Faster Homomorphic Encryption over GPGPUs via hierarchical DGT

Pedro G. M. R. Alves[1] Jheyne N. Ortiz[1] Diego F. Aranha[2]
pedro.alves@ic.unicamp.br

[1]Institute of Computing, University of Campinas
[2]Department of Computer Science, Aarhus University

March 4, 2021

Financial Cryptography and Data Security 2021

Introduction

- Ubiquitous data gathering is here to stay.

Introduction

- Ubiquitous data gathering is here to stay.
- Always a bad thing?

Introduction

- Ubiquitous data gathering is here to stay.
- Always a bad thing?
  - Security,

Introduction

- Ubiquitous data gathering is here to stay.
- Always a bad thing?
  - Security,
  - E-Health,

Introduction

- Ubiquitous data gathering is here to stay.
- Always a bad thing?
  - Security,
  - E-Health,
  - Leisure activities,

Introduction

- Ubiquitous data gathering is here to stay.
- Always a bad thing?
  - Security,
  - E-Health,
  - Leisure activities,
  - Traffic,

Introduction

- Ubiquitous data gathering is here to stay.
- Always a bad thing?
  - Security,
  - E-Health,
  - Leisure activities,
  - Traffic,
  - ...

Introduction

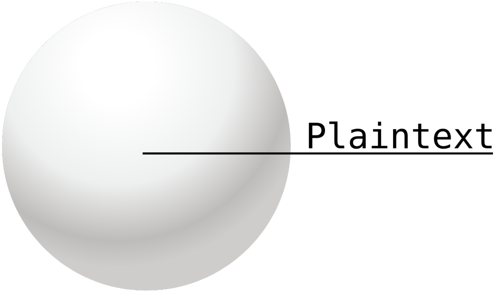
- Ubiquitous data gathering is here to stay.
- Always a bad thing?
  - Security,
  - E-Health,
  - Leisure activities,
  - Traffic,
  - ...

"*Data privacy is a hard problem*"

— Narayanan and Felten, 2014

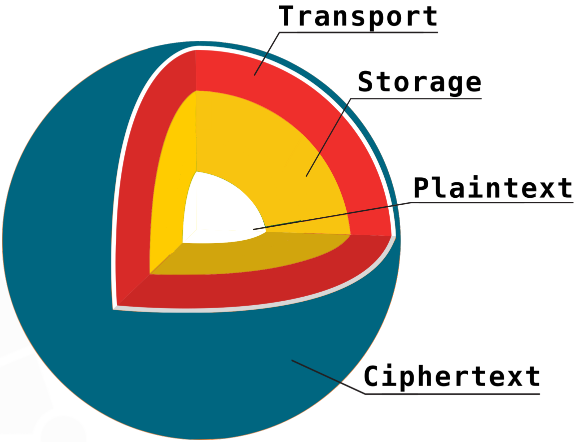
Introduction

Homomorphic Encryption



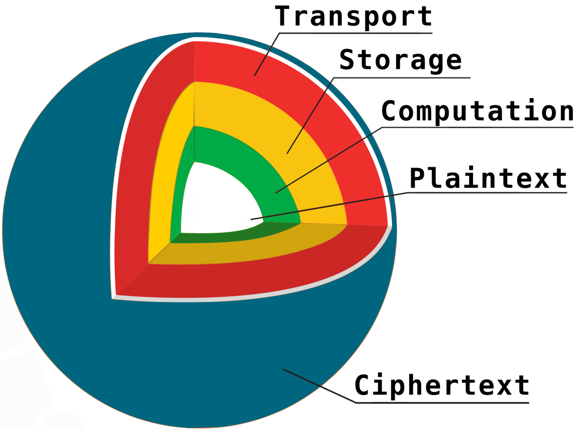
Introduction

Homomorphic Encryption



Introduction

Homomorphic Encryption



Introduction

Homomorphic Encryption

- FHE allows evaluation for addition and multiplication without the need to decrypt.

Introduction

Homomorphic Encryption

- FHE allows evaluation for addition and multiplication without the need to decrypt.
- Standardization efforts are on the way.

Introduction

Homomorphic Encryption

- FHE allows evaluation for addition and multiplication without the need to decrypt.
- Standardization efforts are on the way.
  - There is an open consortium – `homomorphicencryption.org/`

Introduction

Homomorphic Encryption

- FHE allows evaluation for addition and multiplication without the need to decrypt.
- Standardization efforts are on the way.
  - There is an open consortium – `homomorphicencryption.org/`
  - BFV is currently one of the primary schemes.

Introduction

Homomorphic Encryption

- FHE allows evaluation for addition and multiplication without the need to decrypt.
- Standardization efforts are on the way.
  - There is an open consortium – `homomorphicencryption.org/`
  - BFV is currently one of the primary schemes.
- Performance is still a challenge.

Our contributions

- We propose **implementation techniques** for performance enhancement of RLWE-based schemes on **GPUs**.

Our contributions

- We propose **implementation techniques** for performance enhancement of RLWE-based schemes on **GPUs**.
  - **Polynomial multiplication** is a costly operation. DGT can help with that.
    - A divide-and-conquer formulation for the Discrete Galois Transform (**HDGT**).
  - A state machine is described to improve locality.

Our contributions

- We propose **implementation techniques** for performance enhancement of RLWE-based schemes on **GPUs**.
  - **Polynomial multiplication** is a costly operation. DGT can help with that.
    - A divide-and-conquer formulation for the Discrete Galois Transform (**HDGT**).
  - A state machine is described to improve locality.
- A proof-of-concept implementation is compared with state-of-the-art works.

Homomorphic Encryption

Definition

Homomorphic Encryption (HE)

Let

- $\boldsymbol{E}$ and $\boldsymbol{D}$ be a pair of **encryption and decryption** functions,
- m_1 and m_2 be plaintexts.

The pair $(\boldsymbol{E}, \boldsymbol{D})$ forms an **homomorphic encryption scheme** for some operator $\diamond$ if and only if the following holds:

$$\boldsymbol{D}\left(\quad \boldsymbol{E}(m_1) \circ \boldsymbol{E}(m_2) \quad \right) \equiv \boldsymbol{D}\left(\quad \boldsymbol{E}(m_1 \diamond m_2) \quad \right).$$

Homomorphic Encryption

Definition

Homomorphic Encryption (HE)

Let

- $\boldsymbol{E}$ and $\boldsymbol{D}$ be a pair of **encryption and decryption** functions,
- m_1 and m_2 be plaintexts.

The pair $(\boldsymbol{E}, \boldsymbol{D})$ forms an **homomorphic encryption scheme** for some operator $\diamond$ if and only if the following holds:

$$\boldsymbol{D}\left(\quad \boldsymbol{E}(m_1) \circ \boldsymbol{E}(m_2) \quad \right) \equiv \boldsymbol{D}\left(\quad \boldsymbol{E}(m_1 \diamond m_2) \quad \right).$$

For example, in **Paillier's proposal**, $\circ =$ multiplication and $\diamond =$ addition.

BFV

Scheme description

- RLWE-based,

BFV

Scheme description

- RLWE-based,
- Post-quantum secure,

BFV

Scheme description

- RLWE-based,
- Post-quantum secure,
- Basic arithmetic built upon polynomial rings of the form $R_p = \mathbb{Z}_p[X]/(X^N + 1)$,

BFV

Scheme description

- RLWE-based,
- Post-quantum secure,
- Basic arithmetic built upon polynomial rings of the form $R_p = \mathbb{Z}_p[X]/(X^N + 1)$,
- A security parameter λ, a plaintext domain defined as R_t, a ciphertext domain defined as R_q, for $q \gg t$.

BFV

Scheme description

Let **pk** and **evk** be an encryption and a relinearization key, respectively, related to a secret key **sk**.

BFV

Scheme description

Let **pk** and **evk** be an encryption and a relinearization key, respectively, related to a secret key **sk**.

BFV.Encrypt($\mathbf{pk}, m$): Let $\mathbf{pk} = (p_0, p_1)$, sample $u \leftarrow R_3$, and $e_0, e_1 \leftarrow \chi$.
Output: $\left(\lfloor q/t \rfloor \cdot m + u \cdot p_0 + e_0, \quad u \cdot p_1 + e_1 \right)$.

BFV.Decrypt($\mathbf{sk}, \mathrm{ct}$): Let $\mathrm{ct} = (c_0, c_1)$. Output:

$$m = \left[\left\lfloor \frac{t}{q} \left[c_0 + c_1 \cdot \mathbf{sk} \right]_q \right\rceil \right]_t.$$

BFV

Scheme description

Let **pk** and **evk** be an encryption and a relinearization key, respectively, related to a secret key **sk**.

BFV.Encrypt($\mathbf{pk}, m$): Let $\mathbf{pk} = (p_0, p_1)$, sample $u \leftarrow R_3$, and $e_0, e_1 \leftarrow \chi$.
Output: $\left(\lfloor q/t \rfloor \cdot m + u \cdot p_0 + e_0, \quad u \cdot p_1 + e_1 \right)$.

BFV.Decrypt($\mathbf{sk}, \mathrm{ct}$): Let $\mathrm{ct} = (c_0, c_1)$. Output:

$$m = \left[\left\lfloor \frac{t}{q} \left[c_0 + c_1 \cdot \mathbf{sk} \right]_q \right\rceil \right]_t.$$

BFV.Add(c_0, c_1): Output: $\left(c_{0,0} + c_{1,0}, \quad c_{0,1} + c_{1,1} \right)$.

BFV

Scheme description

Let **pk** and **evk** be an encryption and a relinearization key, respectively, related to a secret key **sk**.

BFV.Encrypt(**pk**, m): Let $\mathbf{pk} = (p_0, p_1)$, sample $u \leftarrow R_3$, and $e_0, e_1 \leftarrow \chi$.
Output: $(\quad \lfloor q/t \rfloor \cdot m + u \cdot p_0 + e_0, \quad u \cdot p_1 + e_1 \quad)$.

BFV.Decrypt(**sk**, ct): Let $\mathrm{ct} = (c_0, c_1)$. Output:

$$m = \left[\left\lfloor \frac{t}{q} \left[c_0 + c_1 \cdot \mathbf{sk} \right]_q \right\rceil \right]_t .$$

BFV.Add(c_0, c_1): Output: $(\quad c_{0,0} + c_{1,0}, \quad c_{0,1} + c_{1,1} \quad)$.

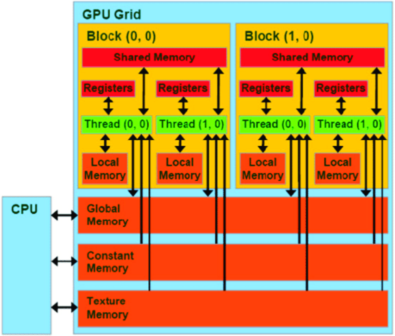
BFV.Mul(c_0, c_1, **evk**): Compute

$$c_0 = \left[\lfloor t/q \cdot c_{0,0} \cdot c_{1,0} \rceil \right]_q ,$$
$$c_1 = \left[\lfloor t/q \cdot (c_{0,0} \cdot c_{1,1} + c_{0,1}, \cdot c_{1,0}) \rceil \right]_q$$
$$c_2 = \left[\lfloor t/q \cdot c_{0,1} \cdot c_{1,1} \rceil \right]_q .$$

and return $\mathbf{c}_{\mathtt{mul}} = \mathtt{Relin}(c_0, c_1, c_2, \mathbf{evk})$.

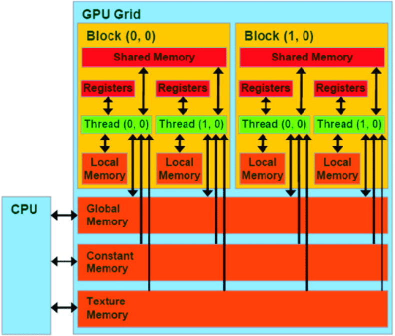
CUDA

- Parallel computing architecture.



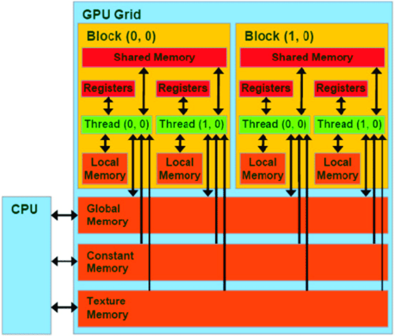
CUDA

- Parallel computing architecture.
- Thread-group oriented (as in a vector processor).



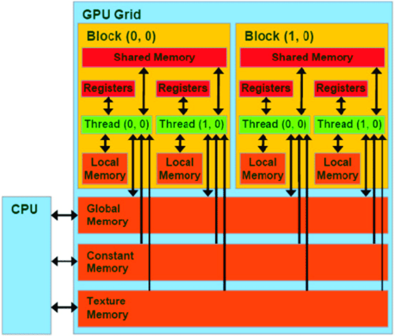
CUDA

- Parallel computing architecture.
- Thread-group oriented (as in a vector processor).
- Multiple memory spaces:



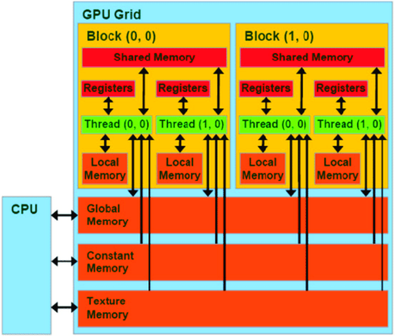
CUDA

- Parallel computing architecture.
- Thread-group oriented (as in a vector processor).
- Multiple memory spaces:
    - Global,



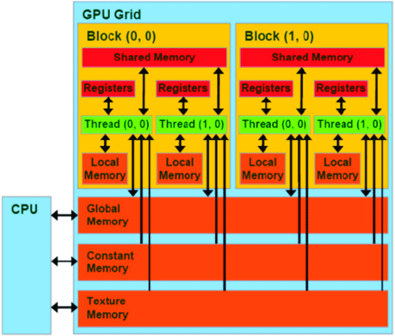
CUDA

- Parallel computing architecture.
- Thread-group oriented (as in a vector processor).
- Multiple memory spaces:
  - Global,
  - Shared,



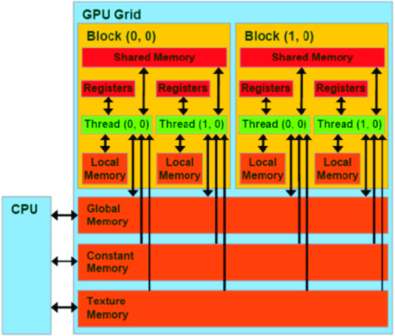
CUDA

- Parallel computing architecture.
- Thread-group oriented (as in a vector processor).
- Multiple memory spaces:
  - Global,
  - Shared,
  - Local,



CUDA

- Parallel computing architecture.
- Thread-group oriented (as in a vector processor).
- Multiple memory spaces:
  - Global,
  - Shared,
  - Local,
  - Constant.



Residue Number System – RNS

Mathematical background

One polynomial with **huge** coefficients

$\Updownarrow$

Many polynomials with **small** coefficients

Residue Number System – RNS

Mathematical background

Let $\{p_0, p_1, \cdots, p_{\ell-1}\}$ be a set of coprimes and $P \in R_q$.

$$P(x) = \sum_{i=0}^{N} a_i \cdot x^i \iff \begin{bmatrix} P(x) \mod p_0, \\ P(x) \mod p_1, \\ P(x) \mod p_2, \\ \dots \\ P(x) \mod p_{\ell-1} \end{bmatrix}$$

Residue Number System – RNS

Mathematical background

- RNS natively supports:
  - Addition,
  - Multiplication,
  - Modular reduction by a cyclotomic polynomial.

Residue Number System – RNS

Mathematical background

- RNS natively supports:
  - Addition,
  - Multiplication,
  - Modular reduction by a cyclotomic polynomial.
- **Does not** support:

Residue Number System – RNS

Mathematical background

- RNS natively supports:
  - Addition,
  - Multiplication,
  - Modular reduction by a cyclotomic polynomial.
- **Does not** support:
  - Integer modular reduction,

Residue Number System – RNS

Mathematical background

- RNS natively supports:
  - Addition,
  - Multiplication,
  - Modular reduction by a cyclotomic polynomial.
- **Does not** support:
  - Integer modular reduction,
  - Non-integer divisions,

Residue Number System – RNS

Mathematical background

- RNS natively supports:
  - Addition,
  - Multiplication,
  - Modular reduction by a cyclotomic polynomial.
- **Does not** support:
  - Integer modular reduction,
  - Non-integer divisions,
  - Roundings.

Residue Number System – RNS

Mathematical background

- RNS natively supports:
  - Addition,
  - Multiplication,
  - Modular reduction by a cyclotomic polynomial.
- **Does not** support:
  - Integer modular reduction,
  - Non-integer divisions,
  - Roundings.
- Halevi et al.'s BFV variant can handle that.

Polynomial multiplication in R_q

- Not a trivial operation.
- Computational complexity can reach $\Theta(N^2)$.
- Widely used by RLWE-based cryptosystems.

Discrete Fourier Transform – DFT

Mathematical background

- DFT-based transforms **reduce the computational complexity** to $\Theta(N)$ in the transform domain.

Discrete Fourier Transform – DFT

Mathematical background

- DFT-based transforms **reduce the computational complexity** to $\Theta(N)$ in the transform domain.
- Variants with **log-linear** complexity.

Discrete Fourier Transform – DFT

Mathematical background

- DFT-based transforms **reduce the computational complexity** to $\Theta(N)$ in the transform domain.
- Variants with **log-linear** complexity.
- Let ω_N be a primitive N-th root of unity.

Discrete Fourier Transform – DFT

Mathematical background

- DFT-based transforms **reduce the computational complexity** to $\Theta(N)$ in the transform domain.
- Variants with **log-linear** complexity.
- Let ω_N be a primitive N-th root of unity.
  1. Fast Fourier Transform (**FFT**): $\omega_N \in \mathbb{C}$.

Discrete Fourier Transform – DFT

Mathematical background

- DFT-based transforms **reduce the computational complexity** to $\Theta(N)$ in the transform domain.
- Variants with **log-linear** complexity.
- Let ω_N be a primitive N-th root of unity.
  1. Fast Fourier Transform (**FFT**): $\omega_N \in \mathbb{C}$.
  2. *Number-Theoretic Transform* (**NTT**): $\omega_N \in GF(p)$.

Discrete Fourier Transform – DFT

Mathematical background

- DFT-based transforms **reduce the computational complexity** to $\Theta(N)$ in the transform domain.
- Variants with **log-linear** complexity.
- Let ω_N be a primitive N-th root of unity.
  1. Fast Fourier Transform (**FFT**): $\omega_N \in \mathbb{C}$.
  2. *Number-Theoretic Transform* (**NTT**): $\omega_N \in GF(p)$.
  3. *Discrete Galois Transform* (**DGT**): $\omega_N \in GF(p^2)$.

Discrete Galois Transform

- $u \in GF(p^2)$ can be **represented as** $u_{re} + i \cdot u_{im}$, where $u_{re}, u_{im} \in GF(p)$ and $i = \sqrt{-1}$, also known as **Gaussian Integers**.

Discrete Galois Transform

- $u \in GF(p^2)$ can be **represented as** $u_{re} + i \cdot u_{im}$, where $u_{re}, u_{im} \in GF(p)$ and $i = \sqrt{-1}$, also known as **Gaussian Integers**.
- There are some convenient properties:

Discrete Galois Transform

- $u \in GF(p^2)$ can be **represented as** $u_{re} + i \cdot u_{im}$, where $u_{re}, u_{im} \in GF(p)$ and $i = \sqrt{-1}$, also known as **Gaussian Integers**.
- There are some convenient properties:
  - Negacyclic convolution,

Discrete Galois Transform

- $u \in GF(p^2)$ can be **represented as** $u_{re} + i \cdot u_{im}$, where $u_{re}, u_{im} \in GF(p)$ and $i = \sqrt{-1}$, also known as **Gaussian Integers**.
- There are some convenient properties:
  - Negacyclic convolution,
  - Polynomial folding.

Polynomial multiplication

Discrete Galois Transform

- Loop dependency forces the use of a single Thread Block to assert synchronization.

Polynomial multiplication

Discrete Galois Transform

- Loop dependency forces the use of a single Thread Block to assert synchronization.
- A Thread Block is limited to 1024 CUDA-Threads.

Polynomial multiplication

Discrete Galois Transform

- Loop dependency forces the use of a single Thread Block to assert synchronization.
- A Thread Block is limited to 1024 CUDA-Threads.
  - It can also be achieved by successive CUDA-Kernel calls, at the cost of a considerable overhead.

Polynomial multiplication

Discrete Galois Transform

- Loop dependency forces the use of a single Thread Block to assert synchronization.
- A Thread Block is limited to 1024 CUDA-Threads.
  - It can also be achieved by successive CUDA-Kernel calls, at the cost of a considerable overhead.
- We propose a formulation named **hierarchical DGT** (HDGT).

Polynomial multiplication

Discrete Galois Transform

- Loop dependency forces the use of a single Thread Block to assert synchronization.
- A Thread Block is limited to 1024 CUDA-Threads.
  - It can also be achieved by successive CUDA-Kernel calls, at the cost of a considerable overhead.
- We propose a formulation named **hierarchical DGT** (HDGT).
  - Targets constrained devices.

Polynomial multiplication

Discrete Galois Transform

- Loop dependency forces the use of a single Thread Block to assert synchronization.
- A Thread Block is limited to 1024 CUDA-Threads.
  - It can also be achieved by successive CUDA-Kernel calls, at the cost of a considerable overhead.
- We propose a formulation named **hierarchical DGT** (HDGT).
  - Targets constrained devices.
  - Originally proposed for the FFT by **Bailey:1990** and **Govindaraju:2008**.

Hierarchical Discrete Galois Transform

Description

$$p(x) = a_0 + a_1 x + \cdots + a_{2N-1} x^{2N-1}.$$

1. Vector representation.

$$\begin{bmatrix} a_0 & a_1 & \ldots & a_{2N-1} \end{bmatrix}$$

Hierarchical Discrete Galois Transform

Description

$$p(x) = a_0 + a_1 x + \cdots + a_{2N-1} x^{2N-1}.$$

1. Vector representation.
2. Folding.

$$\left[\; (a_0 + i a_N) \quad \ldots \quad (a_{N-1} + i a_{2N-1}) \; \right]$$

Hierarchical Discrete Galois Transform

Description

$$p(x) = a_0 + a_1 x + \cdots + a_{2N-1} x^{2N-1}.$$

1. Vector representation.
2. Folding.

$$\begin{bmatrix} \tilde{a}_0 & \tilde{a}_1 & \ldots & \tilde{a}_{N-1} \end{bmatrix}$$

Hierarchical Discrete Galois Transform

Description

$$p(x) = a_0 + a_1 x + \cdots + a_{2N-1} x^{2N-1}.$$

1. Vector representation.
2. Folding.
3. Twisting – Mult. by powers of a primitive N-th root of i mod p.

$$\left[\ (\tilde{a}_0 \cdot h^0) \quad (\tilde{a}_1 \cdot h^1) \quad \ldots\ \right]$$

Hierarchical Discrete Galois Transform

Description

$$p(x) = a_0 + a_1 x + \cdots + a_{2N-1} x^{2N-1}.$$

1. Vector representation.
2. Folding.
3. Twisting – Mult. by powers of a primitive N-th root of i mod p.

$$\begin{bmatrix} \tilde{b}_0 & \tilde{b}_1 & \ldots & \tilde{b}_{N-1} \end{bmatrix}$$

Hierarchical Discrete Galois Transform

Description

$$p(x) = a_0 + a_1 x + \cdots + a_{2N-1} x^{2N-1}.$$

1. Vector representation.
2. Folding.
3. Twisting – Mult. by powers of a primitive N-th root of i mod p.
4. Matrix representation – (n_r, n_c).

$$\begin{bmatrix} \tilde{b}_0 & \tilde{b}_1 & \dots & \tilde{b}_{n_c-1} \\ \tilde{b}_{n_c} & \tilde{b}_{n_c+1} & \dots & \tilde{b}_{2n_c-1} \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

Hierarchical Discrete Galois Transform

Description

$$p(x) = a_0 + a_1 x + \cdots + a_{2N-1} x^{2N-1}.$$

1. Vector representation.
2. Folding.
3. Twisting – Mult. by powers of a primitive N-th root of i mod p.
4. Matrix representation – (n_r, n_c).
5. Apply the canonical DGT through the columns.

$$\begin{bmatrix} \tilde{b}_0 & \tilde{b}_1 & \dots & \tilde{b}_{n_c-1} \\ \tilde{b}_{n_c} & \tilde{b}_{n_c+1} & \dots & \tilde{b}_{2n_c-1} \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

Hierarchical Discrete Galois Transform

Description

$$p(x) = a_0 + a_1 x + \cdots + a_{2N-1} x^{2N-1}.$$

1. Vector representation.
2. Folding.
3. Twisting – Mult. by powers of a primitive N-th root of i mod p.
4. Matrix representation – (n_r, n_c).
5. Apply the canonical DGT through the columns.

$$\begin{bmatrix} B_0 & \tilde{b}_1 & \dots & \tilde{b}_{n_c-1} \\ B_{n_c} & \tilde{b}_{n_c+1} & \dots & \tilde{b}_{2n_c-1} \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

Hierarchical Discrete Galois Transform

Description

$$p(x) = a_0 + a_1 x + \cdots + a_{2N-1} x^{2N-1}.$$

1. Vector representation.
2. Folding.
3. Twisting – Mult. by powers of a primitive N-th root of i mod p.
4. Matrix representation – (n_r, n_c).
5. Apply the canonical DGT through the columns.

$$\begin{bmatrix} B_0 & B_1 & \dots & \tilde{b}_{n_c-1} \\ B_{n_c} & B_{n_c+1} & \dots & \tilde{b}_{2n_c-1} \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

Hierarchical Discrete Galois Transform

Description

$$p(x) = a_0 + a_1 x + \cdots + a_{2N-1} x^{2N-1}.$$

1. Vector representation.
2. Folding.
3. Twisting – Mult. by powers of a primitive N-th root of i mod p.
4. Matrix representation – (n_r, n_c).
5. Apply the canonical DGT through the columns.
6. Multiply $B_{j,k}$ by $g_{j,k} = \omega_{N/2}^{\text{bit-reversal}(j)\cdot k}$.

$$\begin{bmatrix} B_0 g_{0,0} & \dots & B_{n_c-1} g_{0,n_c-1} \\ B_{n_c} g_{1,0} & \dots & B_{2n_c-1} g_{1,n_c-1} \\ \vdots & \vdots & \vdots \end{bmatrix}$$

Hierarchical Discrete Galois Transform

Description

$$p(x) = a_0 + a_1 x + \cdots + a_{2N-1} x^{2N-1}.$$

1. Vector representation.
2. Folding.
3. Twisting – Mult. by powers of a primitive N-th root of i mod p.
4. Matrix representation – (n_r, n_c).
5. Apply the canonical DGT through the columns.
6. Multiply $B_{j,k}$ by $g_{j,k} = \omega_{N/2}^{\text{bit-reversal}(j)\cdot k}$.

$$\begin{bmatrix} C_0 & C_1 & \dots & C_{n_c-1} \\ C_{n_c} & C_{n_c+1} & \dots & C_{2n_c-1} \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

Hierarchical Discrete Galois Transform

Description

$$p(x) = a_0 + a_1 x + \cdots + a_{2N-1} x^{2N-1}.$$

1. Vector representation.
2. Folding.
3. Twisting – Mult. by powers of a primitive N-th root of i mod p.
4. Matrix representation – (n_r, n_c).
5. Apply the canonical DGT through the columns.
6. Multiply $B_{j,k}$ by $g_{j,k} = \omega_{N/2}^{\text{bit-reversal}(j)\cdot k}$.
7. Apply the canonical DGT through the rows.

$$\begin{bmatrix} C_0 & C_1 & \dots & C_{n_c-1} \\ C_{n_c} & C_{n_c+1} & \dots & C_{2n_c-1} \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

Hierarchical Discrete Galois Transform

Description

$$p(x) = a_0 + a_1 x + \cdots + a_{2N-1} x^{2N-1}.$$

1. Vector representation.
2. Folding.
3. Twisting – Mult. by powers of a primitive N-th root of i mod p.
4. Matrix representation – (n_r, n_c).
5. Apply the canonical DGT through the columns.
6. Multiply $B_{j,k}$ by $g_{j,k} = \omega_{N/2}^{\text{bit-reversal}(j)\cdot k}$.
7. Apply the canonical DGT through the rows.

$$\begin{bmatrix} A_0 & A_1 & \ldots & A_{n_c-1} \\ C_{n_c} & C_{n_c+1} & \ldots & C_{2n_c-1} \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

Hierarchical Discrete Galois Transform

Description

$$p(x) = a_0 + a_1 x + \cdots + a_{2N-1} x^{2N-1}.$$

1. Vector representation.
2. Folding.
3. Twisting – Mult. by powers of a primitive N-th root of i mod p.
4. Matrix representation – (n_r, n_c).
5. Apply the canonical DGT through the columns.
6. Multiply $B_{j,k}$ by $g_{j,k} = \omega_{N/2}^{\text{bit-reversal}(j)\cdot k}$.
7. Apply the canonical DGT through the rows.

$$\begin{bmatrix} A_0 & A_1 & \cdots & A_{n_c-1} \\ A_{n_c} & A_{n_c+1} & \cdots & A_{2n_c-1} \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

Hierarchical Discrete Galois Transform

Description

$$p(x) = a_0 + a_1 x + \cdots + a_{2N-1} x^{2N-1}.$$

1. Vector representation.
2. Folding.
3. Twisting – Mult. by powers of a primitive N-th root of i mod p.
4. Matrix representation – (n_r, n_c).
5. Apply the canonical DGT through the columns.
6. Multiply $B_{j,k}$ by $g_{j,k} = \omega_{N/2}^{\text{bit-reversal}(j) \cdot k}$.
7. Apply the canonical DGT through the rows.

$$\text{HDGT}(p(x)) = A_0 + \cdots + A_{N-1} x^{N-1},$$

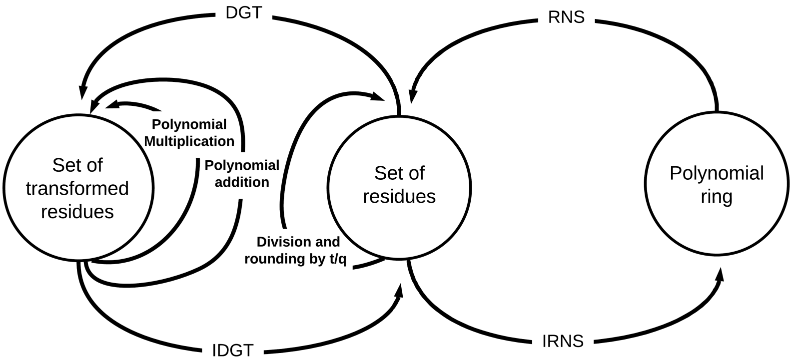
$$\text{s.t. } A_i \in GF(p^2).$$

SPOG - Secure Processing On GPGPUs

- Proof-of-concept implementation written in C++.
- Targets CUDA.
- Applies HDGT for polynomial multiplication.
- Modular implementation, separating polynomial arithmetic and cryptosystem.
  - cuPoly,
  - HPS-BFV.
- cuRAND is used for sampling.

Data locality

SPOG - Secure Processing On GPGPUs



Comparison with other works

Methodology

SPOG is compared with two other works in the literature. Since none of them release the source codes, we replicate the processing environment.

- BPAVR – Badawi, Polyakov, Aung, Veeravalli, and Rohlof
  - Tesla V100,
  - but presents latency for decryption and homomorphic multiplication only.

Comparison with other works

Methodology

SPOG is compared with two other works in the literature. Since none of them release the source codes, we replicate the processing environment.

- BPAVR – Badawi, Polyakov, Aung, Veeravalli, and Rohlof
  - Tesla V100,
  - but presents latency for decryption and homomorphic multiplication only.
- BVMA – Badawi, Veeravalli, Mun, and Aung,
  - Tesla K80,
  - Much more complete latency description.

Parameters

Methodology

Two different setups are considered for Google Cloud VMs running NVIDIA Tesla K80 and V100, referred to as *gc.k80* and *gc.v100*.

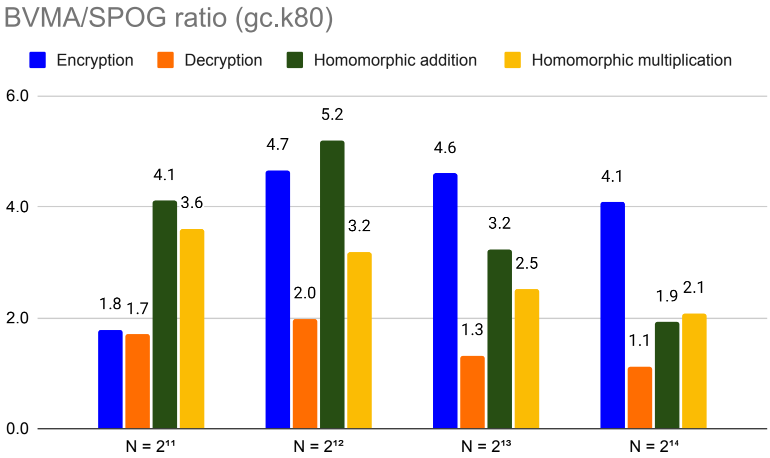
log N	*gc.k80*	*gc.v100*
11	62	60
12	186	60
13	372	120
14	744	360
15	744	600

Table: Lower bound for the size of q in bits.

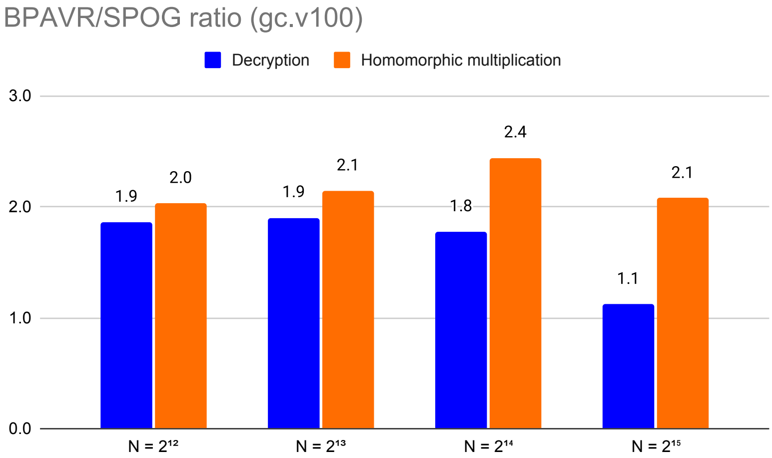
In both cases, $t = 256$.

SPOG vs BVMA

Latencies



SPOG vs BPAVR
Latencies



HDGT

We compare HDGT with two implementations of its canonical formulation:

DGT-I Multi-kernel design

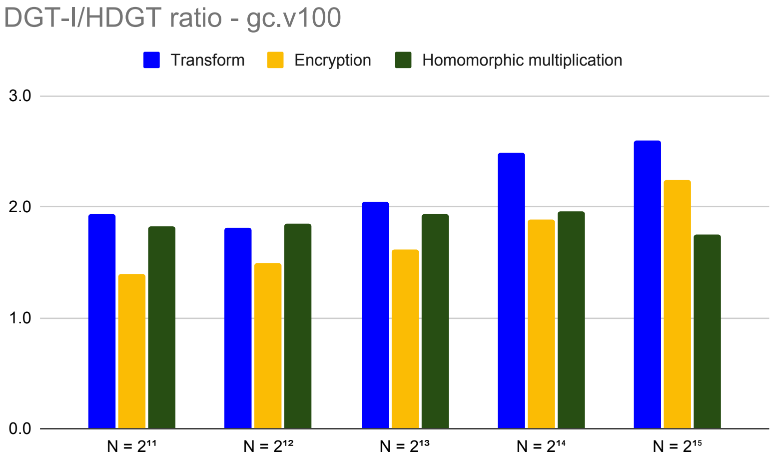
- Synchronization forced through $\log \frac{N}{2}$ CUDA-Kernel calls.

DGT-II Single-kernel design

- Synchronization limited to **block level**.
- Supports up to 2048-degree polynomials.

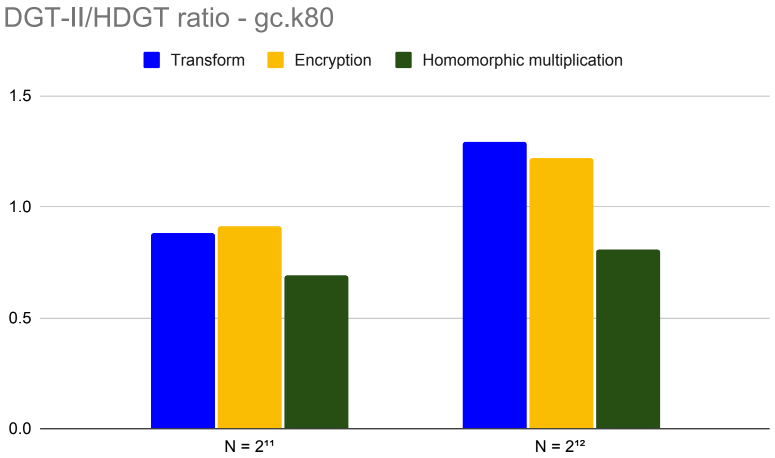
HDGT vs DGT-I

Latencies



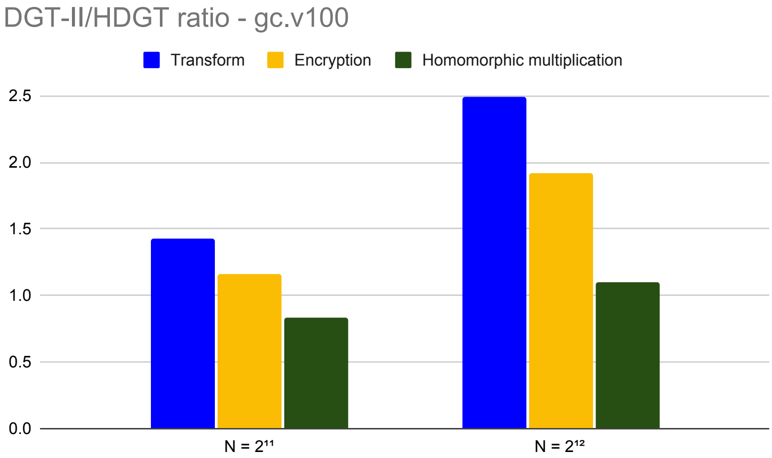
HDGT vs DGT-II – Tesla K80

Latencies



HDGT vs DGT-II – Tesla V100

Latencies



Conclusion

Future work:

- Direct comparison between HDGT, NTT, and HNTT on GPUs.
- SPOG-CKKS.
- Benchmarks of complex applications running over SPOG-BFV and SPOG-CKKS.

Acknowledgements

- CNPq,
- CAPES,
- LG,
- Google.

Acknowledgements

- CNPq,
- CAPES,
- LG,
- Google.

Thank you!

Faster Homomorphic Encryption over GPGPUs via hierarchical DGT

Pedro G. M. R. Alves[1] Jheyne N. Ortiz[1] Diego F. Aranha[2]
pedro.alves@ic.unicamp.br

[1]Institute of Computing, University of Campinas
[2]Department of Computer Science, Aarhus University

March 4, 2021

Financial Cryptography and Data Security 2021

References I