Greater Than-Strong Conditional Oblivious Transfer multiparty protocol using Graphics Processing Units

Prarthana LakshmaneGowda, Nikhil Vidyadhar Mishrikoti, Axel Y. Rivera

Abstract

Greater Than-Strong Conditional Oblivious Transfer (**GT-SCOT**) is a multiparty protocol used to share data between two parties without revealing any private information. It is used to perform operations like private auctions, bargaining and secure database mining. Due to the large number of iterative operations computed sequentially and the increase in size of the input, the algorithm is computationally intensive. Due to these properties GTCOT can only be used for small message sizes (12-18 bits). In this work, is proposed an implementation of **GT-SCOT** using Graphics Processing Units (**GPU**) to accelerate the operations and be able to work over larger message size. Results show that a speed up of 24x can be achieved for message size of 16 bits in comparison with a CPU. Also, message sizes up to 64 bits can be processed in a small amount of time.

1. Introduction

Secure multiparty computation is an important field of cryptography. It deals with computational systems in which multiple parties wish to jointly compute some value based on individually held secret bits of information, but do not wish to reveal their secrets to one another in the process. For example consider two millionaires, Alice and Bob, want to find out who is richer without revealing to each other the amount of wealth they possess (Yao's millionaire problem)[1].

Construction of secure multiparty protocol has been an interesting area of research in the field of cryptography. The most commonly used functionality for this purpose is Greater Than (**GT**) predicate which solves the inequality $a \geq b$. There are a large number of applications which rely on secure evaluation of **GT** such as auction systems and price negotiations, computing on intervals, secure distributed database mining, etc.

The **GT** predicate has been exploited by several protocols for secure multiparty computation. One such strongly secure protocol is the Greater Than-Strong Conditional Oblivious Transfer protocol (**GT-SCOT**) [2]. **GT- SCOT** uses the **GT** predicate as a condition to obliviously exchange secrets between two users. It can be formally described as follows: consider two participants, receiver *R* and sender *S*, who have private inputs (*x* and *y* respectively) and share a public predicate *Q(*,*)*. The sender *S* possesses two secrets, *s0* & *s1*, and (oblivious to himself) sends *s1* if *Q(x, y)* = 1, else sends *s0*. This is a one round protocol, it means that receiver sends a message to the sender, who replies to him and then outputs the result of the computation.

GT-SCOT is a secure two-party protocol, but is computationally intensive. The sender has to do a series of computations on every bit of the message sent by the receiver. The protocol requires approximately *8.5*n*log(N)* modular multiplications, where *n* is the number of bits in the message sent by the receiver and *N* is the product of two random prime numbers used for encryption of the message. Due to this computation overhead of the algorithm, it has only been used in applications where the number of bits in the message sent by the receiver is very small (around 12-18 bits)[3]. Though **GT-SCOT** can be securely used in applications involving distributed database mining, it is not feasible to perform such tasks due to

the massive computation involved. These jobs contain messages where the number of bits could be around 900-1000[4,5].

The purpose of this project is to develop an implementation of the **GT-SCOT** algorithm using Graphical Processing Units (**GPU**) in order to speed up the computations of the algorithm and use it for longer messages. Using **GPU** to solve the **GT-SCOT** has two advantages. First, the computation for every bit of the encrypted message can be done in parallel, accelerating the operations. The second advantage is that large words can be computed in less time, thereby making it feasible to use **GT-SCOT** for processes that require large messages (i.e. data mining applications).

This report is organized as follows. Section 2 provides a background to understand Oblivious Transfer (2.1), Homomorphic Encryption (2.2) and Greater Than Strong Conditional Oblivious Transfer (**GTSCOT**) Protocol (2.3). In section 3 is presented the implementation details and how everything works together. Section 4 and 5 presents the results from various tests and the conclusion, respectively.

2. Background

2.1 Oblivious transfer

Secure multiparty computations use oblivious transfer (**OT**) protocols. In this protocol, Receiver (R) has a single input X; Sender (S) has two input strings s_0 and s_1. Then S performs certain computations which usually involves solving a predicate and sends the result to *R*. In the end of this execution, *R* just learns the secret of his choice linked to the result of computation, s_x, while sender *S* learns nothing. Formally it can be represented by the following functionality.

$$f_{OT}\big(X, (s_0, s_1)\big) = (s_x, empty\ string) \qquad (1)$$

OT was first introduced by Michael O. Rabin in 1981 and since then several variants and reformulations have been proposed over years.

2.2 Homomorphic Encryption

Homomorphic Encryption is a type of security codification where in a particular algebraic operation on the plaintext is equivalent to another algebraic operation on the cipher text (encrypted value). In other words, any operation on the plaintext can also be depicted on the cipher text and vice-versa. Formally an encryption scheme presented as *E = (RanGen, Enc_pk, Dec_sk)*, where *RanGen* is a random number generated, *Enc_pk* is the encryption function using public key *pk* and *Dec_sk* is the decryption function using secret key *sk*. This scheme is homomorphic if for some operations $\oplus$ and $\otimes$ (defined on possibly different domains), it holds that for all public and secret key pairs, $x \oplus y = Dec\_sk(Enc\_pk(x) \otimes Enc\_pk(y))$.

A scheme is called additively or multiplicatively homomorphic if it is homomorphic with respect to the corresponding operation. Most commonly used algorithms such as RSA and ElGamal are multiplicatively homomorphic. Other schemes such as Paillier[6] and Goldwasser-Micali[7] are additively homomorphic. There is no scheme which is both additively and multiplicatively homomorphic known till date.

This project uses the Paillier encryption scheme to code the message, which is an additively homomorphic encryption scheme with a large message domain or plain text. Also, it is used to encrypt every bit in the message sent by the sender. The Paillier's technique consist of composite residuosity classes, i.e. of degree set to a hard-to-factor number $n = pq$ where p and q are two large prime numbers.

The homomorphism in Pailler works as follows. Properties for addition between two numbers are presented as

$$Dec(\big(Enc(m_1, r_1) \cdot Enc(m_2, r_2)\big) \, mod \, n^2) = (m_1 + m_2) \, mod \, n \quad (2)$$

$$Dec((Enc(m_1, r_1) \cdot g^{m_2}) \, mod \, n^2) = (m_1 + m_2) \, mod \, n \quad (3)$$

Equation (2) express the addition when message 1 (m_1) and message 2 (m_2) are encrypted. In equation (3), it used when m_1 is encrypted and m_2 is just plain text (not encrypted value). Here g is used, where it is $n+1$. On the other hand, multiplicative properties are represented as

$$Dec(Enc(m_1, r_1)^{m2} \, mod \, n^2) = (m_1 \cdot m_2) \, mod \, n \quad (4)$$

$$Dec(Enc(m_1, r_1)^{k} \, mod \, n^2) = (k m_1) \, mod \, n \quad (5)$$

For equation (4) and (5), m_1 is cipher text. The only variation is that equation 4 is used when the multiplication is done between m_1 and m_2 (plain text), while equation (5) is for m_1 and k (constant). Also, for all four properties, r_1 and r_2 are random numbers, functions Dec means decrypt and Enc means encrypt.

2.3 Greater Than Strong Conditional Oblivious Transfer (GT-SCOT) Protocol

As mentioned above the Oblivious Transfer protocol returns the result of secure evaluation of a predicate. The most commonly used functionality for this purpose is Greater Than (**GT**) predicate which solves the inequality $a \geq b$. **GT-SCOT** uses the **GT** predicate as a condition to obliviously exchange secrets between two users.

To compute **GT-SCOT**, as a part of the receiver's first move, it sets up the Paillier's encryption scheme with group size $N=p*q$, where p& q are prime numbers and $gcd(pq, (p-1)(q-1))$ is 1. For this property to be assured p& q should be of equal bit length. The computation of **GT-SCOT** is as shown below (consider R as receiver, S as sender and x as the message):

1. R runs the setup phase, then encrypts each bit xi of x with the generated public key pk and sends (pk, $Enc(x1), ..., Enc(xn)$) to S.

2. S computes the following, for each i =1 to n:

(a) an encryption of the difference vector d, where $d_i = x_i - y_i$.
(b) an encryption of the flag vector f, where $f_i = x_i$ **XOR** $y_i = (x_i - y_i)^2 = x_i - 2*x_i*y_i + y_i$.
(c) an encryption of vector γ, where $\gamma_0 = 0$ and $\gamma_i = 2\gamma_i - 1 + f_i$.
(d) an encryption of vector δ, where $\delta_i = d_i + r_i(\gamma_i - 1)$, where $r_i \in_R \mathbf{Z}_N$.
(e) a random encryption of vector μ, where $\mu_i = ((s_1 - s_0)/2) * \delta_i + ((s_1 + s_0)/2)$ and sends a random

permutation π(*Enc(μ)*) to *R*.

3. *R* obtains π(*Enc(μ)*), decrypts it, and determines the output as follows: if μ contains a single $v \in D_S$, output *v*, otherwise abort.

Each operation in the algorithm requires to be transformed using the homomorphic scheme explained in 2.2. Since these transformations are performed over large integers, the **GTSCOT** protocol becomes computation intensive. Also, few steps include the computation of modular inverse and exponentiation using large numbers. In other words, **GTSCOT** becomes slower when the size of message *X* increases.

3. Implementation

3.1 Extended Euclidean Algorithm

Computing the **GT-SCOT** protocol introduces in many occasions the problem of calculating Multiplicative Modular Inverse (**MMI**) of various intermediate values. One can obtain **MMI** of a number by expressing the modular relation between the number and its inverse as a linear equation and solving it using Extended Euclidean Algorithm (**EEA**). The **EEA** solve the linear expression showed in equation (6).

$$ax + by = gcd(a, b) \tag{6}$$

This small example presents how to find the **MMI** between two numbers. The inverse of $g = (n + 1)$ can be obtained, where g and n are constants used in Paillier's encryption scheme, by the following expressions. Let x denote the **MMI** of$g$,

$$x = g^{-1}\ mod\ n^2 \tag{7}$$

Therefore,

$$(gx)\ mod\ n^2 = 1 \tag{8}$$

$$(n + 1)x\ mod\ n^2 = 1 \tag{9}$$

The modular relation can be expressed as,

$$(n + 1)x - 1 = qn^2 \tag{10}$$

where q is some quotient obtained after the division of $(n + 1)x$ by n^2**.** Now equation (9) can be rewritten as,

$$(n + 1)x - qn^2 = 1 \tag{11}$$

Since, $gcd\big((n + 1), n^2\big) = 1$, then equation (10) can be expressed in terms of equation (6) as,

$$(n + 1)x - qn^2 = gcd\big((n + 1), n^2\big) \tag{12}$$

Finally, equation (12) can be solved for x using the **EEA** shown in Table 1

```
function extended_euclid(a, b)
   x := 0    lastx := 1
   y := 1    lasty := 0
   while b ≠ 0
            quotient := a div b
            {a, b} = {b, a mod b}
            {x, lastx} = {lastx - quotient*x, x}
            {y, lasty} = {lasty - quotient*y, y}
   return {lastx, lasty, a}
```

Table 1: Extended Euclid Algorithm (source : Wikipedia)

3.2 Modular Exponentiation:

Like **MMI**, other steps introduce the problem of computing the exponents of very large intermediate results. Due to the size of these numbers, a naive implementation like iterative multiplication to compute exponents consumes a lot of time and memory, making it unfeasible. Instead, it was implemented using an efficient algorithm called "**Right-to-Left Binary Exponentiation**".

In this algorithm the exponent is converted into a binary string and then the exponentiation is expressed as a modular expression by looking at the exponent bits from right to left. Consider that it is needed to calculate the exponent $c \equiv b^e mod\ m$ then,

$$e = \sum_{i=0}^{k-1} a_i 2^i \tag{13}$$

where a_i is either 0 or 1. Hence,

$$b^e = b^{(\sum_{i=0}^{k-1} a_i 2^i)} = \prod_{i=0}^{k-1} \left(b^{2^i}\right)^{a_i} \tag{14}$$

then, the modular exponent can be expressed as

$$c \equiv \prod_{i=0}^{k-1} \left(b^{2^i}\right)^{a_i} mod\ m \tag{15}$$

Table 2 shows the algorithm for calculating modular exponent. In table 3, a small illustration of the algorithm in action is presented

```
function modular_pow(base, exponent, modulus)
    result := 1
    while exponent > 0
            if (exponent & 1) equals 1:
                result = (result * base) mod modulus
            exponent := exponent >> 1
            base = (base * base) mod modulus
    return result
```

Table 2: Right-to-Left Binary Exponentiation Algorithm(source : Wikipedia)

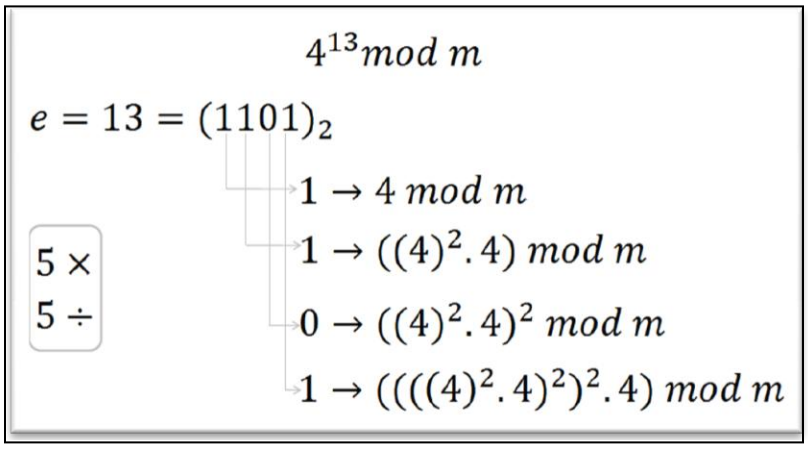

Table 3: Modular Exponentiation Algorithm example

As can be seen from the above example, the number of multiplications and divisions **rises linearly with the bit-length of the exponent** rather than the value of the exponent itself.

3.3 Big Integer Class

Large integers are the key for internet and other computational security. These numbers are needed in all encryption schemes. As intended, it is desired to encrypt information using keys as large as possible.

The GPU programming model, CUDA, accepts numbers as the form **unsigned long long** (up to $2^{64} - 1$ bits) data type. If the input values are as big as the limit, then during intermediate computations, these numbers will keep increasing and eventually will cause an over flow. In other words, even though it is possible to reserve 64 bits for encryption, this size cannot be used to perform all computations.

To corroborate the statement of overflow, some tests were performed. Results showed that the biggest value that did not cause overflow using this data type was 16 bits. This value is small and an intrusion using brute force can break it and decipher the encrypted values. To prevent this, we implemented a class to handle big integers.

The big integer class (called BigInt) creates an array where each entry represents a digit of the whole number. Once the number is defined, the ordinary binary operations can be performed. The arithmetic operations supported are summation, subtraction, multiplication, division and modulo. For comparison operations, the BigInt class has equal and greater than functions.

Using BigInt class we are now able to compute **GT-SCOT** protocol on large numbers. Results from various tests showed that the maximum size that could be handled without causing any problem in the GPU was 64 bits. This means that security was increased from 16 to 64 bits using this new class.

3.4 GPU Implementation

3.4.1 Parallelization

The **GTSCOT** algorithm (2.3) presents a very straight forward parallelization. The implementation of it in GPU is defined as follows. Each thread will take care of each encrypted X_i bit. Figure 1 shows a representation of the implementation. Here T_i means i^{th} - thread and X_i implies each bit of message X. It is clear to see that the number of threads needed depends on the numbers of bits.

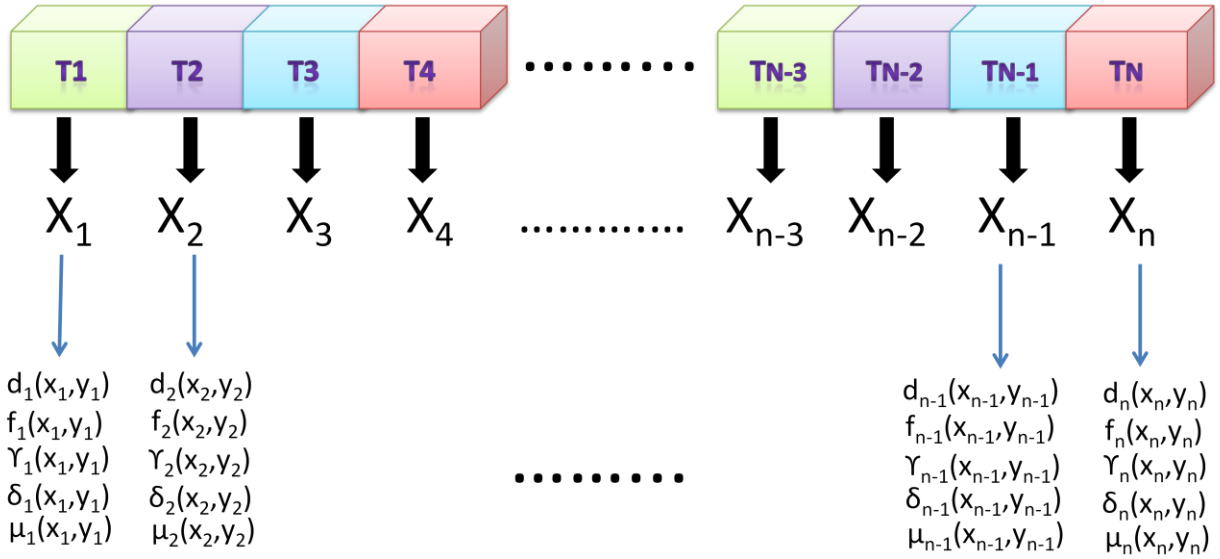

Figure 1: Representation of GTSCOT in a GPU

3.4.2 Kernel calls

Since one GPU thread is lighter than a CPU thread, it is important to take into consideration the amount of computation and kernel calls that can be done. Due to the implementation of the Big Integer class, most of the calculations deal with large numbers. This computation can take some time and might cause a kernel time out. To prevent this problem, **GTSCOT** is split into several kernel calls.

One kernel is called for computing step d_i while the step f_i is split into two calls. The first kernel is to compute the inverse needed in the part $(x_i - 2x_i)$ using the Euclidean Extended Algorithm. The second one is used to finish rest of the computation in equation $(x_i - 2x_i y_i + y_i)$ for f_i .

The step γ_i should be handled with care because this is where most of the computationally heavy part resides. This step contains a dependence since $\gamma_i = 2\gamma_{i-1} + f_i$. To solve it, γ_i can be replaced with equation (16)

$$\gamma_i = \sum_{j=1}^{i} 2^{i-j}.f_j \quad (16)$$

This solution introduces a loop nest since the Binary Method for Modular Exponentiation is needed. Also, the last thread needs to travel through all values on 2^{i-j}. This loop nest introduces the problem that a kernel gets engaged for a long time computing the values and might eventually time out. To prevent it, the synchronization for the summation in Equation (16) is placed on the host. In other words, the computation for γ_i is split in two parts. First, in one kernel call all values are initialized. Then, a kernel that computes $\left(2^{i-j}.f_i\right)$ is called inside an iteration that goes from 1 to the size of message X on the host side. Even though this solution is not very efficient, it allows performing **GTSCOT** using an encryption up to 64 bits.

The steps of δ_i and μ_i have been taken care using the same strategy as for d_i and f_i. One kernel call computes δ_i, mean while μ_i is split into two kernels. The first one is to solve the inverse between $\delta_i$and n^2. The second call finishes the rest of computation for μ_i.

In total, $\alpha + 7$ kernel calls are required to compute **GTSCOT**, where α is the size of message X. This implementation works fast enough for at max a message of 64 bits (see results).

3.4.3 Wrapping everything

In order to verify our results, the newly developed **GTSCOT** was tested with an already created Pailler's encryption scheme (credits to Mike Clark for this implementation). This implementation uses Java and also has a GTSCOT implementation to test. The related files were replaced with the newly developed ones.

A test case works as follow. First, messages X and Y are the inputs. Also the size of the encryption is added. There is something important while choosing the size of the private key. This number should be the desired size over 2 (e.g. 64 bit encryption requires 32 bit private key size). This is because, using the mentioned example, Pailler's will use two 32 bits prime numbers (p & q) to create n. The size of n will be 32, but then since we are computing n^2, this value becomes 64 bits. Then, n^2 is used for encryption of each bit of message X. Due to implementations constrains, at this stage of our project the biggest cipher text size can be up to 64 bits.

After the initial values are set, in the Java part, each bit of message X is encrypted. When the encryption is done, the encrypted message X, Y in binary form, the value 0 but encrypted (needed for Υ_0) and the random numbers produced that are required in δ_i are passed to CUDA. These values are used to compute GT-SCOT on the GPU as explained in 3.4.1 and 3.4.2. Once μ_i is created, it is passed back to Java, where decryption is performed and decided if S_1 or S_0 is returned.

4. Results

4.1 Configuration

The implemented algorithms were tested on CPU and GPU. The CPU version was developed in C++. As for GPU, two versions were created using CUDA. The fist implementation uses only global memory. Mean while, the second one uses memory optimizations such as registers and constant memory. The resources used were:

- GPU: NVIDIA Tesla C2050, 2 GBs on device memory
- CPU: Intel Core i7 930 2.80 GHz , L2 Cache 4 x 256 KB, L3 Cache 8 MB
- DRAM: 4 GBs
- Operating System: Ubuntu 10.04 64 bits, kernel 2.6.32-27
- CUDA: 3.2 64 bits
- G++: 4.3.4

Currently **GTSCOT**, using a cipher text of 64 bits, only works in Fermi architecture (only tested on Tesla, not on 400 GTX generations). Using the older architecture (tested on 260 GTX) caused a kernel time out. This cause has various reasons, but the main problem is due to the BigInt class implementation. Also, the cache memory in the Fermi architecture helps to reduce the transfer overhead from Global memory to threads. Lower size of cipher text was tested on 260 GTX and worked fine (up to 32 bits).

To perform the tests, the configuration used was a cipher text of 64 bits and the size of message X and Y was scaled from 8 bits to 64 bits. The cipher text size was decided as explained in the implementation details 3.3. The time measurements were performed over the computational part only; it doesn't contain the calculation of the copy overhead. Also, since **GTSCOT** depends on random numbers, the time cost might vary for different runs even if the value of X is unchanged. So, the measurements are an average of five runs for each size of X.

4.2 Speed up

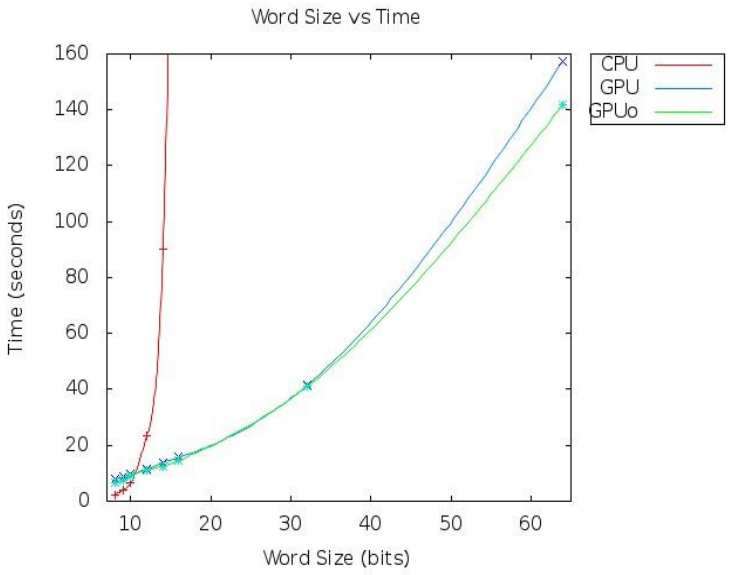

Figure 2: Time cost of computation with increasing message size. Red plot is for CPU, Blue for regular GPU and green for GPU optimized GPU

Size of message X	CPU time (seconds)	GPU time(sseconds)	GPUo (optimized) time(sseconds)
8 bits	2.414536	7.9176 (-3.279x)	6.84809 (-2.836x)
10 its	6.57095	9.52458 (-1.449x)	9.0334 (-1.375x)
12 bits	23.46762	11.5387 (2.034x)	11.0679 (2.12x)
14 bits	90.1756	13.8553 (6.51x)	12.531 (7.20x)
16 bits	356.379	15.7294 (22.66x)	14.6626 (24.30x)
32 bits	-	41.3957	41.0143
64 bits	-	157.397	141.775

Table 4: Speed up difference with increase in size of X. 32 and 64 bits not computed in CPU due to time constrains

It is notable in Figure 2 and in Table 4 that as the message size increases, the time to compute GTSCOT increases. Now, comparing the CPU version with the GPU, there is a difference among both. For message sizes less than 12 bits, it is faster to use CPU. But at 16 bits, the CPU took up to 356.379 seconds, while the GPU computed in 15.7294 seconds. Also, the GPU can compute sizes up to 64 bits using a short amount of time.

Among both the GPU implementations there is a difference too. As expected, the optimized version runs faster than the one that uses global memory. When message sizes are of 64 bits, up to 15.622 seconds can be achieved. In other words, while the length of the words increase, the time difference between both implementations increases, which is expected.

5. Conclusion

The implementation of **GTSCOT** protocol on the GPU produces a notable speed up compared to the CPU version. The naive GPU development itself produced good results and the GPU code with memory hierarchy optimizations provided better acceleration. The GPU implementation works well for message sizes up to 64 bits with the cipher text size of 64 bits. This is highly better than CPU version, since it is futile to go beyond the message size of 16. In general it is observed that GPU implementation of **GTSCOT** not only helps to accelerate the computations involved in the protocol but also allows the use of **GTSCOT** on higher message sizes.

6. Future Work

There is plenty of room for improvement in the developed tools. As for future work, it is planned to optimize the BigInt class. The modifications discussed at this moment are to implement a way to

represent the numbers using less memory and use Fast Fourier Transformations to perform the multiplication (right now multiplication is done by brute force). Improving these two parts will create a speed up in the division and modulo computation, since both arithmetic operations are dependant of multiplication.

Also, though the modular exponentiation is fast using the binary method, it is planned to use the Montgomery algorithm to compute mod powers. It is expected that this method will help to speed up the exponentiation. All this improvements are needed to accelerate the computation, especially in the γ_i step, where it is necessary to remove the loop from the host and reduce the number of kernel calls.

7. Acknowledgements

The authors wish to express their appreciation to Mike Clark. He provided the Java code to compute **GTSCOT** and Pailler's. Also Mike gave valuable advice whenever needed.

8. References

1. A. Yao: *Protocols for Secure Computations (Extended Abstract)* **FOCS** 1982: 160-164
2. I. Blake & V. Kolesnikov: *Strong Conditional Oblivious Transfer and Computing on Intervals*, **ASIACRYPT** 2004: 515-529
3. G. Zhong & U. Hengartner*: A Distributed k-Anonymity Protocol for Location Privacy*, **ACM** 2008
4. M. Kantarcioglu & C. Clifton: *Privacy-preserving distributed mining of association rules on horizontally partitioned data*, **DMKD** 2002
5. Y. Lindell & B. Pinkas: *Privacy preserving data mining*, **CRYPTO** 2000: 20–24
6. P. Paillier: *Composite-Residuosity Based Cryptography: An Overview*, **EUROCRYPT** 2002: 223-238
7. S. Goldwasser, S. Micali: *Probabilistic encryption and how to play mental poker keeping secret all partial information*, Proc. 14th Symposium on Theory of Computing 1982: 365-377