# Accelerating Fully Homomorphic Encryption on Graphic Processing Units

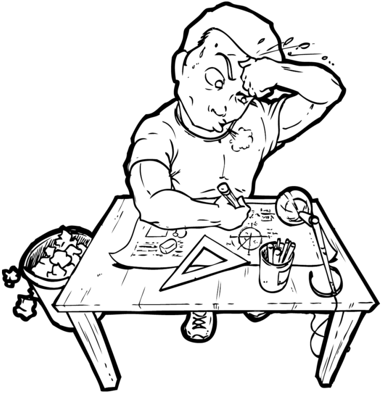

Varun Srivastava    Amit Datta
08CS1001    08CS1045

Guide: Prof. Debdeep Mukhopadhyay

Department of Computer Science and Engineering
IIT Kharagpur

# How it all started?

Amit working on improving time complexity of FHE [1]



[1] Fully Homomorphic Encryption

# How it all started?

Varun working on GPU[1] based acceleration

[1] Graphics Processing Unit

# How it all started?

How about improving the timings of **FHE** by accelerating it on **GPU**?

# How it all started?

How about improving the timings of **FHE** by accelerating it on **GPU**?

We shall show how this will be done...

# Outline

- What is Fully Homomorphic Encryption?
- How do things get accelerated on a GPU?
- Combining the two ideas!

# Part I

## FULLY HOMOMORPHIC ENCRYPTION

# Basic idea of FHE

Stating the goal of FHE, author Craig Gentry said:

> *I want to delegate processing of my data, without giving away access to it*

# Why is so much importance being attached to FHE?

Cloud Computing :

Data is stored on your private Cloud in encrypted form.

In order to perform queries on your data, send an encrypted version $queries$ to Cloud.

Cloud performs $queries$ on encrypted data, gets results and returns them.

You decrypt return values to obtain results.

# Why is so much importance being attached to FHE?

Private Google Search :

You do not want Google to know what you are querying.
Send encrypted queries to Google.
Receive encrypted results.
Decrypt them to obtain your required results.

# FHE schemes

Two existing schemes

**Ideal Lattice Based Scheme** Developed by Gentry [1]. Scheme based on ideal lattices

**Integer Based Scheme** Developed by Dijk et al. [3]. Much simpler scheme based on integers

# The FHE scheme

An encryption scheme $\mathcal{E}$ comprises of the following algorithms:

- $\text{KeyGen}_{\mathcal{E}}$
- $\text{Encrypt}_{\mathcal{E}}$
- $\text{Decrypt}_{\mathcal{E}}$

# The FHE scheme

An encryption scheme $\mathcal{E}$ comprises of the following algorithms:

- $\text{KeyGen}_{\mathcal{E}}$
- $\text{Encrypt}_{\mathcal{E}}$
- $\text{Decrypt}_{\mathcal{E}}$

In addition to them, the FHE has the following algorithms:

- $\text{Evaluate}_{\mathcal{E}}$
- $\text{Recrypt}_{\mathcal{E}}$

# KeyGen$_{\mathcal{E}}$

**Input:** Parameters: #bits-$t$; degree of poly-$N$
**Output:** Public key $pk$ and secret key $sk$

1: Initialize $F \leftarrow x^N + 1$, where $N$ is a power of 2
2: **repeat**
3: Generate a random polynomial $G$ of degree $N - 1$ and $t$-bits
4: Compute $p \leftarrow$ Resultant$(G, F)$
5: **until** $p$ is a prime
6: $F_p \leftarrow F \bmod p$
7: $G_p \leftarrow G \bmod p$
8: $D_p \leftarrow$ poly_gcd$(F_p, G_p)$
9: $Z \leftarrow$ Inverse of $G_p \bmod F_p$

# KeyGen$_{\mathcal{E}}$ continued

// Build public key
$pk.p \leftarrow p$
$pk.\alpha \leftarrow -D_p$.coeff$[0]$
// Build secret key
$sk.p \leftarrow p$
$sk.B \leftarrow Z$.coeff$[0] \bmod 2p$
// Build hint
$B_i \leftarrow B/S2$ // $S1$ - entire set size, $S2$ - subset size in SSSP
$pk.B[0 \cdots (S2-1)] \leftarrow B_i$
$pk.c[0 \cdots (S2-1)] \leftarrow$ Encrypt$_{\mathcal{E}*}(1, pk)$
$pk.B[S2 \cdots (S1-1)] \leftarrow$ random$[-p, +p]$
$pk.c[S2 \cdots (S1-1)] \leftarrow$ Encrypt$_{\mathcal{E}*}(0, pk)$
Add and subtract values to $pk.B[0 \cdots (S2-1)]$, so that the sum remains the same
Shuffle all $pk.B[i]$ values

# Encrypt$_{\mathcal{E}}$

**Input:** Bit $m$; Public key $pk$
**Output:** Integer cipher-text $c$

1: Randomly choose a polynomial $C$ of degree $N-1$ with even coefficients
2: $c \leftarrow C(pk.\alpha) + m \bmod pk.p$

# Decrypt$_{\mathcal{E}}$

**Input:** Cipher $c$; Secret key $sk$
**Output:** Bit $m$
1: $q \leftarrow \lfloor \frac{c * sk.B}{sk.p} \rceil$
2: $m \leftarrow c + q \bmod 2$

# Evaluate$_{\mathcal{E}}$

**Input:** Vector of cipher-texts $\hat{\mathbf{c}}$; Circuit $\mathcal{C}$; Public key $pk$
**Output:** Computed cipher-text $\hat{c}$

1: Modify $\mathcal{C}$ to $\mathcal{C}^{\dagger}$ with boolean AND$(.)$ replaced with integer multiplication $\times$, and boolean EXOR$(^\wedge)$ replaced with integer addition $+$.
2: Convert infix $\mathcal{C}^{\dagger}$ to post-fix $\mathcal{C}^{\dagger\dagger}$
3: Evaluate $\mathcal{C}^{\dagger\dagger}$ plugging in values from $\hat{\mathbf{c}}$ using an implementation with stacks

# Recrypt$_{\mathcal{E}}$

**Input:** Another public key $pk'$; Decryption circuit $D$; Cipher $c$; A vector $\hat{\mathbf{sk}}$, where each $\hat{\mathbf{sk}}[i] \leftarrow$ Encrypt$_{\mathcal{E}}(pk', sk[i])$

**Output:** Refreshed cipher-text $c'$ encrypted under $pk'$

1: Encrypt each bit of $c$ to form a vector $\hat{\mathbf{c}}$, i.e. $\hat{\mathbf{c}}[i] \leftarrow$ Encrypt$_{\mathcal{E}}(pk', c[i])$

2: $c' \leftarrow$ Evaluate$_{\mathcal{E}}(pk', D, \hat{\mathbf{sk}}, \hat{\mathbf{c}})$

# Timing Observations of our Implementation

| $\lambda$ | KeyGen$_{\mathcal{E}}$ | Encrypt$_{\mathcal{E}}$ | Decrypt$_{\mathcal{E}}$ |
|---|---|---|---|
| 9 | 0.195 ms | 0.552 ms | 0.040 ms |
| 11 | 0.199 ms | 0.990 ms | 0.085 ms |
| 13 | 0.193 ms | 2.375 ms | 0.127 ms |
| 15 | 0.197 ms | 4.786 ms | 0.273 ms |

Table: Table showing the variation of the Key Gen, Encryption and Decryption times with the security parameter $\lambda$ on our implementation

# Timing Observations of Gentry's Implementation

| $\lambda$ | KeyGen$_{\mathcal{E}}$ | Encrypt$_{\mathcal{E}}$ | Decrypt$_{\mathcal{E}}$ |
|---|---|---|---|
| 9 | 0.16 sec | 4 ms | 4 ms |
| 11 | 1.25 sec | 60 ms | 23 ms |
| 13 | 10 sec | 0.7 sec | 0.12 sec |
| 15 | 95 sec | 5.3 ms | 0.6 ms |

Table: The same comparisons on Gentry's implementation with lattices

# Is this it?

- Have we already achieved our goal?

# Is this it?

- Have we already achieved our goal?
- NO!
  The $\text{Recrypt}_{\mathcal{E}}$ function has not been implemented.

# Is this it?

- Have we already achieved our goal?
- NO!
  The Recrypt$_{\mathcal{E}}$ function has not been implemented.
- It was too mathematically involved.
  We proceeded to implement the lattice-based scheme, which we assumed at time would provide significant insight to implementing the Recrypt$_{\mathcal{E}}$ function

# Our implementation of the Lattice based scheme

- Development of a polynomial library, which we decided to build ourselves catering to our specific needs.
- Started out with handling polynomials with integer coefficients.
- Which soon became floating point and then complex coefficients.
- Experienced difficulties in computing the inverse of a polynomial modulo another polynomial.

# Our implementation of the Lattice based scheme

- We did succeed (partially) but our KeyGen$_{\mathcal{E}}$ never seemed to terminate.

# Our implementation of the Lattice based scheme

- We did succeed (partially) but our $\text{KeyGen}_{\mathcal{E}}$ never seemed to terminate.

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!Major Disappointment.

# The *Scarab Library*

The *Scarab Library* [2], developed by Perl, Brenner and Smith [2], is a library to demonstrate a working implementation of FHE with integers.

Requirements:

- GMP - GNU Multiple Precision Library
- FLINT - Fast Library for Number Theory
  - MPIR - Multiple Precision Integers and Rationals
  - MPFR - C library for Multiple-Precision Floating-point computations with correct Rounding

---

[2] http://www.hcrypt.com/scarab-library/

# The *Scarab Library*

- It gave us the much needed implementation of the $\text{Recrypt}_{\mathcal{E}}$ function

# The *Scarab Library*

- It gave us the much needed implementation of the $\text{Recrypt}_{\mathcal{E}}$ function
- We developed an extension to this library which enabled it to handle any arbitrary function expressed in AND and EXOR

# Timing Observations

| $bits$ | KeyGen$_{\mathcal{E}^*}$ | Encrypt$_{\mathcal{E}^*}$ | Decrypt$_{\mathcal{E}^*}$ | Recrypt$_{\mathcal{E}}$ |
|---|---|---|---|---|
| 384 | 17.379 sec | 4.742 ms | 0.171 ms | 200.414 ms |
| 512 | 69.761 sec | 4.868 ms | 0.247 ms | 210.767 ms |
| 1024 | 11.65 mins | 14.945 ms | 0.688 ms | 278.591 ms |

Table: Running times of the implementation using *Scarab Library*

# Timing Observations

| This Scheme | | Gentry's Scheme | |
|---|---|---|---|
| $bits$ | $\text{KeyGen}_{\mathcal{E}*}$ | $bits$ | $\text{KeyGen}_{\mathcal{E}*}$ |
| 384 | 17.4 sec | 384 | - |
| 512 | 69.8 sec | 512 | 2.4 sec |
| 1024 | 11.6 mins | 1024 | - |
| 2048 | 1.5 hours | 2048 | 40 sec |
| 8192 | - | 8192 | 8 mins |
| 32768 | - | 32768 | 2 hours |

Table: An explicit comparison between the times required by Gentry's $\text{KeyGen}_{\mathcal{E}*}$ and this implementation

# Timing Observations

| $\#bits = 384$ | | $\#bits = 512$ | | $\#bits = 1024$ | |
|---|---|---|---|---|---|
| $\#runs$ | KeyGen$_{\mathcal{E}*}$ | $\#runs$ | KeyGen$_{\mathcal{E}*}$ | $\#runs$ | KeyGen$_{\mathcal{E}*}$ |
| 61 | 2.35 s | 23 | 3.90 s | 177 | 60.04 s |
| 473 | 10.65 s | 437 | 19.36 s | 1037 | 297.83 s |
| 1469 | 30.86 s | 665 | 26.07 s | 1163 | 318.90 s |
| 2343 | 47.37 s | 1045 | 42.99 s | 1425 | 382.64 s |
| 3101 | 60.96 s | 2317 | 96.68 s | 2569 | 699.57 s |
| 10387 | 202.22 s | 2981 | 122.11 s | 5735 | 1498.60 s |
| avg=19.95 ms | | avg=41.84 ms | | avg=269.09 ms | |

Table: Times required for performing KeyGen$_{\mathcal{E}*}$ and the corresponding number of times primality testing is carried out

Craig Gentry.
Fully homomorphic encryption using ideal lattices.
In Michael Mitzenmacher, editor, *STOC*, pages 169–178. ACM, 2009.

Henning Perl, Michael Brenner, and Matthew Smith.
Poster: an implementation of the fully homomorphic smart-vercauteren crypto-system.
In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 837–840, New York, NY, USA, 2011. ACM.

Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan.
Fully homomorphic encryption over the integers.
Cryptology ePrint Archive, Report 2009/616, 2009.
`http://eprint.iacr.org/`.