Developing Accelerators for Homomorphic Encryption

Efe İzbudak[1]
Eren Özilgili[2]
Rümeysa Bilik[1]
Mehmet Berkay Çatak[3]
Cenker Şahin[3]

Supervisor:
Erkay Savaş

[1]Middle East Technical University
[2]Bilkent University
[3]Sabancı University

29 August, 2024

Abstract

Homomorphic encryption (HE) is a cryptographic method that allows computations on encrypted data without the need for decryption. Thus, HE enables third parties to process data while keeping it confidential, making HE valuable in scenarios where data privacy is essential. Despite the existence of numerous encryption schemes, the widespread adoption of HE remains limited due to its significant computational overhead, which renders it currently impractical for many applications. Hence, accelerators are needed. Candidates for this can be CUDA and FPGA, which are explored in this article. Cryptographic schemes considered in this report are CKKS and TFHE which both use the hardness of LWE (Learning With Errors) problem.

Keywords. Homomorphic encryption, CKKS, TFHE, Acceleration

1 Preliminaries

1.1 DFT (Discrete Fourier Transform)

The DFT is a linear transformation that changes the basis of a complex vector from the standart basis (i.e., the spatial domain) to a Fourier basis (i.e., the frequency domain). The application of the DFT to a vector x of N values is just multiplication by the matrix W which is the $N \times N$ Vandermonde matrix generated by a primitive Nth root of unity.

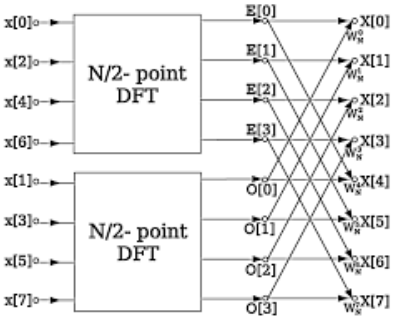
For a vector $x = (a_0, \ldots, a_{n-1})$, which represents the coefficients of a polynomial $a_0 + a_1 + \ldots + a_{n-1}x^{n-1}$, the DFT operation corresponds to evaluating the polynomial at the roots of unity, while the inverse operation corresponds to polynomial interpolation from known values at these roots of unity.

Unfortunately, the time complexity of the DFT and its inverse is $O(N^2)$. The Fast Fourier Trasnform (FFT) is an algorithm that reduces this complexity to $O(N \log N)$ through matrix factorization:

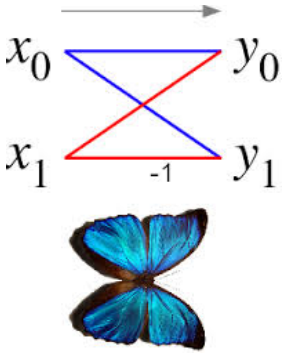
$$W_{2n} = \begin{bmatrix} I & D \\ I & -D \end{bmatrix} \begin{bmatrix} W_n & 0 \\ 0 & W_n \end{bmatrix} P,$$

where W_k is the Vandermonde matrix of size $k \times k$ and D is the diagonal matrix $\mathrm{diag}(1, w, w^2, \ldots, w^{n-1})$ with w the primitive nth root of unity generating W_k's, and P is the bit-reversal permutation matrix.

The FFT and its inverse are also essential for efficient polynomial multiplication as they allow us to switch to the frequency domain, perform coordinate-wise multiplications, and then return to the spatial domain.



(a) N/2-point DFT diagram



(b) Butterfly diagram

FFT can also be used for multiplying two polynomials and uniquely determining the resulting polynomial. Let $p(x)$ and $q(x)$ be polynomials such that their degrees are both $n-1$ and the polynomials satisfy $p(x) \times q(x) = r(x)$. In order to find the coefficients of the polynomial $r(x)$ using inverse FFT, interpolation of the $r(x)$ at $2n-2$ must be known since the degree of $r(x)$ will be equal to $\deg(p(x)) + \deg(q(x)) = (n-1) + (n-1)$ which is precisely $2n-2$. To achieve this, respectively, $p(x)$ and $q(x)$ are padded with n many 0's to view them as a polynomials of degree $2n-1$. Now, applying FFT to both the polynomials $p(x)$ and $q(x)$ will give rise to 2 vectors with length $2n$. These $2n$ values are generated by a primitive $2N$th root of unity. Evaluating the $p(y_i) \times q(y_i) = r(y_i)$ for $i \in \{1, 2, \ldots, 2n\}$ will be sufficient for polynomial $r(x)$ to be interpolated

successfully using inverse of the FFT and the output will be the coefficients of the polynomial $r(x)$.

1.2 LWE (Learning With Errors)

LWE is instantiated by n and q such that $n, q \in \mathbb{Z}^+$, and an error distribution χ over $\mathbb{Z}$. The parameter χ is usually taken to be a discrete Gaussian of width αq for some $\alpha < 1$. LWE distribution over $Z_q^n \times Z_q$ is sampled as outputting $(\mathbf{a}, b = \langle \mathbf{s}, \mathbf{a} \rangle + emodq)$ for vector $s \in Z_q^n$ called the secret, where $\mathbf{a}$ is chosen from Z_q^n uniformly at random and e is chosen as $e \leftarrow \chi$.

There are two main versions of the LWE problem which are search problem and decision problem where given m samples it is hard to attain the secret $s \in Z_q^n$ and where again, given m samples it is hard to distinguish every sample whether they are sampled according to LWE distribution or uniform distribution, respectively.

The schemes TFHE and CKKS both use LWE problem or their variants such as RLWE (Ring Learning With Errors) where the variables are instantiated as polynomials modulo some other polynomial such as X^N+1 where N is a positive integer power of 2.

2 CKKS Scheme

In CKKS, a variant of LWE which is called RLWE (ring-LWE) is used where plaintexts are elements from the ring of integers of a number field and encryption of a plaintext polynomial m is given by

$$(-as + m + e, a)$$

where s is the secret key polynomial, a is public and coefficients of e are sampled from an appropriate Gaussian distribution. Encrypted ciphertexts can be added or multiplied by each other unless the noise e present in the ciphertext does not pass a certain threshold.

CKKS allows for approximate homomorphic encryption, meaning, decryption of ciphertext outputs the value $m + e$ which is an approximation for the encrypted message m. A process called bootstrapping can help reduce the noise present in the ciphertexts and can allow operations to be further executed on ciphertexts.

2.1 Python Implementations for CKKS

Sparse Encoding: We have created a proof-of-concept implementation of the sparse encoding procedure described in the paper "Bootstrapping for Approximate Homomorphic Encryption" by Cheon et. al.

Bootstrapping Optimization: We have explored various different methods of bootstrapping CKKS ciphertexts to find the most efficient methıds. We have decided upon hybrid of multiple approaches expressed by Han and Ki, 2019, Bossuat et al., 2020 and Cheon et al., 2018 to minimize noise growth while keeping the consumed level small and implemented it using Python.

The proof-of-concept implementations reduce the workload on CUDA implementations and provide easier debugging.

3 TFHE Scheme (Fully Homomorphic Encryption Over the Torus)

3.1 Ciphertext Types

3.1.1 LWE

One of the most known and essential problems of construction of Lattice Based Homomorphic Encryption is LWE (Learning with Errors). As Chilotti et al. used in their paper, we will use LWE completely on the real torus.

For encryption of LWE we are using q as the ciphertext modulus and p as the plaintext modulus. We consider delta as q/p.

Let $q, p \in \mathbb{Z}$. We will consider both as 2^π, if not, a rounding is needed. To encrypt a message $m \in \mathbb{Z}_p$ with a secret key $s \in \mathbb{Z}_q$. then encryption is obtained by sampling a pair (a, b) where $a \in \mathbb{Z}_q^n$ is sampled uniformly random and $b = as + \Delta m + e$ where e is the gaussian probability distribution, which will be called as the **noise**.

Using this intuition on LWE we can define the same problem on rings: RLWE.

Definition 3.1. Let $N \in \mathbb{Z}$ and $S \in R_q^n$. Then **RLWE encryption** of the secret message M is a pairing (A, B) where A is uniformly sampled from R_q and $B = AS + \Delta M + E$. E is the noise.

with this definition it is easy to understand how one can decrypt a message which is encrypted with RLWE in two simple steps:

1. $B - AS = \Delta M + E$
2. $\frac{\Delta M + E}{\Delta} = M$

Here the noise we received from Gaussian Distribution must satisfy the condition $|E| < \frac{\Delta}{2}$ in order for message to be decrypted as expected.

The reason why LWE works is the assumption of a torus element constructed as $b = \sum_{j=1}^{n} s_j \cdot a_j + e$ cannot be distinguished from a random torus element even if a_j is known (Joye, 2022).

3.1.2 GGSW

In order to add more levels to LWE, we create a ciphertext type called GGSW. In order to construct it we will start with another ciphertext called GLev, which will add one more level to the LWE ciphertext.

Definition 3.2. A **Glev ciphertext** is a list of GLWE ciphertexts encrypting the same message M with different Δs. These scaling factors are defined using a base β and l representing the number of the levels:

$(\text{GLWE}_{\vec{S},\sigma}(\frac{q}{\beta^1} \cdot M) \times \ldots \times \text{GLWE}_{\vec{S},\sigma}(\frac{q}{\beta^l} \cdot M)) = \text{GLev}_{\vec{S},\sigma}^{\beta,l}(M)$ (Zama AI, 2022)

Now it is quite easy to define GGSW using the definition we gave for GLev ciphertext.

Definition 3.3. A GGSW ciphertext encryption is a list of GLev encryptions, encrypting the multiplication of M with one of the polynomials of secret key S.

$\text{GLev}_{\vec{S},\sigma}^{\beta,l}(-S_0 M) \times \ldots \times \text{GLev}_{\vec{S},\sigma}^{\beta,l}(-S_{k-1} M) \times \text{GLev}_{\vec{S},\sigma}^{\beta,l}(M) = \text{GGSW}_{\vec{S},\sigma}^{\beta,l}(M)$

3.2 Homomorphic Operations

One of the reasons why we tend to use FHE schemes for encryption is that we can perform homomorphic operations on the encrypted data. However performing operations on encrypted data causes a raise in the noise. At this point we will consider the ways to keep noise as minimum as possible but in the following sections we will see that for TFHE we have a specific method to decrease the noise which is called bootstrapping.

Definition 3.4. Consider two GGSW encryptions with the same plaintext and ciphertext modulus. Message M encrypted using secret key S is $C = (A_0, ..., A_k - 1, B) \in \text{GGSW}^{\beta,l}_{\vec{S},\sigma}(M)$ and message M' encrypted using secret key S' is $C' = (A'_0, ..., A'_k - 1, B') \in \text{GGSW}^{\beta,l}_{\vec{S'},\sigma}(M')$. Then one can perform **homomorphic addition** as follows:

$C + C2 = (A_0 + A'_0, ..., A_{k-1} + A'_{k-1}, B + B')$

This operation will cause the noise to grow but not in the sense that it will effect the result. However, this is not the case for the homomorphic multiplication of the ciphertext with a large constant. In this case we have to manipulate the ciphertext into smaller pieces in order the prevent large increase in the noise.

To achieve this purpose, we create a **decomposition operator** which takes a large constant and decompose it into a base β as follows:

$\gamma = \gamma_1 \frac{q}{\beta^1} + \gamma_2 \frac{q}{\beta^2} + ... + \gamma_l \frac{q}{\beta^l}$ (Zama AI, 2022)

Then we can say $\text{Decomp}^{\beta,l}(\gamma) = (\gamma_1, \gamma_2, ..., \gamma_l)$. Using the decomposition operator we can define internal and external products (Chillotti et al., 2017).

Definition 3.5 (External product)**.** We define the product $\boxdot$ as

$$\boxdot : \text{TGSW} \times \text{TLWE} \longrightarrow \text{TLWE}$$

$$(A, \mathbf{b}) \mapsto A \boxdot \mathbf{b} = \text{Dec}_{H,\beta,\epsilon}(\mathbf{b}) \cdot A,$$

Corollary (Internal Product)**.** *Let the product*

$$\boxtimes : TGSW \times TGSW \longrightarrow TGSW$$

be defined as

$$(A, B) \mapsto A \boxtimes B = \begin{bmatrix} A \boxtimes \mathbf{b}_1 \\ \vdots \\ A \boxtimes \mathbf{b}_{(k+1)\ell} \end{bmatrix} = \begin{bmatrix} Dec_{H,\beta,\epsilon}(\mathbf{b}_1) \cdot A \\ \vdots \\ Dec_{H,\beta,\epsilon}(\mathbf{b}_{(k+1)\ell}) \cdot A \end{bmatrix},$$

with A and B two valid TGSW samples of messages μ_A and μ_B respectively and $\mathbf{b}_i$ corresponding to the i-th line of B. Then $A \boxtimes B$ is a TGSW sample of message $\mu_A \cdot \mu_B$.

3.3 Building Blocks

All of the building blocks mentioned in this section will be used to create the method we sneaked to decrease the noise: Bootstrapping.

3.3.1 CMux Gate

We defined the external product mainly to use it to construct a controlled multiplexer, CMux gate. We can consider a CMux gate as an if condition. It takes one selector and two choices as inputs and gives an output according to the matching of choices and the selector as it can be seen from the figure. Unlike the other blocks, CMux Gate is not directly a part of the bootstrapping process but rather a building block for the blind rotation.

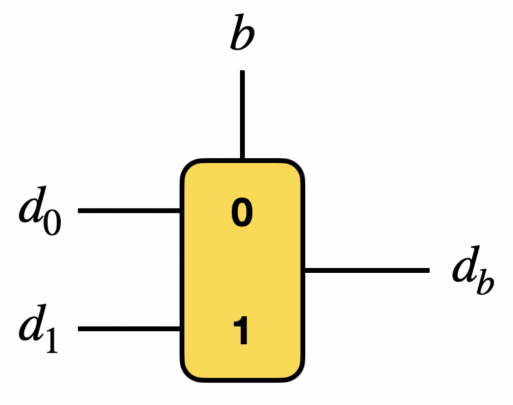


Figure 2: A visualization of CMux gate (Zama AI, 2022)

One can consider this operation as: $b(d_1 - d_0) + d_0 = d_b$.

3.3.2 Key Switching

Key switching is an homomorphic encryption which helps to switch the key of an encryption from S to S'. The idea is to encrypt S using S' and after some cancellations having the same message M encrypted under the key switching key S'. So we will consider key switching key S' as a GLev encryption of S, with base β, level l, standart deviation σ_{KSK}.

So in practice,

$C' = (0, ..., 0, B) - \sum_{j=0}^{k-1} < \text{Decomp}^{\beta,l}(A_j), S'_j >$ is the new encryption of the message which switched the key from S to S'.

3.3.3 Sample Extraction

The aim of sample extraction is to extract a single coefficent as a LWE sample from an RLWE encryption (Chillotti et al., 2017).

3.3.4 Blind Rotation

When rotation is mentioned in the context of TFHE what we are trying to do is to shift the coefficents of a polynomial. In order to do that we have to multiply our polynomial with $X^{-\pi}$ for some $0 \leq \pi < N$. Since we are considering in mod $X^N + 1$, polynomial will be shifted to the left as follows:

$M = m_0 + m_1 X + ... + m_\pi X^\pi + ... + m_{N-1} X^{N-1}$

$M.X^{-\pi} = m_\pi + ... + m_{N-1} X^{N-\pi-1} - m_0 X^{N-\pi} - ... - m_{\pi-1} X^{N-1}$ (Zama AI, 2022)

The logic behind the blind rotation is to encrypt the number of positions that the polynomial will be rotated. Since π is a constant we can express it in its binary decomposition such that $\pi = \pi_0 2^0 + \pi_1 2^1 + ... + \pi_j 2^j + ... + \pi_k 2^k$ and we encrypt each π_j under the secret key S. So now rotation becomes

$M.X^{\pi} = M.X^{\pi_0 2^0 + ... + \pi_j 2^j + ... + \pi_k 2^k} = M.X_0^{\pi} ... X_j^{\pi} ... X_k^{\pi}$

Now we can consider the typical element with M as $M.X_j^{\pi}$ here we realize if π_j is 0 then $M.X_j^{\pi} = M$ if π_j is 1 then $M.X_j^{\pi} = M.X$. This construction simply creates a CMux gate. So continuously evaluating CMux gates will create the aimed blind rotation (Zama AI, 2023).

3.4 Bootstrapping

All of the building blocks mentioned in this paper will build up to one single method, which is essential for TFHE. Bootstrapping is a method developed for FHE schemes which helps reducing the noise and by that aids the process of performing endless homomorphic operations on the encrypted data. To understand how bootstrapping works we have to revisit the idea behind RLWE decryption and use a LUT (Look-Up Table). Using building blocks introduced in each step, we will be able to reduce the noise:

1. $B - AS = \Delta M + E$
2. $(\Delta M + E)/\Delta = M$

We will be starting from associating M to $\Delta M + E$. In order to do that first we need to introduce Δ-long redundancy to M. These blocks containing repetitions of the same coefficients are called mega-cases. Rotating these mega-cases with multiplying them with $X^{\Delta M + E}$, we obtain a association between $(\Delta M + E)/\Delta$ and M.

Now, it is enough to calculate $\Delta M + E$ to achieve the first step of the decryption. Consider the LUT of the redundant messages as V. First thing that has to be done is to rotate V to B positions to obtain AS alone. So rotated V will become $V_0 = VX^{-B}$. This operation turns out to be a blind rotation according to the value of the element of the secret key S since its elements are binary.

So now, we achieved to have message V starting with M, so using sample extraction it is easy to access M at this point. Now, M has less noise than the same M we had at the beginning but it is encrypted with another key. Here we can use key switching which does not increase the noise. At the end of the bootstrapping process we managed to have the same message with less noise which allows us to continue using this data without worrying about the noise. (Zama AI, 2023).

4 FPGA

4.1 Fast Multiplication for TFHE Implementations

In fully homomorphic encryption (FHE) schemes, large polynomial multiplications are crucial due to their role in performing arithmetic operations on encrypted data. To optimize these multiplications, fast multiplication algorithms

such as the Montgomery reduction and the Karatsuba algorithm are employed. These algorithms reduce computational complexity and improve efficiency, making them suitable for FHE implementations. This section explores the implementation of these two algorithms in Verilog, highlighting their advantages and the work done to tailor them for specific use cases.

4.2 Modular Multiplication Using Montgomery Reduction

The Montgomery reduction algorithm is a widely used technique in modular multiplication, particularly in cryptographic applications where operations over large integers are common. The algorithm allows modular multiplication to be performed efficiently without directly executing division operations, which are computationally expensive. Instead, it replaces division with a series of addition and bit-shifting operations, making it well-suited for hardware implementations.

The main advantage of Montgomery reduction lies in its ability to perform modular multiplication without needing an explicit division step, which typically involves costly operations. This efficiency is particularly beneficial when dealing with large numbers, as in the case of FHE implementations. By transforming the operands into the Montgomery domain, modular multiplications can be performed rapidly, followed by a conversion back to the standard domain.

In this project, an existing modular multiplication implementation was provided. The original implementation was syntactically modified to design a Verilog module using generate blocks, with the goal of eventually transforming the entire implementation into a fully parameterizable module. This approach allows the modular multiplication implementation to be tailored for various starting bit sizes and reduction step sizes by simply adjusting these parameters within the Verilog code.

The initial modification was performed on a specific case involving a 64-bit number with a reduction step of 13. The generate blocks were used to create this modular multiplication module, serving as an example for future extensions. The ultimate objective is to eliminate the need for Python scripts currently used to rewrite the Verilog files based on input parameters. Instead, the Verilog code itself will be adaptable to any starting bit number and reduction step size, providing greater flexibility and efficiency in the design of modular multipliers for cryptographic applications.

4.3 Karatsuba Multiplication

The Karatsuba algorithm is a fast multiplication algorithm that reduces the complexity of large integer multiplication from $O(n^2)$ to $O(n^{\log_2 3})$, making it significantly faster than traditional long multiplication for large numbers. The algorithm works by recursively breaking down the multiplication of two large numbers into smaller multiplications, which can be computed more efficiently.

The basic procedure of the Karatsuba algorithm involves splitting each number into two halves and then recursively applying the multiplication. For two numbers A and B, each of size n, the algorithm splits them into $A = A_1 \times 2^{n/2} + A_0$ and $B = B_1 \times 2^{n/2} + B_0$. The multiplication $A \times B$ is then computed using three smaller multiplications:

1. $P_1 = A_1 \times B_1$ (Product of higher halves)
2. $P_2 = A_0 \times B_0$ (Product of lower halves)
3. $P_3 = (A_1 + A_0) \times (B_1 + B0)$ (Product of sums of the halves)

The final product can be expressed as:

$$A \times B = P_1 \times 2^n + (P_3 - P_1 - P_2) \times 2^{n/2} + P_2.$$

The final result is obtained by combining these products using appropriate shifts and additions. This method drastically reduces the number of necessary multiplications, which is the primary factor in improving computational efficiency.

In this project, a 32×32-bit Karatsuba multiplier was first implemented. This module performs three 16×16-bit multiplications internally, following the Karatsuba method. Once the 32×32-bit multiplier was verified, it was used to construct a 64×64-bit Karatsuba multiplier. The 64×64-bit multiplier leverages the previously implemented 32×32-bit multiplier to perform the three required 32×32-bit multiplications, further increasing efficiency. The following is an example of such an algorithm.

Partitioning

```
assign A0 = A[15:0];
assign A1 = A[31:16];
assign B0 = B[15:0];
assign B1 = B[31:16];
```

Pipeline steps

```
assign A0_mx   = (FF_IN) ? A0_q : A0;
assign A1_mx   = (FF_IN) ? A1_q : A1;
assign B0_mx   = (FF_IN) ? B0_q : B0;
assign B1_mx   = (FF_IN) ? B1_q : B1;

assign A0_mx_d1  = (FF_SUB) ? A0_d1 : A0_mx;
assign A1_mx_d1  = (FF_SUB) ? A1_d1 : A1_mx;
assign B0_mx_d1  = (FF_SUB) ? B0_d1 : B0_mx;
assign B1_mx_d1  = (FF_SUB) ? B1_d1 : B1_mx;
assign A1A0_mx   = (FF_SUB) ? A1A0_q : A1A0;
assign B1B0_mx   = (FF_SUB) ? B1B0_q : B1B0;

assign A0B0_mx      = (FF_MUL) ? A0B0_q : A0B0;
assign A1B1_mx      = (FF_MUL) ? A1B1_q : A1B1;
assign A1A0_B1B0_mx = (FF_MUL) ? A1A0_B1B0_q : A1A0_B1B0;

assign A1B0_B1A0_mx    = (FF_SUM) ? A1B0_B1A0_q : A1B0_B1A0;
assign A0B0_mx_d1      = (FF_SUM) ? A0B0_d1 : A0B0_mx;
assign A1B1_mx_d1      = (FF_SUM) ? A1B1_d1 : A1B1_mx;

assign C                   = (FF_OUT) ? AB_q : AB;
```

Karatsuba pre-subtraction

```
// 17-bit in 2s complement
assign A1A0 = $signed({1'b0, A1_mx}) - $signed({1'b0, A0_mx});
// B0 - B1, 17-bit in 2s complement
assign B1B0 = $signed({1'b0, B0_mx}) - $signed({1'b0 , B1_mx});
↪  //changed
```

Multiplications

```
assign A1A0_B1B0 = A1A0_mx * B1B0_mx;
assign A0B0 = A0_mx_d1 * B0_mx_d1;
assign A1B1 = A1_mx_d1 * B1_mx_d1;
```

Karatsuba post-addition

```
//  (A1 - A0) * (B1 - B0) + A1*B1 + A0*B0
assign A1B0_B1A0 = A1A0_B1B0_mx + $signed({1'd0, A1B1_mx}) +
↪  $signed({1'd0, A0B0_mx}); // changed
```

Final summation

```
assign AB = {A1B1_mx_d1, A0B0_mx_d1} + {A1B0_B1A0_mx, 16'd0};
↪  //changed
```

These Karatsuba multiplier implementations are designed with pipelining and other optimization techniques to ensure high performance in hardware environments. By reducing the number of multiplications and utilizing smaller, faster multipliers, the overall multiplication process becomes more efficient, which is particularly advantageous in FHE, where large polynomial multiplications are frequent.

The Verilog modules for these multipliers are structured to be easily integrated into larger systems, allowing for scalable and efficient multiplication operations in FHE and other cryptographic applications.

5 CUDA

CUDA is a programming language that uses GPU's (Graphical Processing Unit) high level parallelism achieved by using vast number of cores. The efficiency in specific tasks are attained because of a greater throughput it produces. In GPU, compared to CPU, more transistors are present rather than components that are used for data caching and controlling the data flow. Usually, GPUs are preferred if a task can benefit more from being executed in parallel instead of depending on serialized operations.

Sequence of operations that CPU executes are called threads. In CUDA, a problem is divided into threads, and each thread performs its corresponding task. Generally, 32 threads are combined to form blocks called warps and thread blocks are the units that are formed by using these warps. Threads in a single thread block can communicate, share data, directly but more complex designs are needed for communication between thread blocks. In GPUs, thread blocks

are mapped into streaming multiprocessors which are individual processing units dedicated for executing tasks in parallel. Each streaming multiprocessor consists of ALUs (Arithmetic Logic Units), control units for these tasks. Finally, thread blocks are grouped together to form the structures called grids which represents how a problem is mapped onto the GPU hardware.

Code snippets that are designed to run on GPUs are called kernel. When a CUDA program instantiates a kernel from the host CPU, the blocks that belong to the grid are distributed to streaming multiprocessors within their capacity of execution. The threads operate on multiprocessors concurrently and multiple thread blocks can be assigned to a one streaming multiprocessors. When some of the thread blocks finish their execution, new blocks are launched on the available multiprocessors.

Thread blocks can be scheduled to available multiprocessors within a GPU, sequentially or concurrently and CUDA can execute with any number of streaming multiprocessors. This allows same codes to run on different GPU architectures and models.

A type of FFT (Fast Fourier Transform) is a Cooley-Tukey algorithm. As mentioned before, FFT reduces the time complexity of linear transformation to $O(N \log N)$ through matrix factorization. It is a recursive algorithm, however, like all recursive algorithms, it can be turned into an iterative one. Hence, Cooley-Tukey can be designed to operate on $\log N$ successive stages and it requires each stage to use the results of the preceding stage. Therefore, a synchronization point, a point for threads to consistently write the data is needed. This process prevents the race conditions and data inconsistencies for the threads that execute on parallel working independent of each other . This iterative algorithm makes Cooley-Tukey algorithm highly suitable for parallel execution because of each butterfly operation being independent of others. Below is the kernel executing the Cooley-Tukey radix-2 FFT algorithm in CUDA.

```
__global__ void parButterFlies(int num, float2* input, bool
↪  coeffToVec){
    //Will have total of num/2 threads since we are doing radix-2
    ↪  Cooley Tukey.
    int logNum = __float2int_rz(log2f((float)num));
    int offset = 1;
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    int currTid = tid;

    thread_block tb = this_thread_block();

    extern __shared__ float2 sharedMem[];

    //Load the part of the global memory for each thread.
    sharedMem[tid] = input[tid]; sharedMem[(num/2) + tid] =
    ↪  input[(num/2) + tid];

    //Synchronise to ensure the data is put into sharedMemory
    ↪  entirely.
    tb.sync();
```

```
    //Start performing the stages, if coeffToVec=true FFT and if
    ↪  not inverse FFT.
    if(coeffToVec){
        for(int i = 0; i < logNum; i++){
            currTid = tid + ((tid / offset) * offset);

            // Calculate the exponent and twiddle factor
            float angle = (tid % offset) * (-2.0f) * M_PI / (2 *
            ↪  offset);
            float2 rootCom = make_float2(cosf(angle),
            ↪  sinf(angle));

            //Perform the butterfly operations
            cooleyButterOp(sharedMem[currTid], sharedMem[currTid
            ↪  + offset], rootCom);

            offset *= 2;
            tb.sync();
        }
    }
    else{
        for(int i = 0; i < logNum; i++){
            currTid = tid + ((tid / offset) * offset);

            // Calculate the exponent and twiddle factor
            float angle = (tid % offset) * (2.0f) * M_PI / (2 *
            ↪  offset);
            float2 rootCom = make_float2(cosf(angle),
            ↪  sinf(angle));

            // Perform the butterfly operations
            cooleyButterOp(sharedMem[currTid], sharedMem[currTid
            ↪  + offset], rootCom);

            offset *= 2;
            tb.sync();
        }
        //Divide by factor of n. Notice that offset is n by the
        ↪  end of the loop.
        invCooleyNormalize(tid, offset/2, sharedMem);
    }

    //Record into global memory, here, namely input pointer.
    input[tid] = sharedMem[tid]; input[(num/2) + tid] =
    ↪  sharedMem[(num/2) + tid];
    //Synchronise the thread block to ensure data is consistent.
    tb.sync();
}
```

This algorithm can be used for doing polynomial multiplication in the process

of encrypting the messages and decrypting the ciphertexts in TFHE. Cython can be used for benefiting from CUDA in Python implementations.

6 Cython

6.1 Overview of Cython

Cython programming language is a superset of Python that enables calling C/C++ functions, declaring C/C++ variables within Python code without major changes made to the Python syntax. Hence, code written in Cython resembles Python code while the moment it is compiled, it has performance close to C/C++. Regardless of its many advantages, Python is a high-level language that lacks the computational capabilities of lower-level languages such as C/C++. Adding several aspects of C/C++ such as memory management and static typing of the variables leads to significant speed-ups and overcome bottleneck caused by performance-related issues. On top of making Python code faster, Cython can also be used to integrate existing C/C++ libraries, classes or functions so that they could be used in Python with minimal performance loss. This integration is called "wrapping" in general.

Wrapping process begins with writing in Cython code and integrating the desired C/C++ objects, libraries or class within. For clear integration and avoiding errors caused by type conflicts, Cython uses three keywords in order to separate pure Python code and C/C++ code. These keywords are `def`, `cdef`, and `cpdef`. The keyword `def` is used before a function and means that the function is a pure Python function that can be used directly from Python program. Keyword `cdef` is used before a C/C++ concept. A function or a statically typed variable and the concepts that have this keyword can't be directly used in a Python program. On the other hand, keyword `cpdef` enables a concept to both have C/C++ traits like `cdef` and to be used within Python code as is. However this approach might have some performance issues compared to previous definitions.

6.2 Wrapping CUDA Programs

CUDA is a programming language that was created by building on top of C/C++. As a result, it is possible to wrap CUDA code in order to use it in Python through Cython. However, there are several necessary additions for wrapping CUDA code compared to wrapping C/C++ code as Cython is created in order to assimilate C/C++ into Python and not CUDA.

Cython code is written using .pyx extension and it is turned into a shared library with a code snippet in Python. For achieving this, Cython code in a `.pyx` file is transformed into a `.c` or `.cpp` file (depending on whether C or C++ code is wrapped) through the Cython programming language's own optimistic static compiler. Then, newly generated `.c` or `.cpp` file is compiled into a .so file which is the desired shared module. This module can be imported from Python program just like an ordinary Python library and be used similarly.

To begin with, CUDA has its own compiler called `nvcc` and it has a special `.cu` extension. The .cu extension and nvcc compiler are not already defined in Cython unlike their C or C++ counterparts. So, it is necessary to define the

`.cu` extension and integrate the `nvcc` compiler accordingly from scratch in order to wrap CUDA code in Cython. In addition, Python has a locking mechanism called global interpreter lock (GIL) that makes concurrent thread execution impossible and creates a bottleneck for multithreaded programs. Considering that CUDA is used for multithreaded applications, it is necessary to use `nogil:`, the Cython functionality that removes GIL for `cdef` functions for maximum performance. Also, unlike C and C++, CUDA utilizes GPU memory. In scope of this, some CUDA functions manage GPU memory transfer data between CPU and GPU. These functionalities regarding GPU are not inherent in C/C++ which means they are not optimized for Python through Cython, leading to some significant performance losses.

For optimized GPU manipulation techniques of CUDA to be used in Python without significant performance losses, usage of special modules like `pycuda` or other optimizations are needed. With the usage of these, performance losses are nearly restricted to 50% when arrays of sizes between 2^{10} and 2^{22} are given as inputs to CUDA through Cython and it gets better with closer to 2^{20}. These sizes are enough for cryptographic applications in schemes like TFHE and CKKS where usually, ring modulus being in the form of $X^N + 1$ with N being a power of two and satisfying $10 \leq N \leq 15$. Below, some execution times are presented in comparison:

Data Size	**CUDA**	**Cython**
2^{10}	0.13 ms	0.204 ms
2^{13}	0.14 ms	0.208 ms
2^{15}	0.17 ms	0.23 ms
2^{20}	0.14 ms	0.17 ms

Table 1: Execution Times with Different Data Sizes

It can be seen that, Cython can allow for efficient usage of CUDA libraries in Python.

7 Conclusion

Implementations were done related to HE (Homomorphic Encryption) algorithms and components of popular schemes using various programming languages. Our main goal was to find parts that can be accelerated. To achieve this, extensive literature review was conducted for HE schemes. We acknowledge that homomorphic encryption is a vast area of research with room for further exploration. We suggest that future work to focus on investigating memory optimization and hybrid scheme implementations.

References

Bossuat, J.-P., Mouchet, C., Troncoso-Pastoriza, J., & Hubaux, J.-P. (2020). Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys. https://eprint.iacr.org/2020/1203

Cheon, J. H., Han, K., Kim, A., Kim, M., & Song, Y. (2018). Bootstrapping for approximate homomorphic encryption. https://eprint.iacr.org/2018/153

Chillotti, I., Gama, N., Georgieva, M., & Izabachène, M. (2017). Tfhe: Fast fully homomorphic encryption over the torus [Accessed: 2024-08-25]. *IACR Cryptology ePrint Archive*.

Han, K., & Ki, D. (2019). Better bootstrapping for approximate homomorphic encryption. https://eprint.iacr.org/2019/688

Joye, M. (2022). Sok: Fully homomorphic encryption over the [discretized] torus [Accessed: 2024-08-25]. *IACR Transactions on Cryptographic Hardware and Embedded Systems*. https://marcjoye.github.io/papers/Joy22dtorus.pdf

Zama AI. (2022). Tfhe deep dive - part 1 [Accessed: 2024-08-25]. https://www.zama.ai/post/tfhe-deep-dive-part-1

Zama AI. (2023). Tfhe deep dive [Accessed: 2024-08-25].