4-18-2024

# GPU Acceleration of Homomorphic Encryption Scheme Conversion

Emilio Ramon Quaggiotto
*University of Windsor*

# GPU Acceleration of Homomorphic Encryption Scheme Conversion

By

**Emilio Quaggiotto**

A Thesis
Submitted to the Faculty of Graduate Studies
through the Department of Electrical and Computer Engineering
in Partial Fulfillment of the Requirements for
the Degree of Master of Applied Science
at the University of Windsor

Windsor, Ontario, Canada

2024

GPU Acceleration of Homomorphic Encryption Scheme Conversion

by

Emilio Quaggiotto

APPROVED BY:

---

I. Saini
School of Computer Science

---

H. Wu
Department of Electrical and Computer Engineering

---

M. Mirhassani, Advisor
Department of Electrical and Computer Engineering

9th of February, 2024

## DECLARATION OF ORIGINALITY

I hereby certify that I am the sole author of this thesis and that no part of this thesis has been published or submitted for publication.

I certify that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis and have included copies of such copyright clearances to my appendix.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other University or Institution.

ABSTRACT

A new category of Fully Homomorphic Encryption (FHE) schemes, first presented in CHIMERA, offers methods of converting between different schemes to circumvent the disadvantages found inherently in each individually. The work done in PEGASUS introduced a similar agile Encryption framework, via an improved repacking algorithm that converts FHEW Ciphertexts into a CKKS Ciphertext. Using PEGASUS as a starting point, the goal was to speed up their novel repacking algorithm through the use of additional GPU computation. In doing so, the accelerated software created achieved a speedup of approximately 50x when comparing replaced portions of the algorithm now running in the GPU, and 25% reduction in the overall repacking algorithm runtime compared with the CPU only computation. This work focuses on the parts of the repacking algorithm that have thus far been unexplored by other GPU acceleration works, while leaving alone some other Homomorphic functions that have already been proven amenable to GPU acceleration. Also, the acceleration function is integrated in such a way that it builds alongside the current Open-Source PEGASUS framework, with minimal invasiveness, which increases potential for coupling this with complimentary works on acceleration of Homomorphic Encryption.

ACKNOWLEDGEMENTS

This couldn't have been done without the support of my family and friends, and the help of my Advisor, and other Mentors. Thank you everyone for supporting me throughout.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

# CHAPTER 1

# *Introduction*

In a world that is increasingly more connected, and reliant upon safe, and secure technology, the topic of Encryption is more important than ever. This is contrasted, at the same time, by the fact that Security-breaching technologies like Quantum Computing are also becoming a realistic threat to our current level of security. This combination of increased connectivity, use, and threats have resulted in a greater desire to improve encryption technologies to meet these new needs and threats. New breakthroughs in Homomorphic Encryption help solve two of the major needs that are emerging in a data centric world. The first being quantum hardness, which implies that the encryption scheme is based on a problem that is difficult even for quantum computers. Secondly, Homomorphic Encryption can also provide no-trust data privacy, which means being able to store, and have data processed remotely without the need to trust the remote device's security. This has major implications for consumer's rights, because as it stands now, with non-homomorphic encryption, if a service is offered that processes a consumer's data, the consumer must at least trust the processor, since the data must first be decrypted to be processed. Probably the industry most interested in taking advantage of a security feature like this is healthcare, which has stringent consumer privacy laws and regulations that make offering data-centric services problematic. One of the major issues with Homomorphic Encryption as it stands today is processing speeds. Compared to the encryption methods typically used today (AES, ECC), Homomorphic operations can be slow to the point of being practically unusable, which puts them in a category where they are not feasible replacements to non-quantum safe encryption. This also shows a great

opportunity to be able to create faster implementations of these encryption schemes, which can improve a variety of security concerns.

## 1.1 Homomorphic Encryption

An Encryption Scheme is considered "Homomorphic" if you are able to perform functions on encrypted data without first decrypting it i.e:

$$Decryption(Homomorphic\,Function(Encrypted(x),\ Encrypted(y))) = Function(x,\ y)$$

There were limitations on earlier homomorphic schemes that attempted this, which is why there is a distinction between "Fully" homomorphic schemes and "Partially", or "Somewhat" homomorphic Schemes, but the schemes mentioned here are all Fully Homomorphic Encryption (FHE) schemes. The issue is that in order to encrypt, an "error", or "noise", is added to the ciphertext, which grows upon operations being performed, if this error grows too large, eventually decryption will be unsuccessful. Despite this, a FHE scheme can perform an unlimited number of operations while still being possible to decrypt. The first work to achieve this was in Gentry's work[16], which has been foundational for further research and development of other FHE schemes. The key concept that Gentry was able to introduce was what is called "bootstrapping", which essentially provides a method of Homomorphically decrypting the ciphertext, which resets the noise down to a level where further homomorphic functions can be called without causing a failure in decryption. Since this original work, which was based on Ideal-Lattices [16] there have been many improvements, and schemes to follow. Most of these new schemes have been based on the Learning with Error (LWE), Ring Learning with Error (RLWE) problem, which are similar to ideal lattices originally used. The key issue that has followed every iteration of these FHE schemes is that the computation is simply much too expensive to be practical in use[4]. Additionally, each scheme brings a set of advantages and disadvantages over the others, causing users to need to make trade-offs if they wish to setup a homomorphic program. This can be seen very clearly in the Figure. 1, from a FHE survey[22] that shows some of the most widely used FHE schemes today. It is obvious, looking

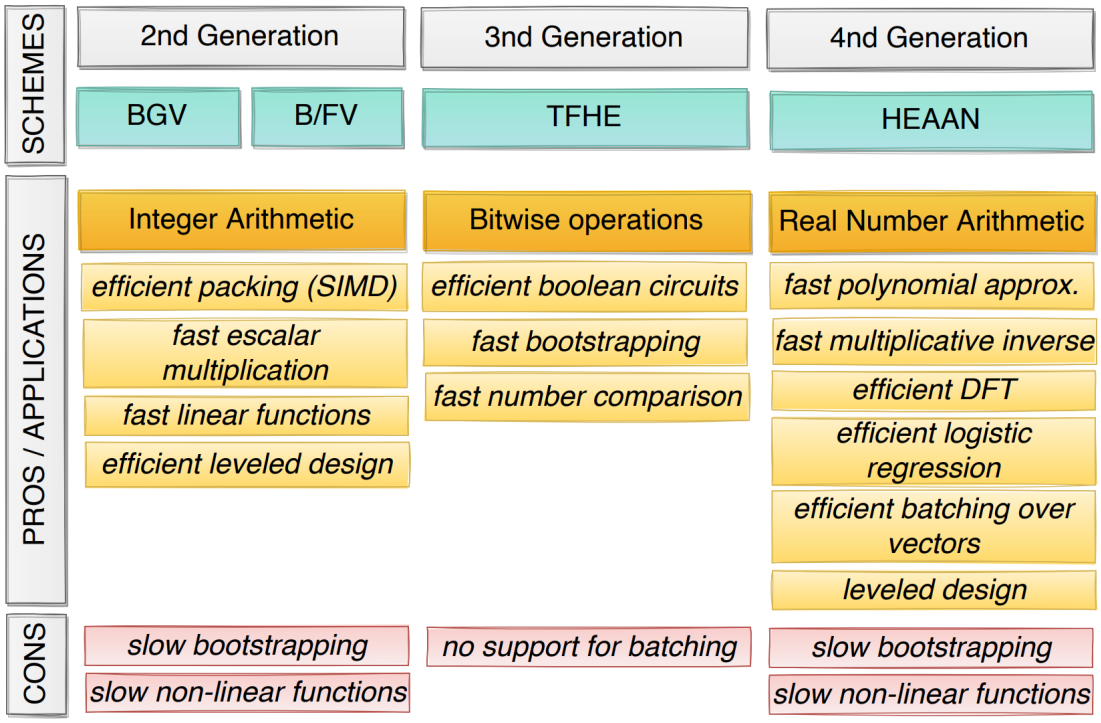



FIGURE 1: FHE Scheme Generations with Key Differences[22]

at the chart, that depending on the type of data, and type of functions that are desired to be performed, there will likely have to be some compromise when choosing a FHE scheme to use. Fortunately, there is another group of FHE frameworks that have a promising way of getting around the limitations that arise from using a single FHE scheme. These "Combination" FHE frameworks find ways to convert the same value between different schemes without decryption, in order to be able to use the best data structure for the Homomorphic Function that the user wishes to use. The first example found that employs an agile FHE framework like this was CHIMERA, which bridged the BF/V, TFHE, and CKKS scheme [7]. The work most focused on for this though is the PEGASUS scheme, which allows for conversions between the FHEW (Predecessor to TFHE [22]), and CKKS encryption schemes[21].

### 1.1.1 FHEW

The FHEW Scheme, created by Ducas and Micciancio, is one of the early binary based FHE schemes that followed Gentry's work, and is the predecessor of the TFHE scheme which is now more commonly used. The main goal of FHEW was to provide a method of bootstrapping that was much faster than any of the work that had been done at that time, as that is generally a bottleneck in most FHE schemes, even now. They achieved this by creating a LWE based FHE structure that works very effectively at a binary level with XOR operations. This combined with a more efficient bootstrapping function provided an effective solution to homomorphic binary operations[14]. One of the main issues with FHEW, and other bit-wise FHE schemes is when larger arithmetic operations are required (Addition/Multiplication), as these will need large circuit depths of Boolean operations in order to function.

### 1.1.2 CKKS

CKKS is a RLWE based encryption that follows a different path than most FHE schemes, mainly in how it is an "Approximate" system, meaning that there there can be bit loss during homomorphic functions. This is due to the fact that it mixes its low bits (least significant bits) with the encryption noise and can essentially treat the noise (encryption error) as precision lost normally found in Floating/Fixed point arithmetic. The bootstrapping method used essentially scales up the ciphertext, to a new and larger modulus. This allows the user to continue performing operations, while maintaining the bit precision of the decrypted value[10].

### 1.1.3 CHIMERA

CHIMERA was developed to provide a coherent framework for working with multiple FHE schemes with the same data. This was achieved by unifying the plain-text space of several RLWE FHE schemes, including the B/FV, TFHE, and CKKS schemes. The reasoning behind this was that, as mentioned previously, each of the upcoming schemes had advantages and disadvantages in how data could be processed and stored

[22]. This work provided a solid foundation for improvements, and further research in this topic. The clear goal of this unification can also be seen in their paper as well, providing a clear path for using Homomorphic Schemes with Deep Networks[7]. This is something that has proven difficult without a Scheme-Conversion type FHE, as you run into one of two problems; You have to deal with non-vectorizable data like TFHE[11], or you cannot easily perform non-linear functions like in CKKS[10].

### 1.1.4 Microsoft SEAL

Microsoft SEAL is an open-source library that aims to provide a simple application programming interface (API) to some of the most used Homomorphic Encryption schemes. Additionally, to this, their aim is to facilitate some of the more difficult Mathematical practices required in getting started with a HE scheme, as by doing this work for the user it lowers the barrier to entry in using these schemes. The majority of the library is written in C/C++, which allows for use across a large variety of platforms, and it also has implemented many of the most commonly used FHE schemes, like CKKS, BFV, and BGV. Additionally, many low-level mathematical tools are implemented in order to provide an improved performance over many simpler implementations, like using the Residue Number System (RNS) for large modulus values, Number Theoretic Transform (NTT) for improved polynomial multiplication, and effective use of Barrett, and Montgomery modulus functions when the situation is suitable[29]. For these reasons and more, Microsoft SEAL has become an attractive platform for building applications that make use of Homomorphic Encryption.

## 1.2 Previous Works

### 1.2.1 PEGASUS

PEGASUS, which is the main precursors to this work, was able to continue CHIMERA's success in unifying FHE Schemes. This trend was again in the pursuit of creating tools that would enable the design of Fully Homomorphic machine learning algo-

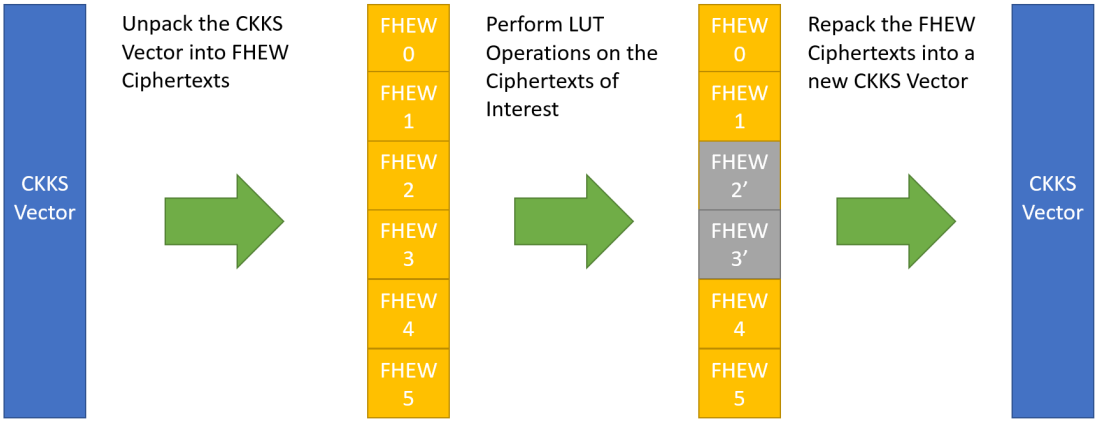



FIGURE 2: Basic PEGASUS Workflow Example

rithms. At a high level, one of the key differences between the two is that PEGASUS focuses solely on the CKKS and FHEW scheme (Which is similar to TFHE but is LWE based instead of (R)LWE), leaving the B/FV scheme that CHIMERA includes out of their framework. That said, the novelty of this work is found in the lower-level improvements. These mathematical changes greatly improve the conversion process, both in terms of time, and space complexity[21]. "To the best of our knowledge, this is the first work that supports practical K-means clustering using HE in a single server setting."[21] Another notable part of the PEGASUS software is that it makes use of Microsoft SEAL's homomorphic library. This allows for low-level speedups using the NTT and RNS functionality of SEAL, and the authors have a public, open-source project available on Github[5].

At its most basic, PEGASUS offers the following standard workflow seen in Figure. 2. Start with a vector of Real values in the CKKS (RLWE) scheme. When a Non-linear lookup-table (LUT) function is to be used, convert the CKKS vector ciphertexts to a corresponding number of FHEW (LWE) ciphertexts, perform the LUT function on any or all LWE ciphertexts, and convert back. Using this paradigm, the user has the advantage of performing simple arithmetic (addition/multiplication) in a vector format using CKKS, but they can also take advantage of the LUT operations

for things like Sigmoid, or ReLU functions which are difficult to perform using solely the CKKS format[21].

### 1.2.2 GPU Acceleration

GPU (Graphics Processing Unit) acceleration for any FHE scheme that is based on the LWE problem is a feasible option due to the highly parallel structure of the underlying lattice math. For instance, the PEGASUS Framework specifically expresses that they'd like to see performance upgrades using hardware like FPGAs or GPUs [21]. Improvements from GPU hardware in previous implementations can see speed ups of 4.5x [35], all the way up to 378.4×[30]. These numbers, on their own are somewhat arbitrary unless one looks at a specific Encryption Scheme, with specific parameters, and a specific CPU being used, but it is still important to note as it shows that there is generally a great deal of potential in using GPUs to accelerate FHE schemes. One commonality with most the papers that were found on this is that there is a focus on the low-level arithmetic, particularly modular multiplication, which is a very slow and common operation in almost all FHE schemes[34] [31] [30]. This makes sense as a focus, as by implementing the speedup of one function, you can see a large amount of improvement over many areas of the FHE Scheme.

There are also a host of works that focus solely on speed up of the Number Theoretic Transform (NTT), which is generally used to speed up Modular Multiplication, and other aspects of FHE schemes[9][26][15][36]. It would have been convenient if there was a simple 64-bit NTT library in Cuda publicly available, but one that met the criteria of this project was not found. This was a somewhat unlikely scenario because this work aimed to parallelize the repacking function from a high-level functional perspective, which takes on the issue of running multiple NTTs in parallel, instead of using the whole GPU processing for a single, larger NTT, which there is far less work on[36].

# CHAPTER 2

# *Mathematical Background*

## 2.1 Mathematical Notation

The cyclotomic polynomial $\Phi_n = X^n + 1$, will be used in describing $R_n = \frac{\mathbb{Z}[X]}{q\Phi_n}$, which in turn is used in representing the polynomial ring $R_{n,\ q} = R_n/qR_n$.

Basic Arrays and Matrices will be described by $M[a,\ b, \ldots,\ c]$ where $a,\ b, \ldots,\ c$ describe the dimensions and their corresponding lengths. Values denoted with a hat refer to CKKS RLWE based ciphertexts ($\hat{ct}$), whereas a check refer to those encrypted using the LWE FHEW scheme ($\check{ct}$). Plaintexts of both schemes will be referred to as simple arrays already previously described.

Algorithms will use the following for loop style: $For(Initialization;\ Condition;\ PostLoopOperation$ where $Initialization$ is done before the loop starts, $Condition$ is the condition checked before each loop, and $PostLoopOperation$ is performed after every loop before starting a new one or checking the $Condition$.

The $a << b$ and $a >> b$ operations denote bit shifting $a$ by $b$ bits to the left and right respectively. The brackets $\lceil \cdot \rceil$ round a value up, and $\lfloor \cdot \rfloor$ down, whereas $[\cdot]_q$ indicates applying modulus $q$ to the value. The $min(a,\ b, \ldots,\ x)$ function returns the smallest value given, and the $max(a,\ b, \ldots,\ x)$ returns the largest. Adding a double plus like $a++$ is shorthand for the increment $a = a + 1$.

## 2.2 Learning with Error Problem

Difficult mathematical problems are the basis of any encryption scheme. From these problems, encryption constructs can be created and based on, with the assurance that solving that problem will be too hard statistically[19]. With that in mind, the Learning with Error problem, (LWE) first introduced by Regev and its related mathematical problems like the Ring Learning with Error (RLWE) can provide a level of security that as of now appears very concrete, even against an attacker with Quantum Computing[27]. A very basic overview of the LWE problem can be seen if one first considers a series of Linear Equations. For example:

$$2a + 3b + 5c = 4$$

$$a + 5b + c = -3$$

$$a + b + 9c = 8$$

Now obviously, these can be solved very straightforwardly under the assumption that you have enough equations (ie. Samples). Now the LWE problem simply adds a degree of uncertainty, or error to each of these instances. For example:

$$2a + 3b + 5c \cong 4 \; or \; 2a + 3b + 5c = 4 + e_1$$

$$a + 5b + c \cong -3 \; or \; a + 5b + c = -3 + e_2$$

$$7a + b + 9c \cong 8 \; or \; 7a + b + 9c = 8 + e_3$$

This has now changed the problem significantly, and with the error it becomes more difficult to solve with the higher the degree as the error when solving will add exponentially as you add more equations to solve it.[28] This problem can be stated as either solving for the variables, which can be equivalent to solving for a secret vector $s$=$\{a,\ b,\ c\}$, or by sorting between a random distribution of equations, and equations that are part of those generated with the same variables. These two problems are called the Search and Decision LWE respectively. With the general idea, we can now construct a simpler way of explaining this problem, namely with a secret key s of

length n, and m≥n equations of the LWE problem, a Matrix A, and vector b can now be generated such that A is an m x n matrix, and b is of length m. This can be shown simpler now as

$$As \cong b$$

Or conversely,

$$As = b + e$$

The problem now breaks down to simply solving for s with the noise present in the system. So far, the LWE problem has been portrayed at a high-level, and to proceed we must now introduce further notation. Generally, if you add a modulus q to all calculations, and only use integers, we can say that variables in this system belong to $\mathbb{Z}_q$, and that the errors are sampled from the distribution $\chi_q$ where the distribution is generally Gaussian, and small enough that its very unlikely to sample an e from $\chi_q$ such that $e \geq q/4$.

$$\chi_{err}\text{: Error Distribution to sample from}$$

$$LWE_{q,n} = \begin{cases} s \in \mathbb{Z}_q & \text{length n,} \\ b \in \mathbb{Z}_q & \text{length n,} \\ A \in \mathbb{Z}_q & \text{size} \geq \text{n·n,} \\ e \leftarrow \chi_{err} & \end{cases} \tag{1}$$

With a basic definition and understanding of the LWE problem, we can now define the Ring Learning with Error (RLWE) problem which is very similar. The main difference here is that we now use a polynomial ring as the data in the LWE system. This is created by adding a polynomial modulus f(x) which is an irreducible polynomial, generally taken as $x^n \mp 1$. This, combined with the regular value modulus means that there can be polynomials < f(x), which has values of modulus q. Put more simply, we are in a system of [modulus q modulus f(x)]. We can now put together a simple RWLE system:

$\mathbb{Z}_q$ - integer ring

$f(x) = x^n \mp 1$ - irriducible polynomial

$R_{q,n} = \mathbb{Z}_q$ modulo f(x) - Polynomial Ring

$\chi_{err}$: Error Distribution to sample from

$$RLWE_{q,n,N} = \begin{cases} s \in \mathbb{Z}_q & \text{length N,} \\ b \in \mathbb{Z}_q & \text{length N,} \\ A \in \mathbb{Z}_q & \text{size} \geq \text{N·N,} \\ e \leftarrow \chi_{err} \in R_{q,n} & \text{length N} \end{cases} \tag{2}$$

## 2.3 Common Algorithms

### 2.3.1 Residue Number System (RNS)

The Residue Number System (RNS) is a useful tool in computing modulus for large numbers, especially when the size of the modulus becomes restrictive in the system that the computation is taking place. This system works based on the Chinese-Remainder Theorem, which effectively allows for reconstructing of a certain range of integers based on a set of relatively-prime moduli. This allows us to evaluate operations in $Z_Q$ at a large modulo $Q$, but broken down into $n$ smaller co-prime moduli $q_i$[23].

$$\begin{aligned} Q &= \prod_{i=1}^{i=n-1} q_i \\ A \in Z_Q &= (a_i \in Z_i, \ldots, a_{n-1} \in Z_{n-1}) \\ B \in Z_Q &= (b_i \in Z_i, \ldots, b_{n-1} \in Z_{n-1}) \\ [A+B]\ mod\ Q &= [(a_i + b_i)\ mod\ q_i, \ldots, (a_{n-1} + b_{n-1})\ mod\ q_{n-1}] \\ [A*B]\ mod\ Q &= [(a_i * b_i)\ mod\ q_i, \ldots, (a_{n-1} * b_{n-1})\ mod\ q_{n-1}] \end{aligned} \tag{3}$$

It is clear from Equation. 3 that a value in $Z_Q$ can instead be represented by an array of smaller values with their own residue modulus, and further so that arithmetic operations (addition, multiplication), can be performed using these as well[32].

### 2.3.2 Barrett Reduction

The Barrett reduction algorithm was first created in the search to do fast public-key encryption operations [6], and continues to be useful to this day. This algorithm is particularly useful when performing many modular reductions with the same modulus as it makes use of pre-computed factors that speed up the calculation. The pre-calculation algorithm for a modulus $n$ can be seen in Algorithm 1, and the actual reduction can be seen in Algorithm 2.

**Algorithm 1** Barret Reduction Precomputation

**Input:** modulus **n**

**Output: k2, r**

1: chose $k$ such that $2^k > n$

2: $r = \lfloor \frac{4^k}{n} \rfloor$

**Algorithm 2** Barret Reduction Computation

**Input:** input **x** modulus **n**, precalculated **k2, r**

**Output: x mod n**

1: $t = x - \lfloor \frac{xr}{4^k} \rfloor n$

2: **if** $t < n$ **then**

3: return $t$

4: **else**

5: return $t - n$

6: **end if**

[6]

### 2.3.3 Fast Fourier Transform (FFT)

A basic version of the Discreet Fourier Transform can be seen in Equation 4, and the Inverse FFT (IFFT) in Equation 5.

$$X_k = \sum_{n=0}^{n=N-1} x_n e^{-\frac{2\pi i}{N} nk} \tag{4}$$

$$x_k = 1/N \sum_{n=0}^{n=N-1} X_n e^{\frac{2\pi i}{N} nk} \tag{5}$$

The Fast Fourier Transform (FFT) comes from the work done originally by Cooley and Tukey, which implemented a faster version of the Discreet Fourier Transform (DFT)[12]. The version of the FFT used in this paper follows the format of a bit-reversal radix-2 FFT based on Cooley and Tukey's work, which was taken and modified from both PEGASUS[21], and Microsoft SEAL's code[29], and can be seen in Algorithm 3. The main advantages of this algorithm is it allows for computations to be done in-place, meaning it does not require additional memory[13], and allows it to be broken down into log(N) computations of size N/2 of the same algorithm, as opposed to the naïve algorithm using $N^2$ computations of size N. In addition to the base algorithm it is sometimes necessary to implement a bit-reversal of the output array depending on the direction (forward or reverse). This was achieved using the algorithm given in PEGASUS[5], and can be seen in Algorithm 4.

**Algorithm 3** General Radix-2 FFT Program Flow

**Input:** **X[N]**, Pre-computed table **Z[N+1]**

**Output:** **X[N]**

**function** BUTTERFLY($x_0,\ x_1,\ z$)
    Modifies $x_0,\ x_1$ in place using the pre-computed value $z$ depending on the type (FFT/NTT) and direction (Forward/Reverse)
**end function**
Main Loop
$w = 0$
**for** $h = N/2; h > 2; m <<= 1, h >>= 1$ **do**
    $x_0 = 0$
    $x_1 = x_0 + h$
    **for** $r = 0; r < m; ++r$ **do**
        **for** $i = 0; i < h; i += 4$ **do**
            $Butterfly(X[x_0], X[x_1], Z[z])$
            $Butterfly(X[x_0+1], X[x_1+1], Z[z])$
            $Butterfly(X[x_0+2], X[x_1+2], Z[z])$
            $Butterfly(X[x_0+3], X[x_1+3], Z[z])$
            $x_0 = x_0 + 4$
            $x_1 = x_1 + 4$
            $z = z + 1$
        **end for**
        $x_0 = x_0 + h$
        $x_1 = x_1 + h$
    **end for**
**end for**
Middle Loop
$m = N/4$
$x_0 = 0$
$x_1 = x_0 + 2$
**for** $r = 0; r < m; ++r$ **do**
    $Butterfly(X[x_0], X[x_1], Z[z])$
    $Butterfly(X[x_0 + 1], X[x_1 + 1], Z[z])$
    $x_0 = x_0 + 4$
    $x_1 = x_1 + 4$
    $z = z + 1$
**end for**
Final Loop
$m = N/2$
$x_0 = 0$
$x_1 = x_0 + 1$
**for** $r = 0; r < m; ++r$ **do**
    $Butterfly(X[x_0], X[x_1], Z[z])$
    $x_0 = x_0 + 2$
    $x_1 = x_1 + 2$
    $z = z + 1$
**end for**

Note: Depending on the type, a bit-reversal of the array may be required here

**Algorithm 4** In-Place Bit Reversal from PEGASUS

**Input: Vector X of length N**

**Output: Vector X of length N in bit-reversed order**

```
1: j = 0
2: for i = 0; i < N − 1; i++ do
3:     bit = length >> 1
4:     while bit < j do
5:         j = j − bit
6:         bit = bit >> 1
7:     end while
8:     j = j + bit
9:     if i < j then
10:        Swap(X[i], X[j])
11:    end if
12: end for
```

### 2.3.4 Number Theoretic Transform (NTT)

The Number Theoretic Transform (NTT) is essentially a version of the DFT that has been applied to a finite field using integer math and a modulus p. Due to this, the actual structure of the algorithm is the same as that found in the FFT section, with the caveat being that all math is performed with respect to the modulus p, and an n-th root of unity ($\omega$) is used to replace the omega value. A very basic description of this can be found in Equation 6[18].

$$X_k = \sum_{n=0}^{n=N-1} x_n \omega^{nk} \text{mod p} \tag{6}$$

One of the main attributes that makes the NTT very attractive for cryptography is that it allows for simpler and more computationally-parallel polynomial multiplication when dealing with polynomial rings. This can be seen in Equation. 7, as the NTT has more options for parallelizing the computational load of computing the multiplication than doing it in the normal domain.

$$
\begin{aligned}
&c,\ a,\ b \in R_{q,\ n} \\
&c = a * b \\
&c = INNT^{Z^{-1}}(NTT^{Z}(a) * NTT^{Z}(b))
\end{aligned}
\tag{7}
$$

## 2.4 PEGASUS Repacking

Taking a closer look at the conversion from FHEW → CKKS (or LWE → RLWE), the authors of PEGASUS have created a novel approach over CHIMERA's method, based around a Linear transform with the matrix formed by the individual LWE ciphertexts. The matrix formed by the LWE ciphertexts can be described by its width (Length of the ciphertexts) and height (number of ciphertexts to be repacked). This approach followed previous work in the area of Homomorphic Linear Transforms, but in adding extra steps, is able to achieve a more flexible product, that's computation cost is independent of the number of ciphers it needs to repack when that number exceeds the size of the ciphertext (i.e a "tall" matrix)[21]. A programmatic version of the original PEGASUS repacking algorithm can be seen in Algorithm. 5.

$Encoding_{CKKS}(a[dim],\ b)$: An encoding of the vector of real values $a[dim]$ with the scaling factor $b$ using the CKKS Encryption scheme. This encodes the real vector into a ring element such that the output plaintext $b_u[n] \in R_{n,q}$. More details on this operation can be found in the original works[10][17].
$Rotation^{x}(a[\cdot])$: Rotates the array $a[\cdot]$ to the left by $x$ values, negative values rotate right.
$Rotation^{x}_{CKKS}(\hat{a}, \hat{b})$: Given the CKKS Ciphertext $\hat{a}$ with the underlying plaintext $A[\cdot]$, and the Rotation Key $\hat{b}$, return the CKKS ciphertext $\hat{c} = Encryption(Rotation^{x}(A[\cdot]))$ (The encryption of the rotated plaintext of $\hat{a}$).
$Rescale(\hat{a}, x)$: Given the CKKS Ciphertext $\hat{a}$ with the underlying encoding $Encoding_{CKKS}(a[dim],\ y$
return a ciphertext $\hat{b} = Encryption(Encoding_{CKKS}(a[dim],\ y/x))$ which has a smaller modulus.

**Algorithm 5** PEGASUS Repacking

**Input:** LWE Cipher Matrix $M[l,\ n]$, Rotation Key $\hat{ct}_{RotK}$, Repacking Key $\hat{ct}_{RPK}$, scaling factor $\triangle_r\prime > 0$, vector $t[l]$

**Output:** RLWE Ciphertext of repacked LWE Ciphers $\hat{ct}_{out}$

Preliminary

$K = min(l,\ n)$

$J = max(l,\ n)$

$B = \sqrt{min(l,\ n)}$

$G = \lceil \frac{min(l,\ n)}{B} \rceil$

$\hat{ct}^j_{RPK} = Rotation^j_{CKKS}(\hat{ct}_{RPK}, \hat{ct}_{RotK})$

Tiling

$tile[K, J]$

**for** $k = 0; k < K; k++$ **do**

  **for** $j = 0; j < J; j++$ **do**

    $tile[k,\ j] = M[j\ mod\ l,\ j + k\ mod\ n]$

  **end for**

**end for**

Giant-Steps

$\hat{ct}_{out} = NULL$

**for** $g = 0;\ g < G;\ g++$ **do**

Baby-Steps

  $\hat{ct} = NULL$

  **for** $b = 0;\ b < B\ and\ (b * G) < J;\ b++$ **do**

    $tile_{current} = Rotation^{-g*B}(tile[g* B + b])$

    $\hat{ct}_{temp} = Encoding_{CKKS}(tile_{current}, \triangle_r\prime)$

    $\hat{ct}_{temp} = \hat{ct}_{temp}\hat{ct}^b_{RPK}$

    $\hat{ct} = \hat{ct} + \hat{ct}_{temp}$

  **end for**

  $\hat{ct}_{out} = \hat{ct}_{out} + Rotation^{g*B}_{CKKS}(\hat{ct}, \hat{ct}_{RotK})$

**end for**

Sum Columns

**if** $l < n$ **then**

  $\gamma = log(n/l)$,

  $\hat{ct}_{temp} = \hat{ct}_{out}$

  **for** $j = 1;\ j \leq \gamma; j++$ **do**

    $\hat{ct}_{temp\ rotated} = Rotation^{l2^j}_{CKKS}(\hat{ct}_{temp}, \hat{ct}_{RotK})$

    $\hat{ct}_{temp} = \hat{ct}_{temp} + \hat{ct}_{temp\ rotated}$

  **end for**

  $\hat{ct}_{out} = \hat{ct}_{temp}$

**end if**

Rescale

$\hat{ct}_{out} = Rescale(\hat{ct}_{out}, \triangle_r) + Encoding(t, \triangle_r\prime)$

## 2.5 GPU Architecture

Nvidia's CUDA programming language was used for the majority of the programming on the GPU. When discussing the relationship between the GPU and main CPU processor *Device* refers to the GPU, whereas *Host* refers to the CPU which is in keeping with the CUDA programming model. The GPU programming model used here can generally be thought of in terms of *threads*, *blocks*, and *kernels*, since grids were not used as a part of this. Threads refer to the lowest level of computation, similar to a CPU thread running operations in sequence. Blocks are groupings of threads that allow synchronization of all threads in a single block. Lastly a Kernel is an instantiation of function that is to be run on the device, with a Block/Thread count that is specified at runtime. A simple summary would be, work is done in threads, synchronization is done in blocks, blocks are run, and grouped, in kernels. Threads and blocks can also be organized in 1 to 3 dimensional grids, which can be used to assist in matching the parallel processing logic to the data to be processed.

Another important aspect of GPU programming involves memory management, which follows a similar structure to that seen in the programming model, but with some additions. *Registers* are memory that only a single Thread can access, *Shared* variables are accessible to all Threads in a Block. Lastly, there is also *Global*, and *Constant* memory, which are both accessible to any thread in any block. In general, these grow in size as you move higher up the hierarchy, while also becoming slower to access, i.e: Registers are the fastest memory for a thread to access, but are very limited in number, whereas Global memory is large, but slower to access. The exception to this is Constant memory, which is fast to access even though it has a broad scope, but it is also limited in size[1]. An additional memory type that is not used in this work is *Texture* memory, which offers access at any level like Global or Constant Memory. An overview of this basic structure can be seen in Figure. 3, where the processing models (thread/block/kernel) have been put together with their memory counterparts (Registers/Shared/Global/Constant).

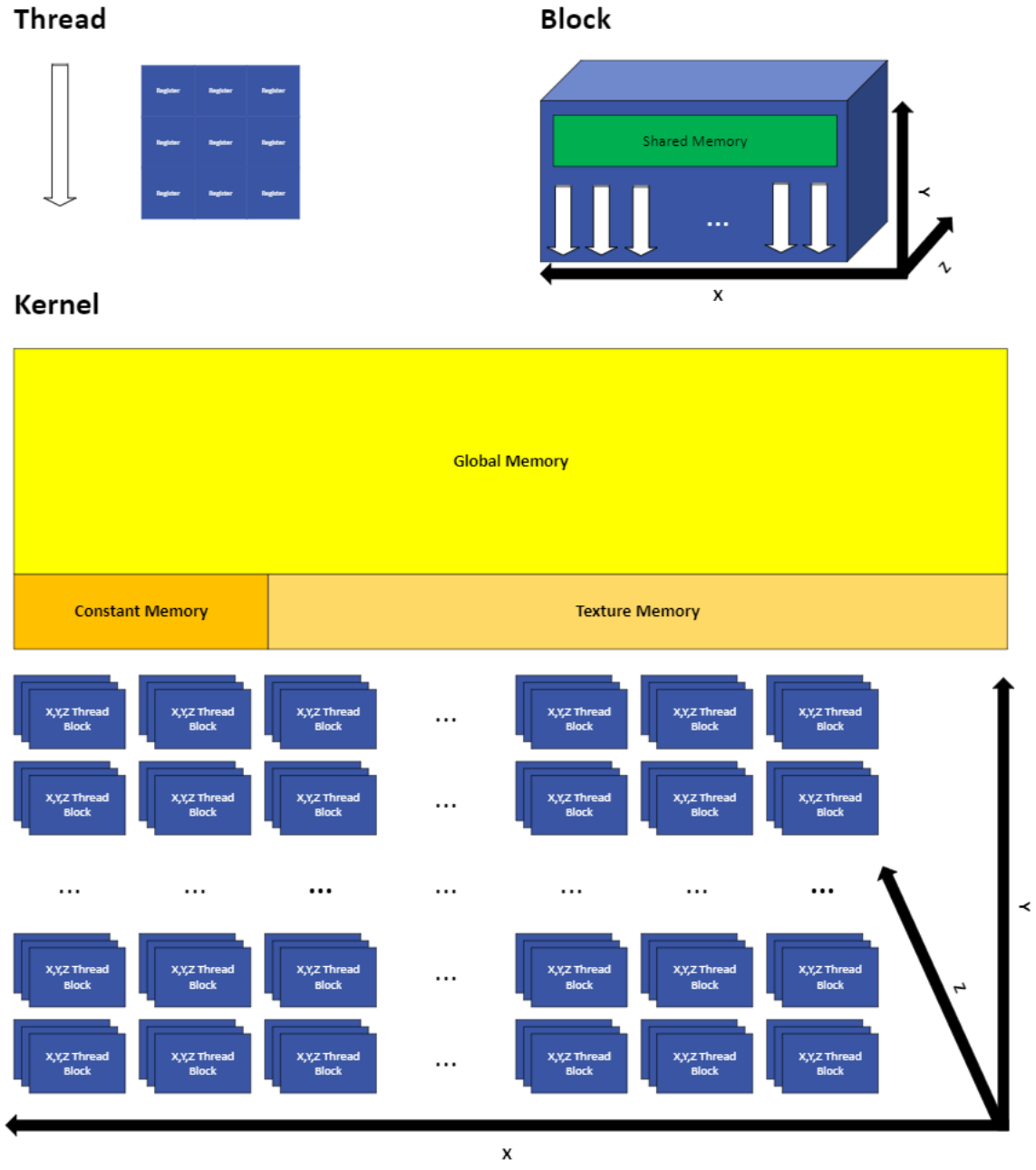



FIGURE 3: CUDA programming Memory and Processing Model

# CHAPTER 3

# *Methodology*

In looking to achieve faster homomorphic conversions, a base set of working software was a great boon in providing a starting point. This allowed for a simple decision in using PEGASUS as a base framework to be added onto. The existing PEGASUS software implementation already integrated a number of appealing systems, most notably of those being Microsoft SEAL. The accelerated functions, or accelerated software, is built alongside the base PEGASUS software, with minor modifications. This allows us to leverage the existing Microsoft SEAL and PEGASUS functions for things like generating parameters (Secret keys, LUT's) and gives us a functional entrance. This additional accelerated functionality uses mainly the CUDA programming language, which provides control over a GPU program connected to the Host CPU running PEGASUS. A GPU was chosen as the acceleration hardware because it allows for rapid prototyping, flexibility, and is an easy to find component that is already compatible with the environment PEGASUS was already made to run with, an everyday desktop computer.

With the goal of accelerating Homomorphic conversion in mind, the repacking algorithm was chosen as the most impactful function. The reasoning behind this is twofold; Firstly, the conversions between Homomorphic Encryption schemes (in this case FHEW→CKKS) are part of the novel contributions of PEGASUS[21], and secondly, the internal structure of the Repacking algorithm contains many independent functions that need to be run. This secondary feature is very important, as there is already a large amount of work done in low-level GPU acceleration for FHE functions

(see "GPU Work" section), which means simply adding a low-level multiplication acceleration library is not very novel and has a predictable outcome. Instead, the focus in this paper was on accelerating this algorithm by focusing on the high-level structure. This is important for future work as seeing the resource usage, and speed that can be achieved at this level can give insight into creating larger systems that are more useful for real world applications.

## 3.1 Repacking

The repacking algorithm achieved in this work still uses many aspects of the original Microsoft SEAL and PEGASUS software. The main goal was to replace the innermost loop of the repacking algorithm with a parallel structure because the outermost loop consists simply of sequential homomorphic operations in the CKKS scheme that have been shown already to be amenable to low-level parallelization[30][35]. A very casual look at PEGASUS repacking algorithm's Baby-Step Giant-Step portion which contains the large summations shows the general structure seen in Equation. 1.

$$\hat{ct} = \sum Rotation_{Homomorphic}(\sum Encoding(Rotated\ Tile) * Rotated\ Repacking\ Key) \tag{1}$$

Looking at this, it is obvious that at a high-level, the internal, non-homomorphic summation, can be done with a high degree of parallelization. In addition to this, the Rotated Repacking Keys are generally pre-computed as this is easy to do and applies to subsequent repacking operations that will use the same repacking key. This means that the internal summation consists of a large number of simpler functions, that are non-dependant on one another. It is for this reason that the accelerating software computes these internal sums in the GPU, then passes this information back to the CPU to complete the repacking in the base PEGASUS software. This is done in 5 phases that can be seen in Figure. 4. Essentially, repacking parameters are first calculated and grabbed from SEAL/PEGASUS, the LWE matrix that is to be repacked is sent to the GPU, the GPU performs all the inner loop calculations and

sums them in 3 synchronizing-steps, and these sums are transferred back to the CPU, where the PEGASUS software finishes the repacking.

## 3.2 Software Strategy

### 3.2.1 Design Goals

As can be seen in Figure. 5, the accelerated software receives inputs from the PEGASUS, and returns results directly back to the PEGASUS system. The idea behind this being that eventually a variety of PEGASUS operations can be accelerated in a similar way, allowing them to work effectively as an optimization library to be used when the required hardware is present. The software created here has been used in Linux Ubuntu, as a base OS as well as with Windows WSL support. Only minor modifications to the base PEGASUS library were required to enable this, mainly consisting of changing parameter visibility to allow access to private data, and splitting functions up to take over some computations in the GPU. Due to the close integration of these systems, which includes Microsoft SEAL, a variety of parameters are required to be mirrored in the GPU, so that when these functions are transferred, they can retain homogeneity in their calculations.

With all these things in mind, there are many considerations to take when looking at how to approach accelerating the Repacking algorithm. The first point that comes up is that, as mentioned previously, the integration of SEAL and PEGASUS are crucial, and because of this an immediate requirement is to be able to perform functions with 64-bit math. This, along with the functional requirements of working with lattice-based math, means having methods of dealing with large bits, like unsigned 128-bit modular reduction that would come from 64-bit multiplication. Another design goal was to ensure that the created acceleration software would be easy to integrate, and not dependent heavily on any outside libraries (other than PEGASUS and SEAL of course). The reason for this being that it should be easy to be added onto and change as PEGASUS and SEAL will likely change in the fu-

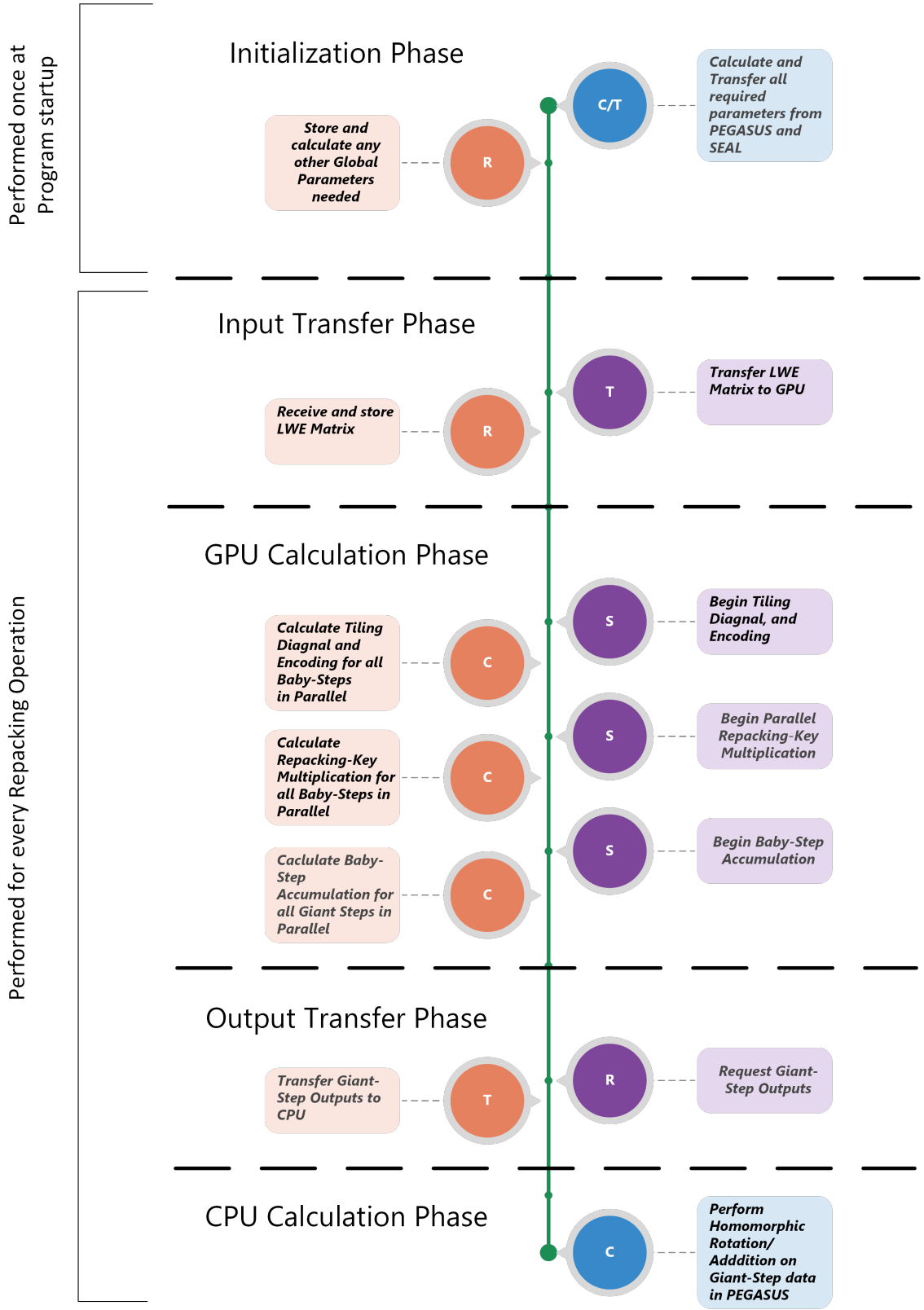



FIGURE 4: GPU and CPU Timing Diagram

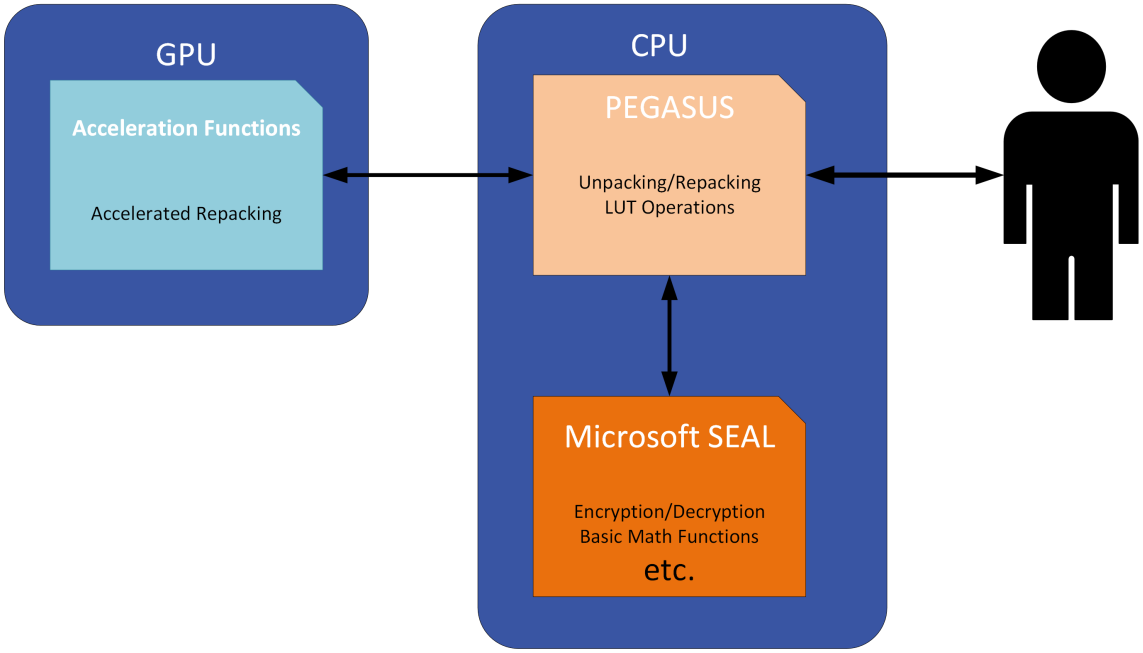



FIGURE 5: Basic overview of Hardware, and Interaction Architecture

ture. Lastly, a focus will be placed on performing high-level functions in parallel, as opposed to optimizing low-level functions. The reason behind this is that there is already a large amount of research in optimizing many of the low-level functions for FHE schemes[31][26]. Providing an implementation of high-level of parallelization can show serious improvements, that can then be further improved in the future with low-level functions that can be used ubiquitously.

### 3.2.2 Data Transfers

To use the GPU unit with CUDA, the data to be processed, along with any subsidiary information like parameters, ect., must be first passed to the GPU, and subsequently the final result must be passed back to the CPU from the GPU. This means at minimum two transfers will be required to complete an accelerated function. The issue that arises is that data transfers between CPU↔GPU are generally slow, and it must be minimized where at all possible. With this in mind, as well as the knowledge that variables in PEGASUS can be large (in the territory of Megabytes per ciphertext), it is crucial to be able to minimize the number of transfers that are required during

runtime. With this in mind, we can parse out all the data that needs to be sent between them into 3 separate stages listed below, essentially adding an additional transfer that is done at setup, so that less data is transferred per operation:

**1. Preparation Phase** – all data transfers and calculations done before program is ready for conversions. This data is non-specific to a particular data repacking and is used among all of them as long as the same Encryption Scheme/Key are being used. This phase is the least important as it is a function of the setup and doesn't impact individual repacking operations.

**2. Input Phase** – Data that must be transferred in order to initiate a single repacking operation. This is the specific LWE Ciphertexts that we wish to repack. This data consists simply of the LWE Ciphertexts formatted as a Matrix.

**3. Collection Phase** – Data that must be transferred in order to grab completed data from a repacking operation. This is the output of the parallelized parts of the repacking algorithm that can then be used to form the repacked RLWE ciphertext. This data consists of several RLWE ciphertexts, which are grabbed individually and re-formatted into the PEGASUS/SEAL framework.

### 3.2.3 GPU Processing Organization

One of the key decisions to make while creating GPU software is in the thread/block dimensionality used at runtime. These decisions must be done while creating the application's logic and are important factors in the final implementation's runtime. In the case of the repacking algorithm, there is a change of paradigm partway through as the function moves from the Matrix of LWE (FHEW) ciphers, to the RLWE Plaintext (CKKS scheme). During this, the dimensions of the data that is being processed is essentially shift from the lower lattice size to the higher lattice size. This is the reasoning for using multiple kernel calls, allowing the dimensions of GPU resources to be changed during runtime. This turned out to be the simplest way to deal with

the parameter change, with the additional benefit of causing all blocks to synchronize. The other important parameters that play into determining the block sizing is the number of inputs. This parameter will affect the input matrix size, which determines the number of Giant-Steps (g), and Baby-Steps (h) required in the algorithm. By using block sizes that match these parameters we can make the logic simpler for parallelization of the algorithm. A general description of a Kernel, with its thread and block structure can be seen previously in Figure. 3, with just the block dimensions changing with the different kernels.

# CHAPTER 4

# *Implementation*

## 4.1 Hardware

All metrics reported were performed using Ubuntu 20.04, with an Intel(R) Xeon(R) CPU @ 2.20GHz 8 Core processor and a Tesla V100-SXM2 16GB Graphics Card. Additional testing and builds were done using Windows WSL with the same version of Ubuntu, along with an Intel(R) Core(TM) i7-9700K CPU @ 3.60GHz 8 Core Processor with a Nvidia GeForce RTX 3080 10GB Graphics Card.

## 4.2 Software

### 4.2.1 Setup and Integration

The software that makes up the acceleration software can essentially be described as a C++ interface to the Cuda GPU instructions. This worked very well with the foundational software (PEGASUS and SEAL) as both are written mainly in C++. To match the PEGASUS build process, the CMAKE program was used to give the compiler instructions on which files makeup the library. This allowed for easy integration and building of the PEGASUS library, as well as its third-party software, alongside the acceleration work being done here. Dependencies were avoided as much as possible, especially third-party software that would be difficult to integrate. The exception to that was code that was taken from another public repository. Some of the low-level math functions which are described later come from such a repository, which helped eliminate both re-doing of already done work, as well as keeping dependencies

to a minimum as the code used does not depend on anything other than standard Cuda library functions. Other than this additional code, and obviously the PEGASUS library itself, no additional software is required aside from standard Cuda libraries.

### 4.2.2 Repacking Program Flow

As mentioned previously, the software work can be separated into the C++ interface that runs on the processor, and the CUDA program that runs on the GPU. The interface has several priorities but runs on the assumption that PEGASUS has already been setup as it grabs parameters from the PEGASUS runtime variable. As can be seen below in the CPU Repacking Program, the first step is to grab all the information that the GPU will need in order to run, the majority of which comes from PEGASUS with a small amount being grabbed directly from Microsoft SEAL. With this done, there are some additional parameters that were either easier to, or had to be calculated manually, so these variables are also all calculated. This mainly includes the parameters for the Barrett reduction, and the Bit-Reversed NTT omega values. Now that all the information has been gathered, it needs to be initialized into the GPU, which involves allocating memory for the variables, then transferring them over. They are first organized into 2 main categories: “Repacking Parameters” and “Encoding Parameters”. The main distinction between these two is “Encoding Parameters” deals with mainly the low-level math like parameters needed for the IFFT, and NTT, whereas “Repacking Parameters” deals more with the Encryption-Scheme level parameters. Lastly, with all the parameters needed to run the algorithm, storage space has to be created for the large variables that are used during computation. This is again done in two levels, with “Repacking Workspaces” dealing with the large CKKS plaintext/ciphertext variables, and “Encoding Workspaces” being used to store the smaller but more numerous FHEW and Encoding parameters. Additionally, space must be created in which the LWE ciphertext matrix can be transferred into, so that this won’t have to be done before each repacking operation. With all this completed, and all data transferred over and workspaces allocated the gpu can begin computations, and concludes the setup for acceleration software, which is lines 1-6

of the CPU Repacking Program. The last portion of the program that is run in the CPU is where we have received the GPU's computations of the inner repacking loop, and now pass this information back into PEGASUS in order to finish the repacking operation.

**Algorithm 6** CPU Repacking Program

**Input: PEGASUS Runtime, LWE Matrix**

**Output: Repacked RLWE Ciphertext**

1: Grab all required runtime parameters from PEGASUS and SEAL
   - Omega Roots and Modulus per CKKS level
   - CKKS prime modulus
   - LWE and RLWE lattice dimensions
   - Repacking Key
2: Calculate any additional runtime parameters in CPU
   - LWE Rotational Grouping
   - IFFT Complex Root values
   - Barret Parameters
   - Bit-Reversed NTT omega values per moduli
3: Initialize Encoding Parameters in GPU
4: Initialize Encoding Workspaces [Giant-Steps * Baby-Steps]
5: Initialize Repacking Parameters in GPU
6: Initialize Repacking Workspaces [Giant-Steps * Baby-Steps]
7: Perform GPU Repacking on LWE Matrix
8: Perform Giant-Step math in PEGASUS
   - Homomorphically shift each Giant-Step
   - Accumulate Shifted data

When all the setup has been completed for the software, the GPU kernels can be run. The transfer of the LWE matrix into the GPU, and the transfer of the accumulated plaintexts back out to the CPU are included in this process as they are done for every repacking operation that is calculated. The actual program is run

in 3 successive kernels, with error checking and synchronization being performed in between each. This is done mainly because it made creating Block-Logic simpler by matching the growing size of data dimensions as the algorithm progresses, and to allow for a way of synchronizing between blocks.

**Algorithm 7** GPU Repacking Program

**Input: LWE Matrix**

**Output: Vector of RLWE Plaintexts**

1: Transfer LWE Matrix CPU → GPU

2: Perform Pre-Encode in GPU [(Giant-Steps, Baby-Steps) blocks / (64,4) Threads]
   - Grab Tiling Diagonal
   - Perform IFFT
   - Round Data

3: Perform Encode + Repacking Key Multiplication in GPU [(Giant-Steps, Baby-Steps, number CKKS moduli) blocks / (64,4) Threads]
   - Perform NTT per modulus
   - Multiply each with Repacking Key

4: Perform Giant-Step Accumulation in GPU [(Giant-Steps, 2, number CKKS moduli) blocks / (64,4) Threads]
   - Accumulate the Baby-Step Plaintexts per Giant-step into a single output per Giant-Step

5: Transfer Accumulated Giant-Step Data GPU → CPU

## 4.3 GPU Block and Thread Organization

### 4.3.1 Thread Dimensions

The thread dimensions were the simplest in terms of planning. At the low thread level, a base (64,4) thread dimension was used. This allows for simple x,y coordinates for use in some of the Cooley-Tukey FFT-based algorithms (IFFT, and NTT), but is simple enough to flatten out into a linear 256 thread block to use in other algorithms.

These thread dimensions are used throughout all the changing block sets. For the testing hardware, larger thread blocks may have been possible, but due to the large amount of per-thread resources required it this level of threads per block was found to be the largest acceptable.

### 4.3.2 Pre-Encoding Block Dimensions

During the portion of the algorithm when the operations are being performed at the LWE Matrix level, the dimensions are mainly compromised of the number of inputs (slots), the LWE lattice size (LVL1 lattice dimension). Knowing this, the blocks used are simply the Giant-Steps as the X, and Baby-Steps as the Y, with Z being kept at 1. This allows us to perform each Pre-Encoding operation in a single block.

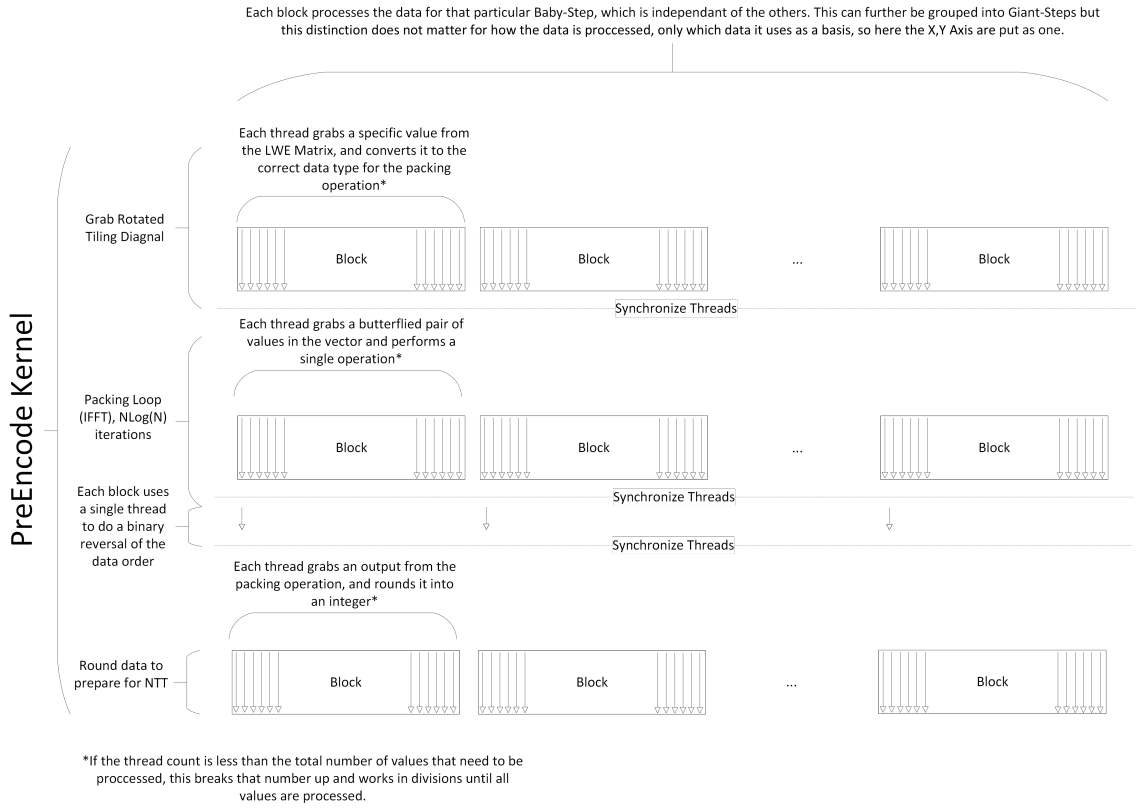



FIGURE 6: PreEncode Kernel Diagram

### 4.3.3 Encoding and Repacking Key Multiplication Block Dimensions

In this part of the algorithm, calculations are performed at the RLWE level. Here the dimensions are determined by the CKKS lattice size, and number of moduli, as well as the Giant-Step, and Baby-Steps. Knowing this, the third Z dimension is added to take care of the multiple moduli in the calculation, this allows us to assign blocks for each Baby-Step in the Giant-Steps (X, Y), as well as a separate block for processing each modulus (Z).

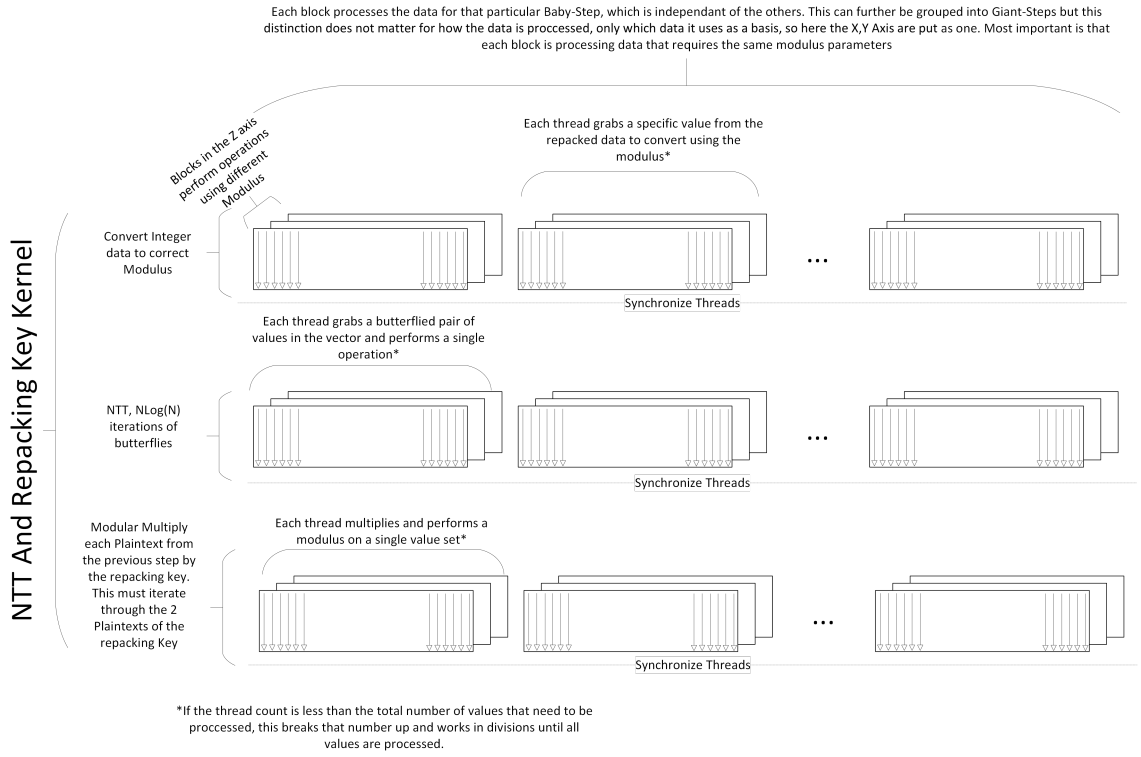



FIGURE 7: Encoding and Repacking Key Multiplication Diagram

### 4.3.4 Giant-Step Accumulation Block Dimensions

The last dimensional assignment comes from the Giant-Step Accumulation functions. Here, the Baby-Step data is now being accumulated into a Giant-Step input, which involves summing all the included Baby-Steps. Because of this, we change the (Y)

dimension of the previous step to now match the Ciphertext Size of 2 to keep the logic simple. Now the dimensions match X as Giant-Steps, Y as CKKS ciphertext Index, and Z as CKKS Modulus for block calculations.

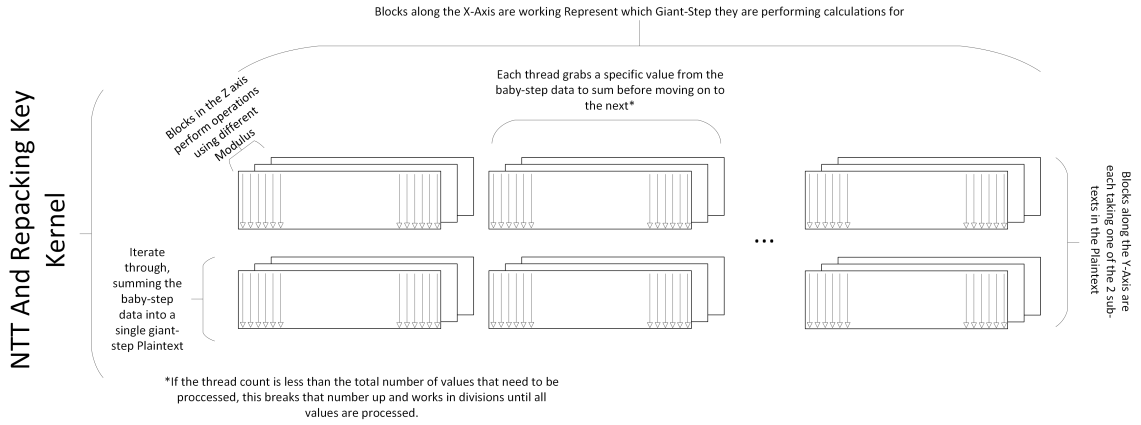



FIGURE 8: Giant-Step Accumulation Block Dimensions

## 4.4 Mathematical Calculations

### 4.4.1 Data Storage

The storage of Parameters and Workspaces for the created program was simple, because of the limitations. The CKKS encryption scheme, that is the larger parameter set used in the repacking algorithm, ends up requiring a lot of memory for typical parameters. An example would be when using $2^{16}$ (65536) means that a single ciphertext requires $2^{16}$ * 2 * 64-bit which equates to approximately 8MB of data. Considering CUDA Constant Memory must be kept under 64KB (value found from doing a "deviceQuery" on the GPU), it is easy to see how most optimized storage options are not going to be available when working in these large data sets. The acceleration software can still make use of some of this constant memory in order to store pointers, and some very basic values like lengths and sizes, which will make program logic simpler, and access to these often-used values quicker. Aside from this though, the large workspaces needed to perform the repacking and encoding must be done in the Global memory, for space requirements and the fact that they must be

Read/Write.

### 4.4.2 Data Types

One of the complications that arose in trying to create a CUDA function that could complete part of the repacking algorithm was simply the large data types required. The math mainly in the NTT, as well as some of the other ciphertext functions required going all the way up to 192-bit precision at times. Generally, we were able to do most math in 64-bit precision, with allowances to 128-bit where required before simplifying back down to 64-bit with the modulus. An understanding of the different functions, and how large they can get was key in order to avoid any possible bit loss when simplifying back down using the modulus. It was necessary to go to these large data sizes in order to keep compliance with Microsoft SEAL's scheme, which uses Uint64 as its basic data type for ciphertext and plaintexts[29].

The most basic solution to this problem would be to use an unsigned integer 128-bit precision library for this math. Ideally, this would have been an Nvidia Cuda standard library, but there were versioning issues in using this library with some of the other things used in the project. Because of this, a separate library, that focuses on 64-bit-NTT speedup was used in part and modified for much of the basic 128-bit unsigned integer math[3]. This gave efficient, low-level solutions to things like Uint64 multiplication, and addition, in Cuda Assembly. Using these solutions, along with some added helper functions allow us to achieve all the required large-bit operations described below.

Because the moduli used have at most 63-bit precision:

**Ciphertext Addition**

*Result:* at most 64-bit

*Solution:* 64-bit modulus operation

**Ciphertext Multiplication**

*Result:* At most 126-bit

*Solution:* 128-bit Barret Reduction

**Ciphertext Multiplication with 128-bit**

*Result:* at most 192-bits

*Solution:* Used only in a specific shift, store as 128-bit + 64-bit carry and specialized function used to extract correct shifted bits

### 4.4.3 Barrett Reduction

For reducing 64-bit modular multiplication the Barret Reduction method was used. The first step to performing this operation requires deriving r, and k2 from the Modulus that is desired. Because these values are fixed by modulus though, they can be pre-computed before the main repacking portion is run and are a part of the program's setup after all the required moduli are decided. The following runtime portion of the algorithm was derived from other work, easily found online, and runs in a single thread.

**Multiply (a, b)** $\rightarrow$ multiplies 2 64-bit values and returns an 128-bit value.

**MultiplyWithCarry (a, b)** $\rightarrow$ multiplies an 128-bit and 64-bit value and returns a 64-bit value and 128-bit value. The 64-bit value are the upper 64-bits (carry), and the 128-bit value are the lower 128-bits.

**Subtract (a, b)** $\rightarrow$ returns a – b of same type as a and b.

**Shift (a, b, c)** $\rightarrow$ returns the 192-bit value represented by a and b shifted right by c bits. The output of this will be $<$ 64-bit because of the modulus and k2 parameters.

**Assumptions:** $0 \leq$ input value $< modulus^2$

We can assume that all values that this algorithm are used on are in this range because firstly they are unsigned integers and thus unable to be less than 0, and secondly modular reduction is performed immediately after any calculation has the potential to be greater than the modulus.

**Algorithm 8** Barret Reduction in Acceleration Software

**Input: input x 128-bit, modulus n 64-bit**

**Output: x mod n**

1: Establish Temporary Variables tmp64[Uint64-bit] and tmp128[Uint128-bit]
2: MultiplyWithCarry (input, r) → tmp64 (high bits), tmp128(low bits)
3: shift (tmp64, tmp128, k2) → tmp64
4: Multiply (tmp64, Modulus) → tmp128
5: Subtract (Input, tmp128) → tmp128
6: **if** tmp128 < modulus **then**
7: Return (lower 64-bits of tmp128)
8: **else**
9: Subtract (tmp128, modulus)
10: Return (lower 64-bits of tmp128)
11: **end if**

### 4.4.4 Encoding

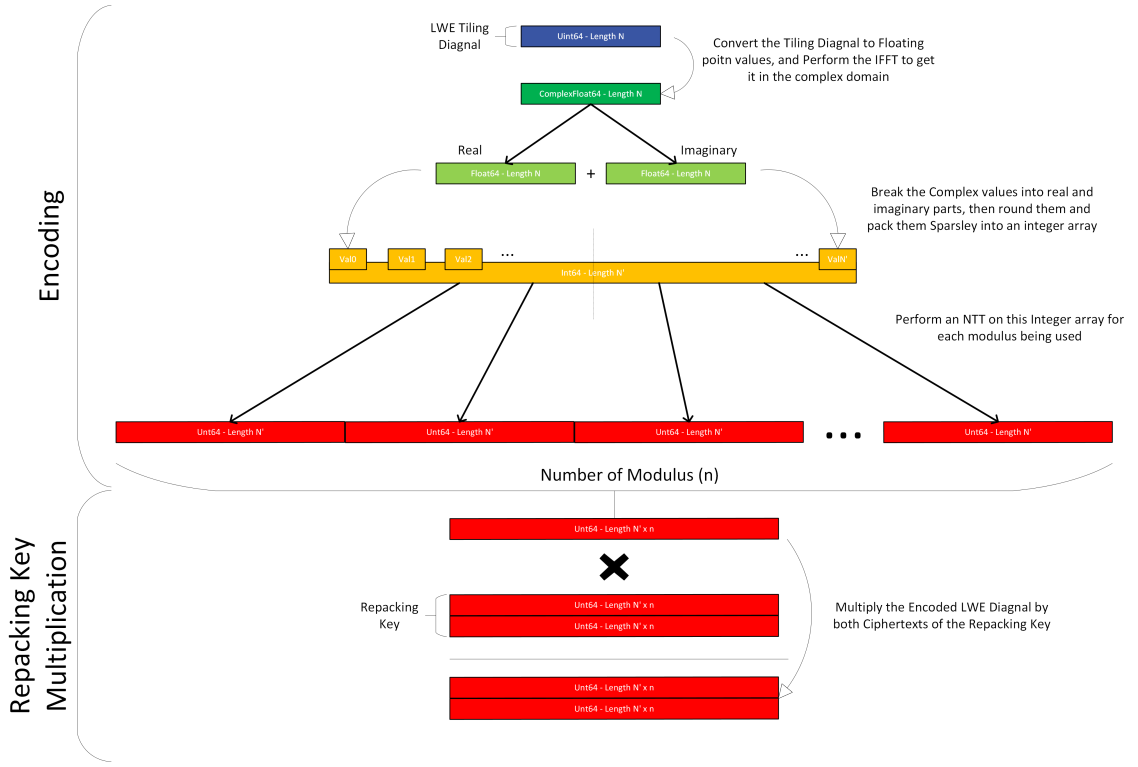



FIGURE 9: Data Dimensions for encoding plaintexts

The encoding operation can be defined as input: Vec(Float64) → output: Vec(CKKS Plaintext). This is accomplished in essentially 3 steps. First an Inverse Fast Fourier Transform (IFFT) is performed on the vector. Next, the values are rounded, and sparsely packed into an Integer vector of the same size as a CKKS Plaintext. Lastly, the values are moved into a vector of CKKS ciphertexts, one for each residue modulus that makes up the CKKS modulus being used, and an NTT is performed on each with their respective moduli. This Multiple-Moduli NTT essentially requires performing an NTT on the same data, but with each different modulus based on the Encryption Scheme parameters of the CKKS ciphertext. This allows for a large degree of parallelization, as not only can the NTT itself be run in parallel, but each modulus can also be calculated in parallel. It is at this point in the calculation that we essentially go from having been working with X-number of LWE samples, to M-modulus CKKS Plaintexts.

### 4.4.5 Parallelization of NTT and FFT

During the encoding process, both an FFT and NTT are encountered at different sections. The similarity of these two functions though allows for a large amount of commonality between how both algorithms are implemented and can thus be discussed at the same time when regarding parallelization. At a high-level, both use a radix-2 Cooley-Tukey butterfly approach to computation, allowing the respective algorithms to be broken down into log(N) computations of size N/2 of the same algorithm, as opposed to the naïve algorithm using $N^2$ computations of size N. This can be described at a high level like in Algorithm. 9, which is a simpler version of the previous description in Algorithm. 3.[20][8]

**Algorithm 9** General Radix-2 FFT Program Flow

**Input: Vector X of length N, Pre-computed table Z**

**Output: Vector X of length N**

1: $t = N$
2: **for** $m = 1 \rightarrow m < N \rightarrow m = 2m$ **do**
3: $\quad t = t/2$
4: $\quad$ **for** $iter = 0 \rightarrow iter < m$ **do**
5: $\quad\quad x_0 = 2 * iter * t$
6: $\quad\quad x_1 = x_0 + (t - 1)$
7: $\quad\quad z = Z[m + iter]$
8: $\quad\quad$ **for** $j = 0 \rightarrow j < x_1 \rightarrow j++, x_0++, x_1++$ **do**
9: $\quad\quad\quad X[x_0], X[x_1] = Func(X[x_0], X[x_1], z)$
10: $\quad\quad$ **end for**
11: $\quad$ **end for**
12: **end for**

Looking at this it is easy to see 2 things, first that each iteration of the loop requires N/2 calculations, which essentially are pairing different values in the array and performing the butterfly operation on them. Secondly, each of these butterflies are independent, meaning a full loop of calculations can be run in parallel in a collision-

free way. This also means that if it was attempted to perform the algorithm at a level of using multiple blocks per iteration, some form of inter-block synchronization would be required in order to avoid a calculation collision on an array value. Due to this, parallelization is done only per-iteration of the smaller NTT or FFT that the larger algorithm is broken into. Doing it in this way allows the program to keep all the processing in a single block, and not require inter-block synchronization, as each iteration of the smaller NTT/FFT is collision free, so synchronizations are only required before starting each.

The thread dimensions were matched to what is used for the full program flow, basically (x, 4) (the x is not important other than that it divides the dimension N of the input array). This works for the high-level parallelization because it is running a large number of these FFT/NTT's in parallel in separate blocks, so even though the smaller thread counts of a single block requires iteration through the arrays generally, a speedup is still seen across the function as a whole. This is compounded when running the NTTs as PEGASUS uses the RNS system, which means multiple NTTs have to be run in parallel for each sub-modulus that makes up full modulus. The general outline for how this is performed can be seen in Algorithm. 10, and started by using the code that was available from PEGASUS [5] and SEAL [29]. This algorithm is broken up into sections in Algorithms. 11,12, 13, and 4 in order to make pairing off the values for butterflying simpler in the code. The parallelization technique uses a similar, but simpler method outlined in other works done on NTT parallelization in GPU[26, 25].

The other issue that comes from running specifically the IFFT version of this, is that at the end of the function the results are in bit-reversed order. This is however easily reversed using a single thread, as it is not a collision-free function that can be performed by multiple threads easily. Here the approach PEGASUS uses is copied for simplicity since this is a small computation cost, and completes the bit-reversed ordering in-place, with no further memory required for storing.

**Algorithm 10** Parallelized Radix-2 FFT Program Flow

**Input: X[N]**, Pre-computed table **Z[N+1]**

**Output: X[N]**

1: **function** BUTTERFLY(a, b, c)

2: Modifies a, and b in place depending on the type (FFT/NTT) and direction (Forward/Reverse)

3: **end function**

4: $valuesperthread = (N/2)/numberofthreads$

5: Main Loop

6: Middle Loop

7: Final Loop

Note: Depending on the type, a bit-reversal of the array may be required here

**Algorithm 11** Parallelized Radix-2 FFT Program Flow - Main Loop

**Used in Parallelized Radix-2 FFT Program Flow**

1: $w = 0$

2: **for** $h = N/2; h > 2; m <<= 1, h >>= 1$ **do**

3: $x_0 = 0$

4: $x_1 = x_0 + h$

5: **for** $divs = 0; divs < valuesperthread; divs++$ **do**

6: $division_{off} = divs * blockDim_x * blockDim_y$

7: $x_{off} = (threadIdx_x * blockDim_y) + threadIdx_y + division_{off}$

8: $z_{off} = x_{off}/h$

9: $h_{off} = z_{off} * h$

10: $Butterfly(X[x_0 + x_{off} + h_{off}], X[x_1 + x_{off} + h_{off}], Z[z + z_{off}])$

11: **end for**

12: Synchronize Threads

13: $w = w + m - 1$

14: **end for**

**Algorithm 12** Parallelized Radix-2 FFT Program Flow - Middle Loop
**Used in Parallelized Radix-2 FFT Program Flow**

1: $x_0 = 0$
2: $x_1 = x_0 + 2$
3: **for** $divs = 0; divs < valuesperthread; divs++$ **do**
4: $\quad division_{off} = divs * blockDim_x * blockDim_y * 2$
5: $\quad$ **if** $threadIdx_y < 2$ **then**
6: $\quad\quad x_{off} = (threadIdx_x * blockDim_y * 2) + threadIdx_y + division_{off}$
7: $\quad\quad z_{off} = 2 * threadIdx_x$
8: $\quad$ **else**
9: $\quad\quad x_{off} = (threadIdx_x * blockDim_y * 2) + (threadIdx_y mod 2) + division_{off} + blockDim_y$
10: $\quad\quad z_{off} = (2 * threadIdx_x) + 1$
11: $\quad$ **end if**
12: $\quad Butterfly(X[x_0 + x_{off}], X[x_1 + x_{off}], Z[z + z_{off}])$
13: $\quad z = z + (blockDim_x * 2)$
14: **end for**
15: Synchronize Threads

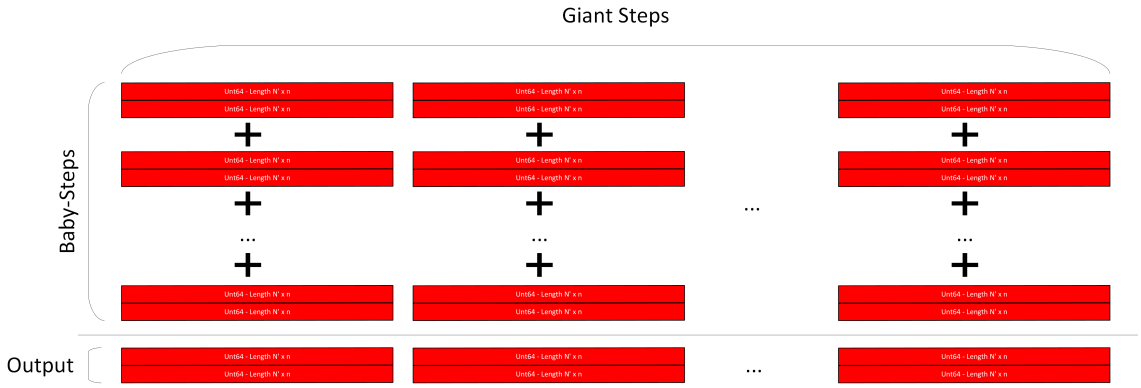
**Algorithm 13** Parallelized Radix-2 FFT Program Flow - Final Loop
**Used in Parallelized Radix-2 FFT Program Flow**

1: $x_0 = 0$
2: $x_1 = x_0 + 1$
3: **for** $divs = 0; divs < valuesperthread; divs++$ **do**
4: $\quad division_{off} = divs * blockDim_x * blockDim_y * 2$
5: $\quad x_{off} = (threadIdx_x * blockDim_y * 2) + (threadIdx_y * 2) + division_{off}$
6: $\quad z_{off} = (threadIdx_x * blockDim_y) + threadIdx_y$
7: $\quad Butterfly(X[x_0 + x_{off}], X[x_1 + x_{off}], Z[z + z_{off}])$
8: $\quad z = z + (blockDim_x * blockDim_y)$
9: **end for**
10: Synchronize Threads

### 4.4.6 Repacking Key Multiplication and Baby-Step Accumulation

In the original PEGASUS Software, a Montgomery Multiply-Accumulate is performed during the step where the repacking key is multiplied by the Encoded LWE Plaintext. This makes sense in this context, as these values need to be summed eventually so this is performing two steps in one function. However, for this work, to increase the ability to allow more threads to run in parallel, this step is run in two separate steps. This allows us to throw a lot of resources at the Multiplication step, and the Addition step, without causing any data collisions, which can be seen in Figure. 10.

FIGURE 10: Repacking Key Multiplication




Also, because PEGASUS originally used a Montgomery Multiply Accumulate, they did some pre-calculations on the Rotated Repacking keys. Since the accelerated software does these steps separate and uses a Barrett reduction here there is no need to modify the rotated repacking key, though we also use a stored version of it rather than calculating the rotations on the fly.

# CHAPTER 5

# *Results*

The results of the Accelerated function compared to running PEGASUS repacking on only the CPU can be seen in Table. 1, where at lower lattice dimensions there can be as much as 25% speedup of the overall repacking. This also equates to a 24x speedup of the part of the repacking algorithm that is currently being done in the GPU. Part of this time comprises the time it takes to transfer the data back from the GPU to the CPU, which in its current form is inefficient. If the full repacking algorithm can be done solely in the GPU, this time will no longer be wasted as that large data set instead is used in the next calculation, and a much smaller amount of data needs to be transferred back to the CPU at the end. When this computation is looked at in more detail, it can be seen that the GPU Data transferring aspect of the runtime accounts for over half of the GPU's total runtime. This means when looking purely at the GPU's calculations, and ignoring the data transfers, a 50x speedup over the regular CPU computation is achieved. In previous works, there has been a large variation in reported speedup over CPU computation when GPU acceleration has been added due to the wide variety in targeted functions, and hardware comparisons. For instance, some have used multiple GPUS and achieve large boosts of 287.2× speedup over particular CKKS functions over CPU only[30]. Others report more modest numbers like in a work that focuses on TFHE (similar to FHEW) boolean and multiplication operations, which achieves a 20x speedup over CPU only work[24]. Although there is a lot of research being done generally in the FHE acceleration space, this work sets itself apart in focusing solely on the conversion process, and achieves speedups that are in line with other similar reported numbers.

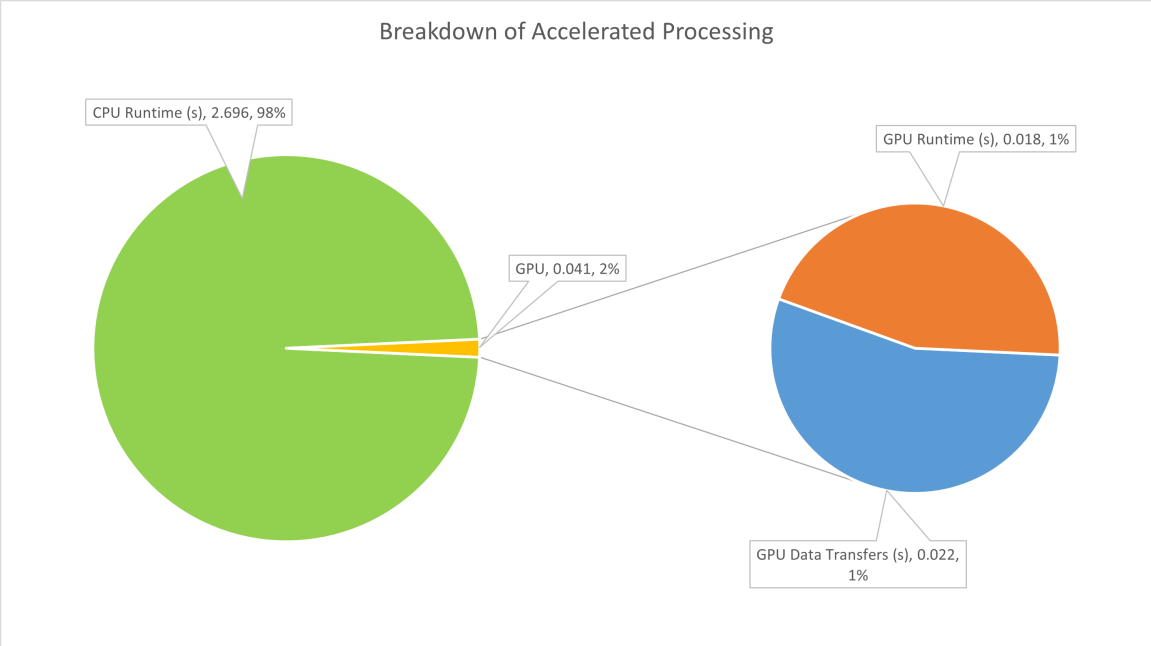



FIGURE 12: Breakdown of Accelerated Repacking

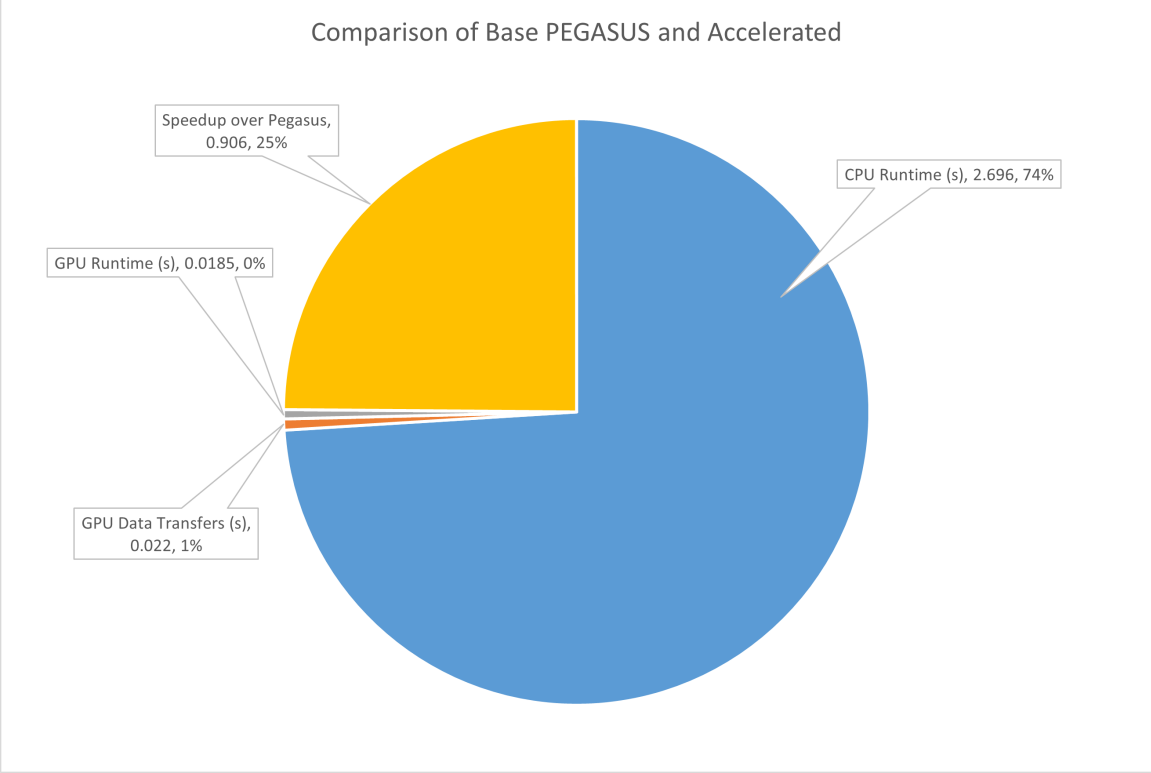



FIGURE 11: Full breakdown of base PEGASUS vs Accelerated Repacking

Another important finding in the results was that the speedup seen was diminished when the dimensions of the larger (CKKS) lattice dimension increased. This likely has to do with the fact that at the larger dimensions you begin to lose "parallelization", as the number of calculations begins to grow larger than the cores being used to calculate. This can be mitigated by changes to the scheme that would involve more cores, or through the use of larger, or multiple GPU's for increased processing power. It would also suggest a lower limit, where at the lowest calculation levels full parallelization is achieved, and below it no further speedups (as a ratio) would occur. Along the same topic, it can be seen that the smaller lattice dimension (TFHE) has little impact on the speedup, and overall runtime. This is likely due to the fact that the lower lattice bounds, as they relate to the larger one, keep it from effecting the parallelization to the same degree that the larger bounds do.

TABLE 1: Performance Improvements in PEGASUS using GPU-accelerated Repacking with Varying CKKS dimensions

| **RLWE(CKKS) Dimension ($2^x$)** | **Speedup Total** (%) | **Speedup (X)** | ***Speedup (X)** |
|---|---|---|---|
| 12 | 24.87% | 23.18 | 51.26 |
| 14 | 21.95% | 17.77 | 39.83 |
| 16 | 21.93% | 16.95 | 38.00 |

TABLE 2: Performance Improvements in PEGASUS using GPU-accelerated Repacking with Varying FHEW dimensions

| **LWE(FHEW) Dimension ($2^x$)** | **Speedup Total (%)** | **Speedup (X)** |
|---|---|---|
| 10 | 21.33% | 16.20606824 |
| 11 | 21.13% | 16.17386519 |
| 12 | 20.68% | 15.554524 |

The Accelerated repacking was compared 1:1 with the base PEGASUS's repacking to ensure that the results were real, this found an extremely small amount of error, that has been hard to pin down. It is possible it comes from a slight difference in the way Floating point math happens early in the encoding process, but it has not been completely determined and solved as of now. That said, as previously stated it is a very small amount, and should not affect normal results of operating with PEGASUS.

TABLE 3: Output Differences between Accelerated and Base PEGASUS

| **Run Number** | **Average Difference** | **Max Difference** |
|---|---|---|
| 1 | 6.45361E-09 | 1.07193E-07 |
| 2 | 5.93735E-09 | 9.82973E-08 |
| 3 | 6.76987E-09 | 9.04605E-08 |
| 4 | 5.1547E-09 | 8.56686E-08 |
| 5 | 6.38357E-09 | 8.04439E-08 |
| 6 | 5.99021E-09 | 9.76073E-08 |
| 7 | 5.85862E-09 | 1.05462E-07 |
| 8 | 5.33757E-09 | 1.16918E-07 |
| 9 | 5.73707E-09 | 9.92051E-08 |

All the results come from running the Accelerated and Base PEGASUS code on the same hardware, Ubuntu 20.04, with an Intel(R) Xeon(R) CPU @ 2.20GHz 8 Core processor and a Tesla V100-SXM2 16GB Graphics Card, obviously only the Accelerated code makes use of the GPU. Additional parameters not mentioned that were used to initialize PEGASUS software are as follows: *CKKS Levels:* 4, *CKKS Scale:* $2^{40}$, *Inputs to Repack:* 512 *PEGASUS CPU Threads:* 4

TABLE 4: Full and Amortized Runtimes for CKKS length $2^{12}$

| CKKS $2^{12}$ | Full (s) | Amortized (s) |
|---|---|---|
| PEGASUS Runtime | 3.643213333 | 0.007115651 |
| Accelerated Runtime | 2.737066 | 0.005345832 |
| Difference | 0.906147333 | 0.001769819 |

TABLE 5: Full and Amortized Runtimes for CKKS length $2^{14}$

| CKKS $2^{14}$ | Full (s) | Amortized (s) |
|---|---|---|
| PEGASUS Runtime | 11.52451467 | 0.022508818 |
| Accelerated Runtime | 8.995158 | 0.017568668 |
| Difference | 2.529356667 | 0.00494015 |

TABLE 6: Full and Amortized Runtimes for CKKS length $2^{16}$

| CKKS $2^{16}$ | Full (s) | Amortized (s) |
|---|---|---|
| PEGASUS Runtime | 45.60303733 | 0.089068432 |
| Accelerated Runtime | 35.60146567 | 0.069534113 |
| Difference | 10.00157167 | 0.01953432 |

# CHAPTER 6

# *Conclusion*

## 6.1 Summary

The work done here has begun the process of accelerating the functions of the PEGASUS framework. This has been achieved through starting with a function that has so far escaped the attention of other GPU work in this field and contained a high degree of parallelization at a functional level. The speedup to the full functionality could be significant and could be useful to other works that make use of a similar FHE conversion method. Additionally, the integration with Microsoft SEAL and PEGASUS makes it an attractive springboard for further acceleration projects. FHE conversion schemes have great potential in creating Fully Homomorphic machine learning functions[21]. This has the potential to revolutionize different consumer services like healthcare[33], and can add consumer privacy and protection to almost any software service that relies on a machine-learning algorithm.

## 6.2 Possible Future Works

The obvious work to be done would be to complete acceleration functions for the full PEGASUS library. This would be useful but may be better done in conjunction with work already done in this line by combining with a previously created FHE GPU acceleration library, like the CARM library [30] on the CKKS side of processing. At a lower level, using something like Cuda Cooperative groups would allow more resources to be used during some of the computations [2]. More work has to be done

on the standardization of the FHE schemes in order to determine best what can be added to the work done on in this paper.

# REFERENCES


[1] Nov. 2023. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[2] June 2023. URL: https://developer.nvidia.com/blog/cooperative-groups/.

[3] URL: https://github.com/CHOSEN24/NTTFusion/releases.

[4] Abbas Acar et al. "A survey on homomorphic encryption schemes". In: *ACM Computing Surveys* 51.4 (2018), pp. 1–35. DOI: 10.1145/3214303.

[5] Alibaba-Gemini-Lab. *Alibaba-Gemini-Lab/openpegasus*. URL: https://github.com/Alibaba-Gemini-Lab/OpenPEGASUS.

[6] Paul Barrett. "Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor". In: *Advances in Cryptology — CRYPTO' 86*. Ed. by Andrew M. Odlyzko. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, pp. 311–323. ISBN: 978-3-540-47721-1.

[7] Christina Boura et al. "Chimera: Combining ring-LWE-based fully homomorphic encryption schemes". In: *Journal of Mathematical Cryptology* 14.1 (2020), pp. 316–338. DOI: 10.1515/jmc-2019-0026.

[8] Hao Chen, Kim Laine, and Rachel Player. "Simple encrypted arithmetic library - SEAL v2.1". In: *Financial Cryptography and Data Security* (2017), pp. 3–18. DOI: 10.1007/978-3-319-70278-0_1.

[9] Steph Cheng and Jeriah Yu. *Parallelization of the Number Theoretic Transform*. Apr. 2023.

[10] Jung Hee Cheon et al. "Bootstrapping for Approximate Homomorphic Encryption". In: *Advances in Cryptology – EUROCRYPT 2018* (2018), pp. 360–384. DOI: 10.1007/978-3-319-78381-9_14.

[11] Ilaria Chillotti et al. "Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds". In: *Advances in Cryptology – ASIACRYPT 2016* (2016), pp. 3–33. DOI: 10.1007/978-3-662-53887-6_1.

[12] James W. Cooley and John W. Tukey. "An algorithm for the machine calculation of complex Fourier series". In: *Mathematics of Computation* 19.90 (1965), pp. 297–301. DOI: 10.1090/s0025-5718-1965-0178586-1.

[13] Thomas H. Cormen et al. *Introduction to algorithms*. Mit Press, 2009.

[14] Léo Ducas and Daniele Micciancio. "FHEW: Bootstrapping homomorphic encryption in less than a second". In: *Advances in Cryptology – EUROCRYPT 2015* (2015), pp. 617–640. DOI: 10.1007/978-3-662-46800-5_24.

[15] Phap Duong-Ngoc et al. "Flexible GPU-based implementation of number theoretic transform for homomorphic encryption". In: *2022 19th International SoC Design Conference (ISOCC)* (2022). DOI: 10.1109/isocc56007.2022.10031464.

[16] Craig Gentry. "Fully homomorphic encryption using ideal lattices". In: *Proceedings of the forty-first annual ACM symposium on Theory of computing* (2009). DOI: 10.1145/1536414.1536440.

[17] Kyoohyung Han, Minki Hhan, and Jung Hee Cheon. "Improved homomorphic discrete fourier transforms and FHE Bootstrapping". In: *IEEE Access* 7 (2019), pp. 57361–57370. DOI: 10.1109/access.2019.2913850.

[18] Zhichuang Liang and Yunlei Zhao. *Number Theoretic Transform and Its Applications in Lattice-based Cryptosystems: A Survey*. 2022. arXiv: 2211.13546 [cs.CR].

[19] Richard Lindner and Chris Peikert. "Better key sizes (and attacks) for LWE-based encryption". In: *Topics in Cryptology – CT-RSA 2011* (2011), pp. 319–339. DOI: 10.1007/978-3-642-19074-2_21.

[20] Patrick Longa and Michael Naehrig. "Speeding up the number theoretic transform for faster ideal lattice-based cryptography". In: *Cryptology and Network Security* (2016), pp. 124–139. DOI: 10.1007/978-3-319-48965-0_8.

[21] Wen-jie Lu et al. "Pegasus: Bridging polynomial and non-polynomial evaluations in homomorphic encryption". In: *2021 IEEE Symposium on Security and Privacy (SP)* (2021). DOI: 10.1109/sp40001.2021.00043.

[22] Chiara Marcolla et al. In: *Survey on fully homomorphic encryption, theory and applications* (2022). DOI: 10.36227/techrxiv.19315202.v2.

[23] P.V Ananda Mohan. *Residue Number Systems Theory and Applications*. Springer International Publishing, Imprint: Birkhäuser, 2016.

[24] Toufique Morshed, Md Momin Aziz, and Noman Mohammed. "CPU and GPU accelerated fully homomorphic encryption". In: *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)* (2020). DOI: 10.1109/host45689.2020.9300288.

[25] Ali Şah Özcan et al. "Homomorphic encryption on GPU". In: *IEEE Access* 11 (2023), pp. 84168–84186. DOI: 10.1109/access.2023.3265583.

[26] Özgün Özerk et al. "Efficient number theoretic transform implementation on GPU for Homomorphic encryption". In: *The Journal of Supercomputing* 78.2 (2021), pp. 2840–2872. DOI: 10.1007/s11227-021-03980-5.

[27] Oded Regev. "On lattices, learning with errors, random linear codes, and cryptography". en. In: *Journal of the ACM* 56.6 (Sept. 2009), pp. 1–40. ISSN: 0004-5411, 1557-735X. DOI: 10.1145/1568318.1568324. URL: https://dl.acm.org/doi/10.1145/1568318.1568324 (visited on 10/12/2023).

[28] Oded Regev. "The learning with errors problem (invited survey)". In: *2010 IEEE 25th Annual Conference on Computational Complexity* (2010). DOI: 10.1109/ccc.2010.26.

[29] *Microsoft SEAL (release 3.5)*. https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA. Apr. 2020.

[30] Shiyu Shen et al. "Carm: Cuda-accelerated rns multiplication in word-wise homomorphic encryption schemes for internet of things". In: *IEEE Transactions on Computers* (2022), pp. 1–12. DOI: 10.1109/tc.2022.3227874.

[31] Kaustubh Shivdikar et al. "Accelerating polynomial multiplication for homomorphic encryption on gpus". In: *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED)* (2022). DOI: 10.1109/seed55351.2022.00013.

[32] William Stallings. *Cryptography and network security: Principles and practice*. 7th ed. Pearson, 2017.

[33] Anamaria Vizitiu et al. "Applying deep neural networks over homomorphic encrypted medical data". In: *Computational and Mathematical Methods in Medicine* 2020 (2020), pp. 1–26. DOI: 10.1155/2020/3910250.

[34] Wei Wang et al. "Accelerating fully homomorphic encryption using GPU". In: *2012 IEEE Conference on High Performance Extreme Computing* (2012). DOI: 10.1109/hpec.2012.6408660.

[35] Hai-bin Yang et al. "Efficiency analysis of TFHE fully homomorphic encryption software library based on GPU". In: *Advances in Intelligent Systems and Computing* (2019), pp. 93–102. DOI: 10.1007/978-3-030-15035-8_9.

[36] Naifeng Zhang and Franz Franchetti. *Generating Number Theoretic Transforms for Multi-Word Integer Data Types*. 2023. URL: https://users.ece.cmu.edu/~franzf/papers/CGO_MPNTT_2023_Final.pdf.

# VITA AUCTORIS

Emilio Quaggiotto was born in 1996 in Windsor, Ontario. He graduated from St. Thomas of Villanova High School in 2014. From there he went on to the University of Windsor where he obtained a B.Sc. in Electrical and Computer Engineering in 2019, and a Minor in Mathematics. Following this, he went to work in the Automotive Industry as a Software Engineer. He is currently a candidate for the Master's degree in Electrical and Computer Engineering at the University of Windsor and hopes to graduate in Winter 2024.