k_o^0, k_o^1\}$ to affine co-ordinates for point equality checking. In the EGG protocol, the most expensive computation is the sender's key generation operations, since it needs expensive point multiplication operations. However, random point generation is independent of the truth table encryption/decryption process, and the number of random points N needed to be generated for each gate is fixed. Pre-computation of random points when computation resources are free is an optional way to reduce the computing cost.

C. 3-DES-based Gate Garbling Protocol (3-DGG Protocol)

Based on the 2-arity CS type gate, the 3-DGG protocol is as follows:

(1)Step *3D-S1*: The sender generates four 160-bit random integers $k_s^0, k_s^1, I_o^0, I_o^1$, and then generates a set of three different 64-bit keys $K_s^0 = \{\text{key}1_s^0, \text{key}2_s^0, \text{key}3_s^0\}$ from k_s^0 , and another set $K_s^1 = \{\text{key}1_s^1, \text{key}2_s^1, \text{key}3_s^1\}$ from k_s^1 .

Like the EGG protocol, the sender prepares point k_c^0 and k_c^1 for the 1-out-of-2 OT that will be invoked in step *3D-S4*.

(2)Step *3D-S2*: The sender generates a set of 64-bit keys $K_c^0 = \{\text{key}1_c^0, \text{key}2_c^0, \text{key}3_c^0\}$ from ECC points k_c^0 , and another set $K_c^1 = \{\text{key}1_c^1, \text{key}2_c^1, \text{key}3_c^1\}$ from k_c^1 . All the keys are generated based on different segments of the point's X and Y co-ordinates.

(3)Step *3D-S3*: For each truth table entry T_{ij} , the index i and j represent the input value from the sender and the chooser, respectively. Suppose the plain output value of this entry is z , z is 0 or 1, then $T_{ij} = I_o^z \ll 32 + 0x0000$, so that the chooser can easily identify which entry is decrypted successfully because the probability of generating a 32-bit consecutive 0 using two levels of 192-bit triple-DES with incorrect keys is very low. Encryption of entry T_{ij} is $E(T_{ij}) = 3\text{-des}(K_c^j, (3\text{-des}(K_s^i, T_{ij}), \text{ENCRYPT}), \text{ENCRYPT})$. Sender sends $\varepsilon_m = \{\text{randomly re-ordered } \{E(T_{ij})\}, k_s^x\}$ to the chooser.

(4)Step *3D-S4*: The sender invokes a 1-out-of-2 OT instance (sender side) in order to transfer either k_c^0 or k_c^1 to chooser, the point chooser finally gets is denoted by k_c^y .

(5)Step *3D-C1*: The chooser invokes a 1-out-of-2 OT instance (chooser side) to receive k_c^y . The chooser also receives ε_m .

(6)Step *3D-C2*: The chooser generates $K_s^x = \{\text{key}1_s^x, \text{key}2_s^x, \text{key}3_s^x\}$ from k_s^x , and generates K_c^y from ECC point k_c^y .

(7)Step *3D-C3*: The chooser decrypts truth table entries as $D(E(T_{ij})) = 3\text{-des}(K_s^x, (3\text{-des}(K_c^y, E(T_{ij}), \text{DECRYPT}), \text{DECRYPT}))$. In the end, only $D(E(T_{ij}))$ that has the last 32-bit as $0x0000$ is the correct decryption result. The chooser stores $I_o^z = D(E(T_{ij})) \gg 32$.

The size of data transfer from the sender to the chooser is $2^{\text{arity}} \cdot 192 + 160$. Given arity equals to 2, this size is 928 bit.

IV. FUNCTION LAYER

The task of the *gate garbling task dispatcher* is to extract information about the gates and identifying the type of GGFs/GDFs needed to be invoked. Although all the GGFs/GDFs in the GPU-based/multicore-based groups follow the prototype of EGG /3-DGG protocols respectively, there are slight differences in certain steps. Taking the 3-arity gate as an example, its three 1-bit inputs may come from sender, chooser or predecessor gate(s): if a gate's input is from the sender or the chooser, the sender needs to one pair of points/large integers for itself in step *S1/3D-S1*. If one gate itself and all its predecessor gates only take the sender's inputs, this gate can be treated as an input wire from the sender, because the sender himself can decide the output value of this gate. For the gate that does not directly get input from the chooser, OT can be ignored.

Besides invoking various types of GGFs/GDFs, the gate garbling task dispatcher need to fetches the encryption keys/secure computing results from predecessor gates' GGFs/GDFs for each gate, so that gates and circuits with interdependency can be connected. The key issue for gate connection and circuit connection is key re-using. Taking the gate-level hierarchy as an example, when a complex circuit contains hundreds or thousands of gates, it is possible that some gates' input(s) are not from the sender or chooser, but from intermediate results of their predecessor(s). For succinctness, we still assume all gates are 2-arity, and show one example of gate connection:

As shown in Fig. 5, two *CS* type GGFs/GDFs are invoked for gates 1 and 2, and an *II* type (*Intermediate-Intermediate*) of GGF/GDF are invoked for gate 3. Taking the GPU-based GGFs/GDFs as an example, gate 3's GGF does not need to generate the four random points $k_{3s}^0, k_{3s}^1, k_{3c}^0$ and k_{3c}^1 that are used to encrypt Gate 3's truth table. Instead, its GGF encrypts gate 3's truth table entries by using the points $k_{10}^0, k_{10}^1, k_{20}^0$ and k_{20}^1 that represent the possible output of gates 1 and 2. In Step *S1*, the sender only needs to generate k_{30}^0 and k_{30}^1 that represents to gate 3's possible output values 0 and 1. When gate 1 and 2 complete their garbling processes, the sender holds the four points $k_{10}^0, k_{10}^1, k_{20}^0$ and k_{20}^1 ; the chooser holds either k_{10}^0 or k_{10}^1 , and either k_{20}^0 or k_{20}^1 , denoting by k_{10}^z and k_{20}^w . Similar with *S2* in Fig. 4, the sender encrypts gate 3's truth table entries by adding k_{10}^0 or k_{10}^1 , and k_{20}^0 or k_{20}^1 , to k_{30}^0 or k_{30}^1 . On the other hand, since the chooser has already known k_{10}^z and k_{20}^w , he uses these two points to decrypt the truth table entries, described as *C2* Fig. 4. In the end, he can only successfully decrypt one of four entries. In such a process described above, *security property 1* is held because the chooser does not fully hold the two keys to obtain the other three plain truth table entries. *Property 2* is true because gate 1 and gate 2 hold *property 2* and gate 3 inherits these inputs from gate 1 and 2, and does not accept extra inputs. *Property 3* holds because gate 1 and gate 2 hold *property 3*, so that a 3rd party adversary cannot decrypt gate 1's and 2's truth table and thus they do not know k_{10}^z and k_{20}^w . Consequently, they cannot decrypt gate 3's truth table. The multicore-based GGFs/GDFs follow the same idea and are omitted here.

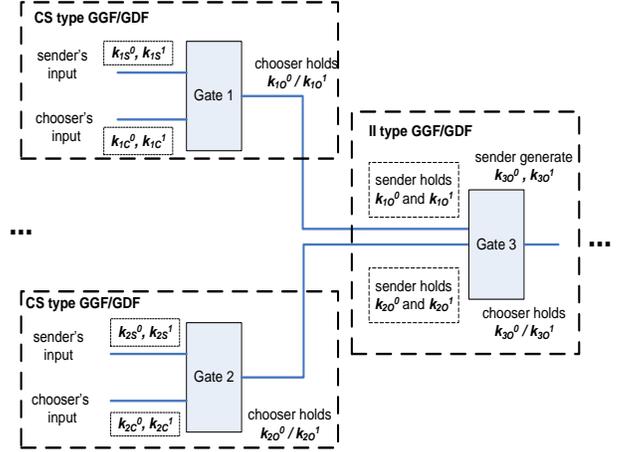


Figure 5. GGFs/GDFs connection for gate-level hierarchy

Furthermore, such a gate connection scheme also supports the multi-level circuit hierarchy if several circuits are combined together to form a more complicated computation logic. Supposing circuit A provides 1-bit output to circuit B. When secure computing of A is done, the sender holds the two points k_{A0}^0 and k_{A0}^1 , meanwhile the chooser saves the point k_{A0}^z as the secure computing result of A. In the secure computing of B, The sender will use k_{A0}^0 and k_{A0}^1 to encrypt the truth table entries of the gate that accepts A's output. As a result, the chooser can only decrypt one entry successfully because he only knows one of k_{A0}^0 and k_{A0}^1 exclusively.

V. ARITHMETIC LAYER

A. GPGPU Parallel Computing Architecture

The architecture of GPGPU follows the SIMD (*single instruction multiple data*) architecture. That is, all processing unit executes the same instruction with various operands. Currently, two of the most popular GPGPU architectures are NVIDIA CUDA (Compute Unified Device Architecture) and ATI Stream. The CUDA architecture includes two parts: *host* and *device*. Host determines what data is fed to the devices, when devices start to run, how parallel computation is organized and how the data are saved and accessed on device. In hardware, the device contains one or more GPU boards, and each board has several streaming multiprocessors (SM). By containing multiple pipelined ALUs, one SM is able to support simultaneous execution of a warp of threads, whose size is 32, and one SM is able to schedule up to 24 warps in the most CUDA platforms. Registers and local memory are private for each thread; shared memory is private for each SM, and all the threads running on this SM share it; the global memory is open for all threads on the board. The delays of accessing a register and shared memory are almost the same, without considering write conflicts. Accessing global memory or local memory is hundreds times slower than accessing register or shared memory. In software, the CUDA programming language provides the software interface for developers to access and control devices on the host. The program that runs on devices is called as *kernel* program. Before a kernel program

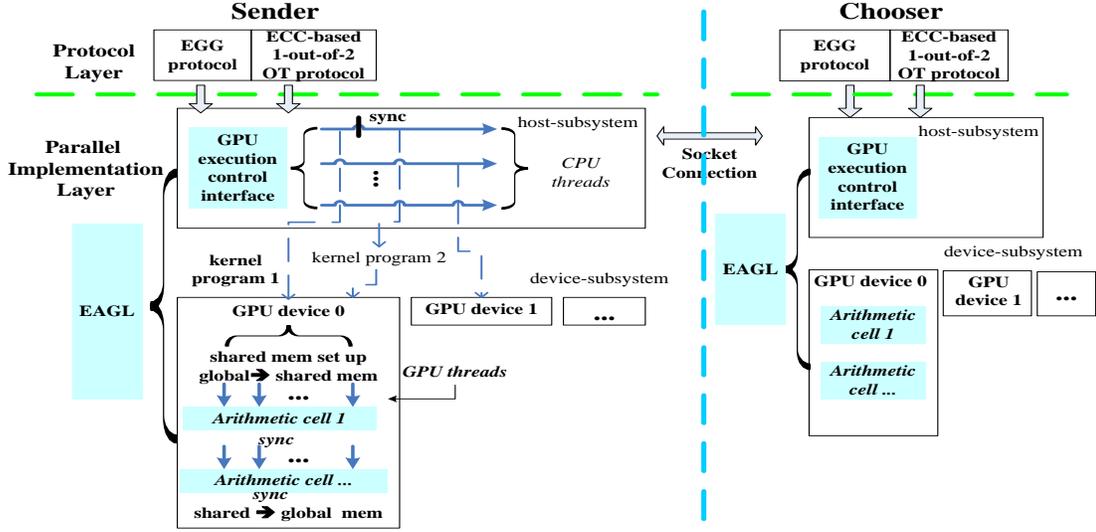


Figure 6. The execution process of GPU execution control interface

runs, its parallelism parameters – *number of blocks* and *number of threads per block* are specified. One SM can only concurrently run one block and threads in this block is organized as one or multiple warps. In sum, the parallelism that one GPU board provides equals to (*number of blocks* * *number of threads per block*). More details are referred to [42].

Our motivation for parallel execution of ECC arithmetic on a GPU platform is based on two observations. First, one of the most frequent ECC arithmetic operations in the EGG protocol and the ECC-based 1-out-of-2 OT protocol is point multiplication $[k]P$, where P is a fixed unique base point and can be shared by all threads. Consequently, part of point multiplication can be moved to offline pre-computing, and the shared memory space is saved by maintaining one copy of P for all threads on the same SM. Furthermore, as we will show later, the size of the ECC point (320 bit) is small and thus ECC arithmetic is suitable for porting to a computation platform with constrained fast-accessing memory resource.

Porting a complicated system from CPU-based programming model to a GPGPU based model is challenging. For example, SMs on most GPGPU platforms do not have a branch predictor, threads in the same warp have to wait until all members calculate their jump address and finish the IF (instruction fetch) phase when they meet a branch. Identifying the potential write conflicts is another critical issue. The constrained fast-accessing memory resource requires highly efficient space utilization mechanism for buffering intermediate results, i.e. buffer re-using and dynamic data structure size optimization, etc.. The accessing of slow local memory and global memory has to be minimized in usual case, however, sometimes there is a tradeoff between accessing high delay memory and maintaining a high degree of parallelism. For complicated programs, it is critical to identify which part of code is most suitable to run on a CPU, and which part is better for GPGPU-based execution. In the end, most CUDA platforms

only support debugging in simulation mode, not real-time debugging.

The rest of this section presents the two part of ECC Arithmetic GPU Library (EAGL): GPU execution control interfaces running on the host sub-system, and the EAGL arithmetic cells running on device sub-system.

B. ECC Arithmetic GPU Library (EAGL)

In the secure protocol layer, any operation that uses elliptic curve cryptography arithmetic needs EAGL to be involved in the parallel implementation layer. The theory of *Elliptic Curve Cryptography* (ECC) was first proposed by Koblitz [1] and Miller [2]. Let K be a finite field, and suppose $E(K)$ is an additive group of points on an elliptic curve E over K , $E(K)$ is defined as the set of points (x, y) , $\forall x, y \in K$, (x, y) satisfy $y^2 = x^3 + ax + b$ (the curve used in our implementation). Here we took K as F_p - as a finite prime field, where p is a large prime number. Here we use P, Q, R to denote points in $E(K)$, and k to denote a large integer. The regular elliptic curve arithmetic operations are point addition ($R=P+Q$), point subtraction ($R=P-Q$), point doubling, ($Q = [2]P$), and point multiplication ($Q = [k]P$). Given certain elliptic curve $E(F_p)$, the arithmetic routines of all the operations discussed above can be computed on *projective coordinates* [23]. The main reason is that elliptic curve group operations on projective coordinates do not need inversions and are more efficient than those on affine coordinates. Before running these ECC arithmetic routines on GPU platforms, GPU execution control interface need to set up an execution environment, and then these routines are fulfilled by one or several EAGL arithmetic cells.

1) GPU Execution control Interfaces

The relationship of GPU-based GGF or GDF to GPU execution control interface is 1:1, so that there is no branch for identifying whether certain gate garbling behavior, i.e. 1-out-of-2 OT or key generation, needs to be done or not. As shown in Fig. 6, the GPU execution control interface first sets up CPU threads and bind each CPU thread to an

available GPU device. As CPU threads start to execute, the GPU execution control interface takes the charge of runtime synchronization of GPU and CPU threads and their data movement, GPU context initialization and destroying, GPU parallelism configuration, GPU execution runtime exception handling. To minimize the overheads on GPU context initiation and device memory allocation, the GPU execution interface always stays in the same GPU context as long as it is not interrupted by necessary data communication, and always allocates a large chunk of device memory and partitions it for all the EAGL arithmetic cells invoked in its life time.

The GGFs/GDFs in the multicore-based group do not depend on GPU execution control interfaces as much as those in GPU-based group. The GPU execution interface is invoked only in step 3D-S1 for parallel generation of a large volume of random large integers, and in step 3D-S4 and 3D-C1 where ECC-based 1-out-of-2 OT instances are called.

2) EAGL Arithmetic cells (EAC)

When the GPU execution control interface dispatches kernel programs to GPU devices, the kernel programs' programming model generally follows a 3-step scheme, including (1) setting up shared memory space and moving data from global to shared memory, (2) calling certain arithmetic cell, and (3) updating the result back to global memory.

The basic data structures in EAC are shown in Fig. 7 – the *large integer* and *ECC point*. Large integer precision is $32*k$ bits. Although 160-bit keys are secure enough as the key length in ECC, k is set as 7 in EAC for preventing possible overflow. In certain arithmetic cell like large integer modular multiplication, k temporarily grows to 11 and recovers back to 7 after it completes. However, points are normalized and k is cut to 5, Z coordinate and the marker are removed in protocol communication messages for saving

```

struct large_int{
    unsigned int array[k];
    unsigned in length;
}

struct Point{
    struct large_int X;
    struct large_int Y;
    struct large_int Z;
    unsigned int marker;
}

```

Figure 7. Data structures of large integer and ECC point

bandwidth.

All EAGL's arithmetic cells share public knowledge $\{a$ point P , a large prime integer q , and the set $Z_q = [0, q)\}$. The public q is used in the generating random large integer k up bound by q . As shown in Fig. 8, this paper uses two of the most frequently invoked arithmetic cells – random large integer generation and point multiplication $[k]P$ to illustrate how a cell interacts with its peer, how the offline and online part are split in original code base, and how the shared memory space is organized, given the parallelism environment as G available GPU devices, each device runs b blocks, and there are n threads per block.

The random large integer generation adopts the *Marsaglia & Zaman random number generator* [43, 44], where $mod\ q$ is involved. According with [45], $mod\ q$ needs to compute q 's division normalized form. Since q is fixed, q 's division normalized form can be pre-computed. So does the sliding window table $\{(m')^P\}$ ($m' = 3, 5, 7, \dots, 15$) for signed m' -ary sliding window point multiplication algorithm. By doing so, redundant computing is removed, and the consuming of shared memory at runtime is reduced. There are two types of data in the data flow; one is the R-only data that is constant and shared by all threads, the other type is RW data, which is owned by each thread. The data flow follows a three-tier dispersion model. The row on the top represents initial value and expected computing result in memory of the host sub-system. When it is moved to global

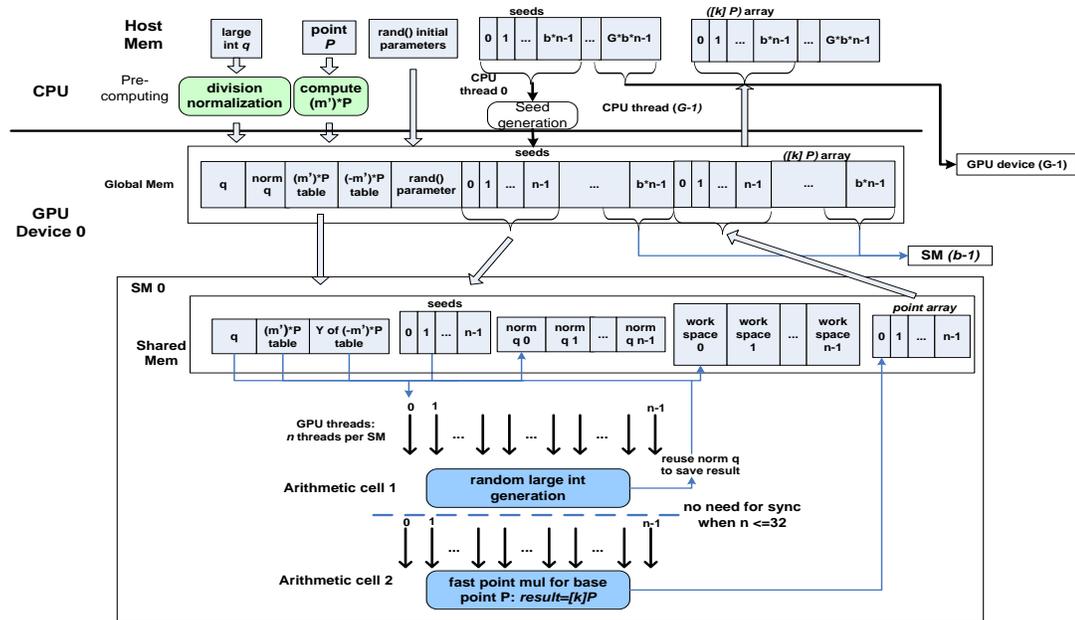


Figure 8. The online data flow of EAGL's arithmetic cell rand+pt mul

memory, each RW data is split into G equal pieces, each piece is assigned to one GPU device, as shown in the middle row in Fig. 8. Then, the RW data in the middle row is divided into n equal parts, each part is copied to one block owns by a SM on the GPU device. When $n \leq 32$, all threads are in the 1st warp, and concurrently runs the same cell, which leads to no need of thread-level synchronization. Overall, the parallelism on each GPU device = $b*n$. Denoted the size of shared memory consumed by each thread as x , then n equals to $\lfloor (SM's\ shared\ memory\ size)/x \rfloor$. By minimizing the temporary variable saving in the work space and reducing the space waste in point and integer structure, most of EACs support the configuration of a full warp as the size of shared memory per SM = 16384 Bytes, so that the SM is fully loaded in these EACs.

However, $[k]P$ is not the only type of point multiplication utilized in the ECC 1-out-of-2 OT protocol. As part of the homomorphic operations, the sender side needs to compute $[r_0]PK_y$ and $[r_1]PK_{1-y}$ and the chooser side needs to compute $[k]([r_0]P)$ or $[k]([r_1]P)$. In any one of these four point multiplications, each thread's curve points $PK_y, PK_{1-y}, ([r_0]P)$ or $([r_1]P)$ are different, and thus each thread needs to compute the sliding window table of these point and maintain a unique table for each point online. In these four point multiplications, we found that it is impossible to save all threads' sliding window tables in the shared memory. There are two alternative solutions to solve this problem: either reducing the number of thread per block from n to $n/4$ so that all the sliding window tables can still be saved in the shared memory, or putting the slide window tables $[m']P$ in global memory and loads the specific table entry into the shared memory when necessary so that n keeps to be the size of a full warp. The latter one is preferred because it saves the time of launching and exiting kernel function, synchronization in EAC. And Fastplay selects the latter solution in $[r_0]PK_\sigma, [r_1]PK_{1-\sigma}, [k]([r_0]P)$ and $[k]([r_1]P)$ to keep the parallelism. However, we choose the former solution in computation of PK_σ 's, $PK_{1-\sigma}$'s, $([r_0]P)$'s and $([r_1]P)$'s sliding window tables because there would be large number of if-else branches if the form solution is adopted.

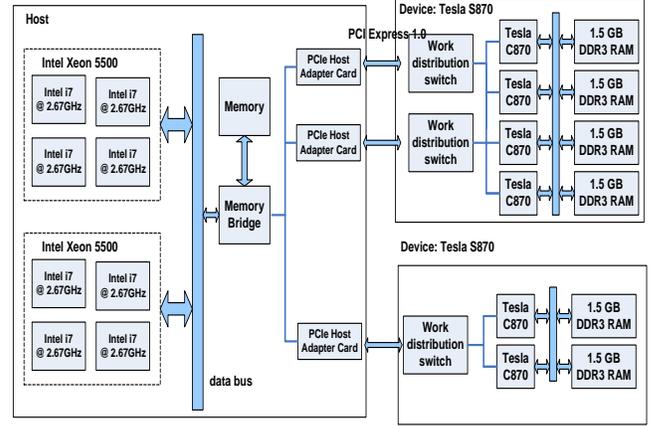


Figure 9. The experiment environment

And this reduction on parallelism degree is hidden for GPU execution control interfaces.

VI. PERFORMANCE EVALUATION

The elliptic curve E used in the experiment is $E: y^2 = x^3 + ax + b \pmod p$, where $a = -3, b = 157$. The large prime modular p and the order of the curve q are:

$$p = 1461501637330897725906826297907101233233312874497, \text{ and}$$

$$q = 1461501637330897725906824301401491257823677986243, \text{ respectively.}$$

The affine coordinate (x, y) of the base point P is $x = 1285014020286381351588483375298861384381400195358$, and $y = 496848534033173203241698588902391838420626211936$, respectively.

The parameters are chosen to construct an ordinary elliptic curve with sufficient protection strength of ECDLP. We do not choose other elliptic curves in MIRACL [5] because they are either super-singular elliptic curves, which are known to have some security weakness [21][22], or have larger parameters.

The experiment platform is illustrated in Fig. 9. The host

Table 1. Execution time (unit: ms) of GGFs (sender side)

GGFs used & input circuit	(1) key gen	(2) GCID	(3) ENC			(4) OT		
			DMA	KPE	Sum	INIT	ONLINE	Sum
GPU GGFs + equality check	133146	2328.9	3863.0	2996.7	6859.8	48190.4	24414.1	72604.5
Multicore GGFs+ eq check	1304.6	2.4	4790.0			47764.3	24323.2	72087.5
serial 3-DES GGFs+OT+eqcheck	24201	--	12333.8			2588.3*10 ³		
GPU-based GGFs & min of 2	23008.9	369.1	687.1	521.2	1208.3	12177.0	5388.8	17565.8
multicore-based GGFs & min of 2	267.5	3.5	838.8			12269.0	5398.4	17667.4
serial 3-DES GGFs+OT&minof2	4699	--	2017.3			612.1*10 ³		

sub-system has two Intel Xeon 5500 quad-core processors with 12GB host memory. The device sub-system is consisted of one and a half Tesla S870 cards that contain 6 Tesla C870 units, or $G=6$. Each Tesla C870 has 16 streaming multiprocessors (SM), which support concurrent running of 16 *blocks* of threads ($b=16$). Each SM has 16k bytes of shared memory. Every two C870 chips share a PCI-Express v1.0 to communicate with the host.

Let the *native parallelism degree* (n -degree) denote the maximum number of thread that a GPU cluster can run concurrently. N -degree is equal to $6*16*32=3072$ in our experiment. Let the *actual parallelism degree* (a -degree) denote the number of actual threads assigned to be run on the cluster. In other words, a -degree = $G' * b' * n'$, where G' is the actual number of GPU device utilized, b' the actual number of blocks, and n' the actual number of threads per block. Given $G' = G$, when a -degree is greater than n -degree, context switching occurs in order for a device run multiple tasks.

As for the multicore-based GGFs/GDFs on the host, the degree of parallelism is 6.

A. Execution Time and Bandwidth

The goal of the first experiment is to evaluate the acceleration benefits of GPU cluster based SMC parallelization. For the multicore implementation, we built a serial version of 3-DES based GGFs/GDFs, and ECC-based OT protocol. The program was tested on a system with 2.53GHz duo core CPU and 4GB memory. All the ECC-based computations in this serial program are based on the MIRACL [5]. The a -degree is made equal to n -degree in all GPU based experiments. Due to the simplicity of the 3-DES operations, and its minor computing costs, we did not implement it on GPU, but run it on the multicore host for both the sender side, and the chooser side. Effectiveness on acceleration of OT by GPU is the primary design concern.

The circuits used in this experiment are the *equality check* circuit with 32-bit from each party, which has 190 gates, and the *minimum of 2* circuit with 8-bit from each party, which has 30 gates. The gate net-lists are generated by

SFDL compiler. In the experiment, 3072 *equality check* circuits and 3072 *minimum of 2* circuits are parallel executed

The results shown in the OT column of Table 1 and Table 2 show that the speed up on the GPU-based version is generally achieves 35~40 times of acceleration over the serial version, with both the INIT (which was often excluded as off-line preparation in some evaluations) and ONLINE computations included. Details on breakdowns of different computing elements are discussed next.

The execution time for each sub-task in sender side and chooser side is shown in Table 1 and Table 2. Each reported result is the average of 20 runs. More details on four types of measurements in Table 1 are discussed follows:

(1) key $k_s^0, k_s^1, k_o^0, k_o^1$ generation (*key gen*) for all gates in the gates net-list. These keys are ECC points in EGG protocol and 160-bit data for 3-DGG protocol. This part is traditionally counted as the *initialization phase* of gate garbling protocol. Key generation has the highest computing cost, when points are used as keys based on point multiplication. For each gate, GPU-based GGF needs to construct at least two random points k_o^0 and k_o^1 , and may also need to construct k_s^0 and k_s^1 . Our experiments showed that the average execution time of point multiplication is around 280~310ms, but the point addition and doubling usually takes several milliseconds.

(2) The *GPU context initialization and destroying (GCID)* when GGFs/GDFs re-enter and leave the GPU execution environment. Multiple entering and leaving of GPU by 1-out-of-2 OT instances are accounted for separately.

(3) Encryption (*ENC*) in GPU-based GGFs, i.e., sender's S2 step in Fig. 4. This part is traditionally counted as a part of the online phase in gate garbling protocol. It is further split into (3-1) device memory allocation (*DMA*), and (3-2) kernel program execution and host/device memory updating via PCI-E (*KPE*). The running time for ENC grows linearly with the circuit size, with the DMA overheads and the KPE actual computations evenly divided the total execution time. As a result, the encryption scheme in GPU-based GGFs may not outperform multicore-based GGFs in certain system

Table 2. Execution time (unit: ms) of GDFs (chooser side)

GGFs used & input circuit	(1) GCID	(2) DEC			(3) OT		
		DMA	KPE	Sum	INIT	ONLINE	Sum
GPU GGFs + equality check	1336.9	10792.9	5136.3	15929.2	9846.2	12623.8	22470.0
Multicore GGFs+ eq check	--	4834.5			9896.4	12288.5	22184.9
serial 3-DES GGFs+OT+eqcheck	--	12392.3			746.9*10 ³		
GPU-based GGFs & min of 2	163.441	1684.9	925.5	2610.4	2412.4	3215.4	5627.8
multicore-based GGFs & min of 2	--	904.9			2416.9	3100.8	5517.7
serial 3-DES GGFs+OT&minof2	--	2025.7			170.5*10 ³		

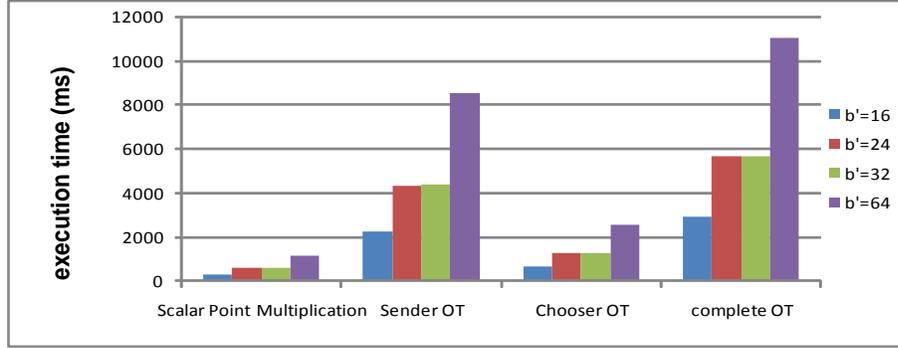


Figure 10. The execution time of point multiplication and 1-out-of-2 OT with various b'

configurations, because of the substantial DMA overheads.

(4) The sender side's 1-out-of-2 OT instance execution phase (*OT*) invoked by all GPU-based GGFs when necessary. *OT part* is divided into two sub-parts -- *INIT* sub-part that generates key $k_c^0, k_c^1, [r_0]P, [r_1]P, [s]P$, and the *ONLINE* sub-part that performs the rest in step OT-S2.

For this part of experiments, a total number of 3072×32 1-out-of-2 OT instances are executed for an *equality check* circuit, and a total number of 3072×8 1-out-of-2 OT for a *min of 2* circuit. The OT time in *min of 2* is three to four times smaller than that in *equality check*. A large proportion of time is occupied by *INIT*. The average execution time of one ECC-based 1-out-of-2 OT instance is 2226.75ms, of which around 1470ms are spent in generating random points $[l_c^0]P, [l_c^1]P, [r_0]P, [r_1]P, [s]P$.

The chooser side exhibited similar performance characteristics as the sender side. The three parts of measurements report in Table 2 are summarized as follows:

(1) *GCID* is similar to that of the sender side. It has relatively small running time. (2) Decryption (*DEC*) in GPU-based GDFs, or chooser's C3 step in Fig. 4. Similarly, this part can be divided into *DMA* and *KPE* as in the sender side; The DEC time grows linearly with the circuit size. (3) The chooser side's 1-out-of-2 OT instance execution phase (*OT part*) invoked by all GPU-based GDFs when necessary. Generation of $[k]P$ in step OT-C1 is counted as *INIT*, and the rests as *ONLINE* in OT related evaluation.

There are several ways to optimize the cost of key generation phase. For example, if the *n-degree* is much larger than the number of circuit simultaneously requested to execute, then generating k_c^0, k_c^1 in sender side's OT does not fully utilized the parallelism, and thus, the rest of random

key generated in this phase can be assigned to k_s^0, k_s^1, k_o^0 , or k_o^1 . Some other methods are like maintaining a large pool of random key for $k_s^0, k_s^1, k_o^0, k_o^1$ if it is allowed, or generating $k_s^0, k_s^1, k_o^0, k_o^1$ for first several gates in the circuit so that the SFE process can start, and generating them for the rest of gates when GPU platform is free, such as the interval of data communication between two parties. And the multicore-based GGFs/GDFs are an alternative heuristic for the cases when these optimization methods are infeasible.

Table 3 illustrates the bandwidth of data communication generated by EGG, 3-DGG, and ECC-based 1-out-of-2 OT protocols respectively in this experiment. Comparing with EGG protocol, 3-DGG protocol needs fewer bandwidths because of shorter truth table entries. In the basic 1-out-of-2 OT protocol presented in [3][42], the size of data communication between sender and chooser is 5×1024 bit, given the fact that DLP requires 1024 bit group element for maintaining sufficient security. For comparison, that in the ECC-based 1-out-of-2 OT protocol is 5×320 bit. As a result, the bandwidth of this part is cut to about 1/3 of the previous OT implementation by utilizing ECC. The bandwidth would be further decreased by applying point compression technique that represents a point in the form of its X coordinate only, and computes its Y coordinate on the fly.

B. Overloading Effect

This experiment is going to explore the effect on the execution time of various parts when *a-degree* exceeds the *n-degree*. Similar to previous experiment, the execution time of OT is also concerned in this experiment.

Given the formula $a-degree = G * b' * n'$, the 1st term is limited by G , and the augmentation of the 3rd term is limited by the size of shared memory in each SM. In all point multiplications used in OT, the shared memory is almost fully utilized when n' is set as the size of one full warp, and the potential gap of increasing this term is small. When the size of threads per block is greater than 32 and smaller than 64, all the newly increased threads are scheduled into the 2nd warp, which cannot concurrently run with the 1st one. Increasing the 3rd term brings trivial augmentation on *a-degree* in this experiment. Hence, only the 2nd term will be enlarged to various values greater than b ($b=16$). Subsequently, certain SMs are assigned to more than one

Table 3. Bandwidth (unit: MB) of EGG, 3-DGG and ECC-based 1/2-OT protocols

bandwidth	ECC 1/2-OT	EGG	3-DGG
equality check	22.5	133.6	60.6
min of 2	5.6	20.5	9.7

block. It is not clear whether the performance benchmark will be hurt to overheads of block switching or be improved because of the reduction on times of kernel launching/exiting and the augmentation of the *a-degree*. It is unknown which factor plays a heavier role. We tests three various cases of overloading by increasing *number of blocks* from 16 to 24, 32, and 64, and the *a-degree* grows from 3072 to 4068, 6144, and 12288.

Fig. 10 illustrates the effect on execution time. When b' increases from 16 to 24, the execution time is nearly doubled. It is because that the newly increased 8 blocks are suspended until first 16 blocks completes. When b' increases from 16 to 32, the newly created 16 blocks can be executed on 16 SMs simultaneously. As a result, enlarging b' from 24 to 32 does not bring significant growth on execution time. The experiment results illustrate when the *a-degree* is doubled by doubling b' , the corresponding execution time of a full ECC-based 1-out-of-2 OT instance grows 1.93 times, and when the *number of blocks* is enlarged 4 times, the execution time of a full ECC-based 1-out-of-2 OT instance grows 3.78 times. The low overhead on system overloading further proves that speeding up by enlarging the *a-degree* is highly efficient.

VII. RELATED WORK

The idea of oblivious transfer was first proposed in [18]. In its initial form [18] the sender sent a message to the chooser with probability 1/2, and he does not know whether or not the chooser received the message. A more useful form known as 1-out-of-2 was proposed in [3][17]. It allowed a chooser to choose one secret from any two secrets generated by a sender, without knowing the other one. During the process the sender could not know which secret the chooser chose. It was then generalized to 1-out-of- n oblivious transfer [19], where the chooser got exactly one secret from a sender's n secrets without disclosing which secret was chosen to the sender. A more recent OT protocol was proposed in [20], which was universally composable in the common reference string model.

The idea of secure multi-party computation (SMC) was first proposed by Yao [8] based on the solutions to the Millionaire's Problem. A general secure two-party computation model was later presented in [6]. The idea of SMC was to enable multiple untrusting parties to compute certain common function based on their own private inputs. After a successful secure multi-party computation each party only knows the output of the function based on their inputs and does not know other's inputs. For example, a secure two-party computation protocol enables two parties P_1 and P_2 to compute a function f based on their inputs x and y . During the interaction P_1 and P_2 learn the output $f(x, y)$ but nothing else. The main framework of two-party SMC [6] [9] was to allow one party (sender) to build circuits that can fulfill a function f with inputs from him and the other party (chooser). An important issue in SMC protocols is the classification of adversaries. The two most studied models are the *semi-honest model* (where an adversary follows the protocol but tries to learn more than he should by launching passive attacks like eavesdropping) and the *malicious model*

(where an adversary can behave arbitrarily). It was proven that any probabilistic polynomial time functionality can be securely computed under the semi-honest adversary model [6] and malicious adversary model [10]. Different SMC protocols have been proposed based on different designs. One secure two-party computation protocol proposed in [11] was based on the construction of a single circuit. This solution only needed a constant number of exponentiations per gate of the circuit. It was very efficient when the circuit size is small. The SMC protocol in [15] adopted the cut-and-choose methodology to build circuits. Its computational complexity was linear to the input length with respect to the number of oblivious transfers. Their protocol was then improved in [16] with less computing cost based on decisional Diffie-Hellman (DDH) assumption. The LEGO protocol [12] also followed the cut-and-choose methodology. In their design the sender first sent the chooser many gates, and then the chooser asked the sender to open some of them to check whether they were correctly constructed. Then the two parties interacted to securely combine the separate circuits into a complete one for secure computation. The methods proposed in [13][14] allowed the parties to simulate a virtual multi-party protocol with an honest majority. The cost of the protocols basically consisted of running a semi-honest protocol for computing the multiplication of additive shares. In [38] an OT protocol was also proposed based on ECC. Both our solution and their solution followed the standard framework of the original OT protocol. The difference is that our protocols used point addition to hide the sender's inputs and was fully implemented on GPU with dedicatedly chosen parameters.

VIII. CONCLUSION

In this paper we propose a parallelization model, named as *Fastplay*, for implementation of secure multi-party computation on GPU cluster architecture. More specifically, Fastplay is partitioned into 4 layers: the specification layer adopts SFDL for specification of applications and compilation of an SFDL program into SHDL; the function layer contains the high-level control functions for gate garbling and 1-out-of-2 oblivious transfer; the protocols layer specifies how gate garbling and 1-out-of-2 OT are fulfilled based on ECC arithmetic operations, and the lowest parallel implementation layer provides support for high parallel implementation of ECC arithmetic on CUDA based GPU cluster architecture. The performance benchmarking results illustrate that the ECC-based OT on the GPU cluster is accelerated 35~40 times faster than its serial version. The experiment results further demonstrate that the fine-grained and large scale parallelization architecture is able to achieve significant acceleration improvement for SMC.

REFERENCES

- [1] N. Koblitz, "Elliptic curve cryptosystems." Mathematics of Computation, Vol.48, No.5, pp.203-209, 1987.
- [2] V. Miller, "Use of elliptic curves in cryptography." Proceedings of CRYPTO'85, LNCS 218, pp.417-426, Springer-Verlag, 1986.
- [3] Mihir Bellare, Silvio Micali, "Non-interactive oblivious transfer and applications", In CRYPTO '89, 1989.

- [4] Lawrence C. Washington. *Elliptic Curves: Number Theory and Cryptography*. Published by Chapman & Hall/CRC, 2003.
- [5] Shamus Software Ltd. MIRACL. URL: <http://www.shamus.ie/>
- [6] A. C. Yao, "How to Generate and Exchange Secrets", In 27th FOCS, pages 162–167, 1986.
- [7] Henri Cohen, Gerhard Frey, Roberto Avanzi, "Handbook of Elliptic Curve and Hyperelliptic Curve Cryptography." CRC Press, 2006.
- [8] A. C. Yao, "Protocols for secure computations", In Proceedings of the 23rd Annual IEEE Symposium on Foundations of Computer Science, 1982.
- [9] Y. Lindell and B. Pinkas, "A Proof of Yao's Protocol for Secure Two-Party Computation", Cryptology ePrint Archive, Report 2004/175, 2004.
- [10] O. Goldreich, S. Micali and A. Wigderson, "How to Play any Mental Game – A Completeness Theorem for Protocols with Honest Majority", In 19th STOC, pages 218-229, 1987.
- [11] S. Jarecki and V. Shmatikov, "Efficient Two-Party Secure Computation on Committed Inputs", In EUROCRYPT 2007, Springer-Verlag (LNCS 4515), pages 97-114, 2007.
- [12] J.B. Nielsen and C. Orlandi, "LEGO for Two-Party Secure Computation", In TCC 2009, Springer-Verlag (LNCS 5444), pages 368-386, 2009.
- [13] Y. Ishai, M. Prabhakaran and A. Sahai, "Founding Cryptography on Oblivious Transfer – Efficiently", In CRYPTO 2008, Springer-Verlag (LNCS 5157), pages 572-591, 2008.
- [14] Y. Ishai, M. Prabhakaran and A. Sahai, "Secure Arithmetic Computation with No Honest Majority", In TCC 2009, Springer-Verlag (LNCS 5444), pages 294 -314, 2009.
- [15] Y. Lindell and B. Pinkas, "An Efficient Protocol for Secure Two-Party Computation in the Presence of Malicious Adversaries", In EUROCRYPT 2007, Springer-Verlag (LNCS 4515), pages 52-78, 2007.
- [16] Y. Lindell and B. Pinkas, "Secure Two-Party Computation via Cut-and-Choose Oblivious Transfer", Cryptology ePrint Archive, Report 2010/284.
- [17] S. Even, O. Goldreich and A. Lempel, "A Randomized Protocol for Signing Contracts", In Communications of the ACM, 28(6):637-647, 1985.
- [18] Michael O. Rabin, "How to exchange secrets by oblivious transfer", Technical Report TR-81, Aiken Computation Laboratory, Harvard University, 1981.
- [19] Giovanni Di Crescenzo, Tal Malkin, and Rafail Ostrovsky, "Single Database Private Information Retrieval Implies Oblivious Transfer", In Proceedings of Advances in Cryptology (EUROCRYPT-2000) Springer-Verlag Lecture Notes in Computer Science.
- [20] C. Peikert, V. Vaikuntanathan and B. Waters, "A Framework for Efficient and Composable Oblivious Transfer", In CRYPTO' 08, Springer-Verlag (LNCS 5157), pages 554-571, 2008.
- [21] M. Scott and P. S. L. M. Barreto, "Generating more MNT elliptic curves," Cryptology ePrint Archive, Report 2004/058, 2004.
- [22] D. Page, N. P. Smart and F. Vercauteren, "A comparison of MNT curves and supersingular curves," Cryptology ePrint Archive, Report 2004/165, 2004.
- [23] I. F. Blake, G. Seroussi and N. P. Smart, "Elliptic Curves in Cryptography", Volume 265 of London Mathematical Society Lecture Note Series. Cambridge University Press, 1999.
- [24] D.V. Chudnovsky and G.V.Chudnovsky, "Sequences of numbers generated by addition in formal groups and new primality and factorization tests", Adv. In Appl. Math., 1987.
- [25] G. Reitwiesner. "Binary Arithmetic". Adv. In Comp., 1, 231-308, 1960.
- [26] W.E. Clark and J.J. Liang. "On arithmetic weight for a general radix representation of integers". IEEE Trans. Info. Theory, 19, 823-826, 1973.
- [27] S. Jha, L. Kruger, and V. Shmatikov, "Towards practical privacy for genomic computation", In 2008 IEEE Symposium on Security and Privacy, 2008.
- [28] R. Wang, X. Wang, Z. Li, H. Tang, M. Reiter and Z. Dong, "Privacy-Preserving Genomic Computation Through Program Specialization", CCS'09.
- [29] H. Ringberg, B. Applebaum, M. J. Freedman, M. Caesar, J. Rexford, "Collaborative, Privacy-Preserving Data Aggregation at Scale", available at <http://eprint.iacr.org/2009/180.pdf>.
- [30] M. Naor, B. Pinkas, and R. Sumner, "Privacy preserving auctions and mechanism design", In ACM Conf. on Electronic Commerce, pages 129-139, 1999.
- [31] P. Lincoln, P. Porras and V. Shmatikov, "Privacy-Preserving Sharing and Correlation of Security Alerts", In Proceedings of the 13th USENIX Security Symposium, 2004.
- [32] C. Cachin. "Efficient Private Bidding and Auctions with Oblivious Third Party". In proc. of the 6th ACM conference on Computer and communication security. Page 120–127. Nov. 1999.
- [33] D. Malkhi, N. Nisan, B. Pinkas and Y. Sella, "Fairplay – A Secure Two-Party Computation System", 13th USENIX Security Symposium, pages 287-302, 2004.
- [34] B. Pinkas, T. Schneider, N.P. Smart and S.C. Williams, "Secure Two-Party Computation Is Practical", In ASIACRYPT 2009, Springer-Verlag (LNCS 5912), pages 250-267, 2009.
- [35] M. Blanton. "Empirical Evaluation of Secure Two-Party Computation Models". CERIAS Tech Report 2005-58. 2005.
- [36] P. Bogetoft, D. L. Christensen, I. Damgard, M. Geisler, T. Jakobsen, M. Kroigaard, J.D. Nielsen, J. B. Nielsen, K. Nielsen, J. Pagter, M. Schwartzbach, T. Toft. "Secure Multiparty Computation Goes Live". In proc. of the Financial Cryptography and Data Security: 13th International Conference. Feb. 2009.
- [37] G. Wang, T. Luo, M. Goodrich, W. Du, Z. Zhu. "Bureaucratic Protocols for Secure Two-Party Sorting, Selection, and Permuting". In proc. of the 5th ASIAN ACM Symposium on Information, Computer and Communications Security, Apr. 2010.
- [38] A. Parakh, "Communication Efficient Oblivious Transfer Using Elliptic Curves". In proc. of Conference on Theoretical and Applied Computer Science (TACS'09), Oct. 2009.
- [39] The OpenMP API Specification for Parallel Programming. Available at <http://openmp.org/wp/>.
- [40] N. Gura, A. Patel, A. Wander, H. Eberle, S. C. Shantz. "Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs". In proc. of workshop on Cryptographic Hardware and Embedded Systems 2004 (CHES 2004), Aug. 2004.
- [41] NVIDIA CUDA programming guide 3.2 <http://developer.nvidia.com/page/documentation.html>
- [42] M. Naor, B. Pinkas, "Efficient Oblivious Transfer Protocols". In proc. of the 12th annual ACM-SIAM symposium on Discrete algorithms (SODA '01), Jan., 2001.
- [43] M.D. Maclaren, G. Marsaglia, "Uniform Random Number Generators". In proc. of Journal of the ACM. Vol 12, issue 1, page 83–89. Jan., 1965.
- [44] G. Marsaglia, A. Zaman. "A New Class of Random Number Generators". In proc. of the Annual of Applied Probability. Vol. 1, No. 3 page 462–480. Aug., 1991.
- [45] D. E. Knuth, "The Art of Computer Programming", Volume 2. Addison Wesley, 1998