Performance of Hierarchical Transforms in Homomorphic Encryption

A case study on Logistic Regression inference

Pedro Geraldo M. R. Alves[⋆1,2*], Jheyne N. Ortiz[1] and Diego F. Aranha[2]

[1*]Institute of Computing, University of Campinas, Campinas, Brazil.
[2]Department of Computer Science, Aarhus University, Aarhus, Denmark.

*Corresponding author(s). E-mail(s): pedro.alves@ic.unicamp.br;
Contributing authors: jheyne.ortiz@ic.unicamp.br; dfaranha@cs.au.dk;

Abstract

Recent works challenged the Number-Theoretic Transform (NTT) as the most efficient method for polynomial multiplication in GPU implementations of Fully Homomorphic Encryption schemes such as CKKS and BFV. In particular, these works argue that the Discrete Galois Transform (DGT) is a better candidate for this particular case. However, these claims were never rigorously validated, and only intuition was used to argue in favor of each transform. This work brings some light on the discussion by developing similar CUDA implementations of the CKKS cryptosystem, differing only in the underlying transform and related data structure. We ran several experiments and collected performance metrics in different contexts, ranging from the basic direct comparison between the transforms to measuring the impact of each one on the inference phase of the logistic regression algorithm. Our observations suggest that, despite some specific polynomial ring configurations, the DGT in a standalone implementation does not offer the same performance as the NTT. However, when we consider the entire cryptosystem, we noticed that the effects of the higher arithmetic density of the DGT on other parts of the implementation is substantial, implying a considerable performance improvement of up to **15%** on the homomorphic multiplication. Furthermore, this speedup is consistent when we consider a more complex application, indicating that the DGT suits better the target architecture.

Keywords: NTT, DGT, Fully Homomorphic Encryption, CKKS, CUDA, Polynomial multiplication, Privacy-preserving computing

1 Introduction

In 1978, Rivest *et al.* first conceived the notion of Homomorphic Encryption (HE) schemes [41]. Their objective was to preserve some mathematical structure after encryption that enables the evaluation of arithmetic circuits over ciphertexts without decryption or knowledge of the secret key. The computation outcome is naturally encrypted, and an observer learns nothing regarding the operands, the result, or the decryption key. The first HE schemes had limited capability, supporting only additions or multiplications,

[⋆]Results were partially obtained while visiting the Department of Computer Science at Aarhus University for 12 months.

and because of that they were called Partially Homomorphic Encryption (PHE) schemes. ElGamal's and Paillier's are notorious examples of such PHE schemes [24, 37]. Since they cannot support both operations simultaneously, their suitability to real-world implementations is limited. It took around 30 years after this initial work for the first practical construction of a Fully Homomorphic Encryption (FHE) scheme supporting an unlimited number of both operations to be introduced [26]. Unfortunately, the first proposals did not stand out for performance regarding latency or memory consumption. Still, Gentry was successful in drawing a blueprint that has guided many FHE schemes. His main contribution [26] was the proposal of a bootstrapping operation that homomorphically evaluates the decryption procedure to remove the upper bound on the complexity of supported functions. The following decade was dedicated to security and performance improvements [12–15, 22, 27, 34].

Modern schemes have significantly reduced the performance overhead imposed on computation over ciphertexts [16, 23]. Their implementation relies on polynomial arithmetic, so developers have to find efficient ways to handle known costly operators, such as polynomial multiplication and division. Concerning the former, the literature has established the suitability of the Number-Theoretic Transform (NTT), a variant of the Discrete Fourier Transform (DFT) that operates over integers, to compute the polynomial multiplication with linear complexity within the transform domain. Nonetheless, some recent works suggest that the Discrete Galois Transform (DGT) may be a better candidate when the target hardware is a CUDA-enabled GPU [2, 4, 6]. CUDA is a SIMD architecture developed and maintained by NVIDIA to employ GPUs' potential for data parallelism in tasks beyond graphical processing. In particular, the suitability of current devices for polynomial arithmetic made CUDA an essential tool for the efficient implementation of FHE schemes [21, 32].

However, the particularities of the architecture impose some challenges. For example, its processing flow demands careful planning to align possible conditional branches with certain thread groups, and its memory paradigm considers several structures with different dimensions and latency characteristics, apart from the machine's main memory. So, part of the difficulty of its use involves tailoring classical methods into variants that conveniently fit the GPU.

1.1 Related Work

Current HE schemes are built on top of Gentry's proposal of a bootstrappable cryptosystem, a scheme that homomorphically evaluates its own decryption circuit [26]. The bootstrap procedure enables these proposals to be used as fully homomorphic since they can perform an unlimited number of additions and multiplications. Some of the primary schemes available in the literature are BFV [22], CKKS [14], and TFHE [15]. All these schemes share a common core for polynomial arithmetic, which allows using a DFT-based method to accelerate polynomial multiplication. For this task, one may use the Fast Fourier Transform (FFT), the NTT, or the DGT. In particular, we target the application of these methods to the implementation of CKKS on GPUs.

The applicability of DFT-based algorithms to efficiently implement polynomial multiplication is ubiquitous in the FHE literature. In 1966, Gentleman and Sande [25] proposed the hierarchical FTT, and it remained forgotten in the interim until 1989 when Bailey rediscovered it as the "four-step" FFT algorithm [7]. Later on, in 2008, Govindaraju et al. implemented the four-step hierarchical FFT in the context of GPUs [28]. A few years later, Harvey developed arithmetic techniques for reducing the number of reductions modulo p during the computation of the NTT [30]. The NTT is a DFT variant that works in a finite field, so its arithmetic better suits cryptographic contexts, avoiding the use of floating-point arithmetic that is inherent to the FFT. Because of that, the NTT became the norm in the literature. In 2018, Dai et al. recursively applied the four-step Cooley-Tukey algorithm [17] to obtain NTTs with size 64 instead of performing the transform on integer vectors of 2048 and 4096 coordinates [20]. Recently, Jung et al. applied a hierarchical implementation of the NTT in the fastest up-to-date implementation of logistic regression over GPUs [31]. Following a slightly different branch, Badawi et al. explored the suitability of the DGT for FHE [2]. The DGT works in the finite field $\mathbb{F}_{p^2}$ for prime p. Thus, it is similar to the NTT but also adds the possibility of halving the operands' degree. In this same direction, the DGT was used to accelerate polynomial multiplication on a BFV implementation on GPUs [6], and lately, Alves et al. improved the use of the DGT for polynomial multiplication in GPUs through a hierarchical implementation [4].

The DGT is favored by the fact that arithmetic in the field $\mathbb{F}_{p^2}$ is performed in the set of Gaussian integers $\mathbb{Z}_p[i]$, for some convenient choice of prime p. By working with Gaussian integers, the operands are converted from a N-degree to a $N/2$-degree polynomial ring via a folding procedure. Consequently, the degree of the inputs used inside the transform is halved and so is the number of roots that will be loaded and stored into memory during the transform computation. Moreover, a CUDA implementation can also reduce the number of required threads, which avoids the parallel performance degradation in bigger instances [2, 6]. Nevertheless, the representation in $\mathbb{Z}_p[i]$ implies denser arithmetic operations, which saves in memory bandwidth consumption. Notice that arithmetic in $\mathbb{F}_p$ consists in coefficient-wise operations modulo p between two N-degree polynomials. On the other hand, when the base field is $\mathbb{F}_{p^2}$, addition and multiplication operations are similar to the ones over complex numbers. This means that an increased arithmetic density is natural to the DGT and may improve performance if the implementation explores the processing hardware special capabilities.

1.2 Our contributions

In this paper, we describe our efforts to resume the investigation started by Badawi et al. on the possible advantages of replacing the NTT with the DGT for the implementation of polynomial multiplication in FHE cryptosystems [2]. In particular, we target results claiming that the DGT is more suitable than NTT for GPUs and memory-bounded platforms [5]. To the best of our knowledge, no previous work provided a deep analysis of the advantages of each transform in the context of GPU execution.

We developed two implementations of the CKKS scheme using DGT and NTT as the underlying transform to perform multiplication in the ring $\mathbb{Z}_q[x]/(x^N + 1)$. The implementations are referred to as AOA-DGT and AOA-NTT, respectively. We compare the latencies of the transforms executed independently and also within CKKS' homomorphic primitives. Moreover, we present a case study verifying how the performance of the logistic regression inference is affected by each. For that, we compute the inference score using both implementations for a trained model over the MNIST database considering the case when the model is encrypted, protecting its secrecy, and when it is handled as plaintext [33]. Furthermore, we analyze how the problem scales for approaches found in the literature that run the inference with and without the computation of the activation function.

Our experiments reveal that the NTT offers a clear performance advantage over the DGT in most cases and especially in smaller instances. For 8192-degree polynomial rings, however, the DGT more efficiently takes advantage of the processing hardware and can overcome the NTT.

Nonetheless, the AOA-DGT's homomorphic multiplication performs better even though its related transform implementation does not. We found procedures in the critical path of that primitive not directly related to the transform itself but that are impacted by the associated implementation decisions. For instance, a 10% slowdown on the Logistic Regression inference executed on AOA-NTT is observed, caused by the less efficient basis extension methods. We show that this result can be reversed by increasing arithmetic density in AOA-NTT to match the one in AOA-DGT.

Organization

This document is organized as follows. Section 2 describes the adopted notation and relevant basic building blocks at Sections 2.2 and 2.4; and the target FHE scheme, at Section 2.3. Section 3 presents our experiments and methodology, and discusses the obtained results. Furthermore, Section 4 presents a case study on how the selection of each transform affects the performance on the inference of a homomorphic logistic regression implementation.

2 Background

Most implementations of cryptosystems based on the Ring Learning-With-Errors (RLWE) problem requires the construction of basic building blocks that offer polynomial arithmetic on a cyclotomic ring. An example is the cryptosystem CKKS [14], a leveled homomorphic encryption scheme. In particular, the implementation of algorithms for polynomial multiplication has been a challenging task. A simple schoolbook algorithm requires quadratic complexity on the operands degree, and because of that it is utterly unsuitable on instances of cryptographic size. The typical approach in the literature involves variants of the DFT, which offer linear complexity when the operands lie in their domains. In this work, we focus on the efficient implementation of the NTT and DGT.

In this context, this section describes the notation used throughout the document, defines the relevant primitives of CKKS, and discusses formulations for the NTT and the DGT.

2.1 Notation

We use bold letters to denote vectors e.g., $\mathbf{a}$ and $\mathbf{A}$. For a vector $\mathbf{a}$, we refer by a_i the i-th element. We denote by $[x]_q$ the reduction of an integer x modulo q, that is, $[x]_q := x \mod q$, for some integer q. Furthermore, let $\mathbf{X}$ be a matrix, then $x_{*,i}$ is the set of all the elements on column i of $\mathbf{X}$. In the same way, $x_{i,*}$ is the set of all the elements on row i of $\mathbf{X}$. Also, we use $\lfloor x \rceil$ to define the rounding to the nearest integer operation.

Let K be the $2N$-th cyclotomic number field and $R = \mathcal{O}_K$ its ring of integers. We represent R in its polynomial form, that is, $R = \mathbb{Z}[x]/(x^N + 1)$. Moreover, for an integer $q \geq 2$, R_q denotes the quotient ring $R_q = \mathbb{Z}_q[x]/(x^N + 1)$.

Consider that $\mathcal{C} = \{q_0, q_1, \ldots, q_\ell\}$ is a set of coprime integers and $q = \Pi_{i=0}^{\ell} q_i$. If $s \in R_{\mathcal{C}}$, then $\exists S \in R_q$ such that $s := \{[S]_{q_0}, [S]_{q_1}, \ldots, [S]_{q_\ell}\}$. Arithmetic operations as addition and multiplication over elements in $R_{\mathcal{C}}$ are taken coefficient-wise, that is, if $a, b \in R_{\mathcal{C}}$ then $a+b = \left\{[a_0 + b_0]_{q_0}, \ldots, [a_\ell + b_\ell]_{q_\ell}\right\}$. Furthermore, we denote by $[x]_{q_0}$ the operation that selects the 0-th residue of x.

2.2 DFT-based Transforms

NTT is a variation of the DFT that replaces the primitive N-th complex root of unity by a primitive N-th root of unity ω_N in a ring $\mathbb{Z}_p$ [40]. For N a power of two, the NTT requires p to be a prime number and that $N \mid (p-1)$. In this case, Pollard proved that there exists a primitive N-th root of unity in $\mathbb{Z}_p$ that can be computed as $r^{(p-1)/N}$, where r is the primitive root modulo p. For a polynomial $a(x) = \sum_{j=0}^{N-1} a_j x^j \in \mathbb{Z}[x]$, the N-point NTT computes

$$\mathrm{NTT}_{\omega_N}(a(x)) = \left(a(\omega_N^0), \ldots, a(\omega_N^{N-1})\right). \quad (1)$$

Conversely, the inverse transform is

$$\mathrm{INTT}_{\omega_N^{-1}}(a(x)) = \left[N^{-1} \cdot \mathrm{NTT}_{\omega_N^{-1}}(a(x))\right]_p, \quad (2)$$

where N^{-1} is the multiplicative inverse of N modulo p. The polynomial multiplication $c(x) = a(x) \cdot b(x) \mod x^N + 1$ can be done via NTT as

$$c(x) = \omega_{2N}^{-1} \cdot \mathrm{INTT}_{\omega_N^{-1}}(\mathrm{NTT}_{\omega_N} \hat{a}(x)) \odot \mathrm{NTT}_{\omega_N}(\hat{b}(x))),$$

in which $\hat{a}(x) = \omega_{2N} \cdot a(x)$. This technique of scaling the operands by powers of ω_{2N} is known as the negative wrapped convolution [35].

DGT is an alternative to the NTT over the finite field $\mathbb{F}_{p^2}$, where p is a prime. It was presented as a method for integer convolution by Crandall [18] and was further considered for polynomial multiplication by Badawi et al. [6]. The DGT and its inverse transform are defined in the same way as NTT in Equations 1 and 2, but now the operands are vectors with elements in $\mathbb{F}_{p^2}$.

In this context, the field elements may be represented using the set of Gaussian integers modulo p, denoted $\mathbb{Z}_p[i]$, which is defined as $\mathbb{Z}_p[i] = \{a + ib \mid a, b \in \mathbb{Z}_p\}$, for $i = \sqrt{-1}$. The arithmetic in $\mathbb{Z}_p[i]$ is similar to the one in $\mathbb{C}$ but both real $(\Re)$ and imaginary $(\Im)$ parts are taken modulo p. When the polynomial ring is $\mathbb{Z}[x]/(x^N + 1)$, Crandall [18] defines a transform on a vector $\mathbf{a}$ with size $N \equiv 0 \mod 2$ to a vector $\mathbf{A}$ with size $N/2$, denoted as *folding*, such that

$$A_j = a_j + i a_{j+\frac{N}{2}} \in \mathbb{Z}_p[i].$$

This transform maps the coefficients of the polynomial from $\mathbb{Z}_p^N$ to $\mathbb{Z}_p[i]^{\frac{N}{2}}$. The inverse transform from $\mathbb{Z}_p[i]^{\frac{N}{2}}$ to $\mathbb{Z}_p^N$ is denoted as *unfolding* and it is given by

$$a_j = \Re(A_j) \quad \text{and} \quad a_{j+\frac{N}{2}} = \Im(A_j),$$

for $0 \leq j \leq N/2 - 1$. Crandall [18] also defines a "right-angle" convolution that multiplies the folded vector $\mathbf{A}$ by powers of $\tau = \tau_{\frac{N}{2}}$, an $\frac{N}{2}$-th root of i modulo p. We refer to this convolution as *twisting*, since powers of τ are the twisting factors defined as

$$A_j = A_j \cdot \tau^j.$$

The corresponding inverse convolution is given by the multiplication of the vector $\mathbf{a}$, which is the output of the unfolding procedure, by growing powers of the inverse $\frac{N}{2}$-th root of i, denoted τ^{-1}.

For completeness, we present the polynomial multiplication in the ring $\mathbb{Z}_p[x]/(x^N+1)$ via DGT introduced by Badawi et al. [6] in Algorithm 1. Notice that it operates on the coefficient vectors $\mathbf{a}$ and $\mathbf{b}$ of two polynomials $a(x), b(x) \in \mathbb{Z}_p[x]/(x^N+1)$. Similarly, the algorithm outputs the coefficient vector $\mathbf{c}$ corresponding to the computation $c(x) = a(x) \cdot b(x) \in \mathbb{Z}_p[x]/(x^N+1)$.

Algorithm 1 Polynomial multiplication in $\mathbb{Z}_p[x]/(x^N+1)$ via DGT

```
Require: Vectors $\mathbf{a}, \mathbf{b} \in \mathbb{Z}_p^N$, $p$ a prime number,
    $N$ a power-of-two integer, and $\tau$ a primitive $\frac{N}{2}$-
    th root of $i$ modulo $p$.
Ensure: A coefficient vector $\mathbf{c} \in \mathbb{Z}_p^N$.
  for $j = 0;\ j < N/2;\ j = j + 1$ do
      $a'_j = a_j + i a_{j+N/2}$
      $b'_j = b_j + i b_{j+N/2}$
  end for
  for $j = 0;\ j < N/2;\ j = j + 1$ do
      $a'_j = \tau^j \cdot a'_j \pmod p$
      $b'_j = \tau^j \cdot b'_j \pmod p$
  end for
  $a' = \mathrm{DGT}(a')$
  $b' = \mathrm{DGT}(b')$
  for $j = 0;\ j < N/2;\ j = j + 1$ do
      $c'_j = a'_j \cdot b'_j \pmod p$
  end for
  $c' = \mathrm{IDGT}(c')$
  for $j = 0;\ j < N/2;\ j = j + 1$ do
      $\mathrm{aux} = \tau^{-j} \cdot c'_j \pmod p$
      $c_j = \Re(\mathrm{aux})$
      $c_{j+\frac{N}{2}} = \Im(\mathrm{aux})$
  end for
  return c
```

2.3 CKKS Scheme

Cheon-Kim-Kim-Song proposed a leveled homomorphic encryption scheme known as CKKS [14] in which the plaintext domain is composed of complex numbers. The array of complex numbers is mapped into elements of the ring R using an encoding method that works as follows.

Let $\mathbf{z}$ be a vector of N complex numbers and Δ a scalar. In practice, we have that $\mathsf{decode}(\mathbf{z}) = \lfloor \mathrm{FFT}(\mathbf{z}) \cdot \Delta^{-1} \rceil$. In this sense, encode is defined simply as the inverse procedure. Through this approach, CKKS becomes capable of operating with non-integer numbers using fixed-point arithmetic. In this representation, Δ is called the *scaling factor* and is responsible for setting its precision.

A CKKS ciphertext, denoted $\mathsf{ct} = (c_0, c_1)$, is a pair of elements in $R_{\mathcal{C}}$, for $\mathcal{C} = \{q_0, \ldots, q_L\}$ a RNS basis. In other words, $\mathsf{ct} = \{(c_0^{(i)}, c_1^{(i)})\}_{0 \leq i \leq L}$ such that $(c_0^{(i)}, c_1^{(i)}) \in R_{q_i} \times R_{q_i}$. At this moment, we say that this ciphertext has level $\ell = L + 1$. We have that c_0 and c_1 are built by the CKKS encryption algorithm such that $[c_0 + s \cdot c_1]_{q_0} \approx m$, for a secret key s.

A decryption imprecision is expected, and its error is considered part of the cryptosystem's inherent noise. Let $\mathrm{ct}_0 = (\mathrm{ct}_{0,0}, \mathrm{ct}_{0,1})$ and $\mathrm{ct}_1 = (\mathrm{ct}_{1,0}, \mathrm{ct}_{1,1})$ be encryptions of m_0 and m_1 under a secret key s, respectively. Then, $m_0 \cdot m_1 \approx [(\mathrm{ct}_{0,0} + s \cdot \mathrm{ct}_{0,1}) \cdot (\mathrm{ct}_{1,0} + s \cdot \mathrm{ct}_{1,1})]_{q_0}$. Therefore, the property that is expected to be conserved to offer homomorphic multiplication is

$$\begin{aligned} m_2 = m_0 \cdot m_1 \approx \big[&\mathrm{ct}_{0,0} \cdot \mathrm{ct}_{1,0} \\ &+ s \cdot (\mathrm{ct}_{0,1} \cdot \mathrm{ct}_{1,0} + \mathrm{ct}_{1,1} \cdot \mathrm{ct}_{0,0}) \\ &+ s^2 \cdot \mathrm{ct}_{0,1} \cdot \mathrm{ct}_{1,1}\big]_{q_0}. \end{aligned}$$

The problem with this construction is that the outcome of a ciphertext multiplication would be a ciphertext composed of three parts, which are the coefficients of powers of s. This is not desirable for storage efficiency, and can also become a significant computational problem for following homomorphic operations, especially for homomorphic multiplications. Thus, a procedure to recover the ciphertext's linearity, i.e. write it as a linear combination of $\{1, s\}$, is crucial in this context.

The relinearization procedure for this scheme extends the polynomial representation of the quadratic coefficient from an element of $R_{\mathcal{C}}$ to an element of $R_{\mathcal{C}+\mathcal{D}}$, for a secondary basis $\mathcal{D} = \{p_0, p_1, \ldots, p_k\}$ coprime to $\mathcal{C}$. A multiplication by an evaluation key, evk, is done in this bigger basis, and then the representation is shrank back to the basis $\mathcal{C}$. These basis conversion steps are done through approximate modulus switching functions referred to as ModUp and ModDown.

Bajard et al. [8] define a fast basis extension procedure from $\mathcal{C}$ to $\mathcal{D}$ as follows:

$$\mathrm{Conv}_{\mathcal{C}\to\mathcal{D}}(a) = \left(\left[\sum_{j=0}^{\ell-1} [a^{(j)} \cdot \hat{q}_j^{-1}]_{q_j} \cdot \hat{q}_j \right]_{p_i} \right)_{0\le i\le k}$$

where $\hat{q}_j = \prod_{j'\neq j} q_{j'}$. This procedure can be used for ModUp, computing the approximated representation of a in a bigger basis. This approximation, in the context of the CKKS, is close enough to add negligible noise to the cryptosystem.

The inverse procedure, ModDown, aims at computing $b \approx P^{-1} \cdot \tilde{b} \in \mathbb{Z}_{\mathcal{C}}$ for $P = \prod_{p\in\mathcal{D}} p$, given as input the representation $\tilde{b} \in \mathbb{Z}_{\mathcal{C}+\mathcal{D}}$.

$$\textsc{ModDown}_{\mathcal{C}+\mathcal{D}\to\mathcal{C}}\left(\left\{\tilde{b}_{\mathcal{C}}, \tilde{b}_{\mathcal{D}}\right\}\right) = \left(P^{-1} \cdot \left(\tilde{b}_{\mathcal{C}}^{(j)} - \mathrm{Conv}_{\mathcal{D}\to\mathcal{C}}(\tilde{b}_{\mathcal{D}})^{(j)}\right)\right)_{0\le j\le \ell}.$$

Notice that, after a homomorphic multiplication, the encoding scaling factor of the outcome is Δ^2. For maintaining the representation precision, CKKS uses a rescaling method to restore the original scaling factor (or an approximation of it). In this sense, one of the residues used to represent the ciphertext is consumed, leading to a $\ell' = (\ell - 1)$-level ciphertext. When $\ell' = 0$, no further rescaling is possible.

Let χ_{key} be a secret key distribution, and χ_{err} an encryption key distribution over the ring R. In practice, χ_{key} is usually defined as a narrow distribution, sampling uniformly from $\{-1, 0, 1\}$, and χ_{err} is taken as a discrete Gaussian. Also, let $\mathcal{C} = \{q_0, q_1, \ldots, q_L\}$ and $\mathcal{D} = \{q_{L+1}, q_{L_2}, \ldots, q_{L+k}\}$ be two RNS basis coprime to each other. In the following we define some primitives relevant for this work:

CKKS.SecKeyGen(1^λ): Sample $s \leftarrow\$\, \chi_{key}$ and set the secret key as $\mathsf{sk} := (1, s)$.

CKKS.PubKeyGen(sk): Sample $\left(a^{(0)}, \ldots, a^{(L)}\right) \leftarrow\$\, R_{\mathcal{C}}$ and $e \leftarrow\$\, \chi_{err}$. Set the public key as $\mathsf{pk} := \left(\mathsf{pk}^{(j)} = (-a^{(j)} \cdot s + e \mod q_j, a^{(j)})_{0\le j\le L}\right)$.

CKKS.RelinKeyGen(sk): Sample $\left(a^{(0)}, \ldots, a^{(k+L)}\right) \leftarrow\$\, R_{\mathcal{C}\bigcup\mathcal{B}}$ and $e \leftarrow\$\, \chi_{err}$. Let $b^{(j)} := -a^{(j)} \cdot s + \left[\prod_{i=0}^{k-1} p_i\right]_{g^j} + e \mod g^j$, for $g_j \in \mathcal{C} \bigcup \mathcal{B}$. Set the relinearization key $\mathsf{rlk} := \left(\mathsf{rlk}^{(j)} = (b^{(j)}, a^{(j)})\right)_{0\le j\le L}$.

CKKS.Encrypt(m, pk): For $m \in R_{\mathcal{C}}$, sample $v \leftarrow\$\, \chi_{enc}$ and $e_0, e_1 \leftarrow\$\, \chi_{err}$. Output the ciphertext $\mathsf{ct} := \left(c^{(j)} = \left[v \cdot \mathsf{pk}^{(j)} + (m + e_0, e_1)\right]_{q_j}\right)_{0\le j\le L}$

CKKS.Decrypt$(\mathsf{ct}, \mathsf{sk})$: Output $[\mathsf{ct} \cdot \mathsf{sk}]_{q_0}$.

CKKS.Add(c_a, c_b): Let $c_a = (a_0^{(j)}, a_1^{(j)})$ and $c_b = (b_0^{(j)}, b_1^{(j)})$ for $j \in \{0, \ldots, L\}$. Output $([a_0^{(j)} + b_0^{(j)}]_{q_j}, [a_1^{(j)} + b_1^{(j)}]_{q_j})_{0\le j\le L}$

CKKS.DR2(c_a, c_b): Let $c_a = (a_0^{(j)}, a_1^{(j)})$ and $c_b = (b_0^{(j)}, b_1^{(j)})$. Output $\left(d_0^{(j)}, d_1^{(j)}, d_2^{(j)}\right) = \left(a_0^{(j)}b_0^{(j)}, a_0^{(j)}b_1^{(j)} + a_1^{(j)}b_0^{(j)}, a_1^{(j)}b_1^{(j)}\right) \in R_{q_j}^3$, for $j \in \{0, \ldots, L\}$.

CKKS.Mul(c_a, c_b): Let $c_a = (a_0^{(j)}, a_1^{(j)})$ and $c_b = (b_0^{(j)}, b_1^{(j)})$.

(a) $\left(d_0^{(j)}, d_1^{(j)}, d_2^{(j)}\right) = \mathrm{CKKS.DR2}(c_a, c_b)$,

(b) Having the representation of d_2 in base $\mathcal{C}$, compute its representation in base $\mathcal{D}$: $\tilde{d}_2 := \textsc{ModUp}_{\mathcal{C}_\ell\leftarrow\mathcal{D}_\ell}(d_2)$

(c) $\tilde{ct}^{(j)} := \tilde{d}_2^{(j)} \cdot \mathsf{evk}^{(j)} \mod q_j$ for $q_j \in \mathcal{C} + \mathcal{D}$.

(d) Having $\tilde{ct}^{(j)}$ in base $\mathcal{C} + \mathcal{D}$, compute its representation in base $\mathcal{C}$: $\hat{c}^{(j)} := \textsc{ModDown}_{\mathcal{D}_\ell\leftarrow\mathcal{C}_\ell}(\tilde{ct}^{(j)})$

(e) Output $\left(\hat{c}_0^{(j)} + d_0^{(j)}, \hat{c}_1^{(j)} + d_1^{(j)}\right)_{0\le j\le L}$.

CKKS.Rescale(c_a): Let a ℓ-level ciphertext $c_a = (a_0^{(j)}, a_1^{(j)})$ for $j \in \{0, \ldots, \ell\}$. Compute $c_b := q_\ell^{-1} \cdot \left(a_0^{(j)} - a_0^{\ell}, a_1^{(j)} - a_1^{\ell}\right)_{0\le j\le \ell-1}$. The result, c_b, is a $(\ell - 1)$-level ciphertext.

CKKS.AddPlain(c, p): Let $c = (a^{(j)}, b^{(j)})$ be a ℓ-level ciphertext and $p \in R_{\mathcal{C}}$ a plaintext, for $j \in \{0, \ldots, \ell\}$. Output $(a^{(j)} + p^{(j)}, b^{(j)})_{0\le j\le L} \in R_{\mathcal{C}}$

CKKS.MulPlain(c, p): Let $c = (a^{(j)}, b^{(j)})$ be a ℓ-level ciphertext and $p \in R_{\mathcal{C}}$

a plaintext, for $j \in \{0, \ldots, \ell\}$. Output $(a^{(j)}p^{(j)}, b^{(j)}p^{(j)})_{0 \leq j \leq L} \in R_{\mathcal{C}}$.

2.4 Hierarchical Transforms

The memory paradigm of GPUs involves different layers that should be considered prior to the computation. Usually, the processing workflow starts with the data copy from the machine's main memory to the GPU global memory, which is the largest but also slowest memory space accessible by CUDA threads. Thus, before computation really starts, data needs to be copied from the global memory to a faster memory. Each CUDA thread is member of a three-dimensional block of threads, which shares a fast but small memory space, called *shared memory*. By doing that, the performance is considerably increased. Yet, this also imposes a constraint on the space consumption.

As discussed by Alves et al., the implementation of the DGT or NTT in memory-constrained devices, such as GPUs, is not straightforward [4]. The synchronization calls needed by these algorithms limits the dimension of the input to the size of a block of threads, or requires the use of a software-based mechanism like Cooperative Threads. To avoid such issue, a hierarchical strategy can be adopted to reduce the size of the transforms to a feasible dimension, which can be easily supported by the processing hardware. Originally proposed by Bailey and revisited by Govindaraju et al. for the FFT, the idea is to split the input polynomial, of degree N, into smaller instances as close as to $\sqrt{N}$ [7, 28].

We present a general description of a forward hierarchical transform, which in our case is either the hierarchical NTT or hierarchical DGT. The forward hierarchical transform is done by executing the following four steps on an N-length integer vector $\mathbf{a}$. We also consider that the arithmetic operations are taken modulo a prime number p. When dealing with the hierarchical NTT transform, we require that $p \equiv 1 \pmod{2N}$. But, when the transform is the DGT, we require that $p \equiv 1 \pmod{4N}$ in order to the $\frac{N}{2}$-th primitive root of i exist modulo p.

1. Apply the weight corresponding to either NTT or DGT to the operand $\mathbf{a}$. When the NTT is the transform, the weight is the negative wrapped convolution, which multiplies the coefficients of $\mathbf{a}$ by powers of the $2N$-th primitive root of unity modulo p. When the transform is the DGT, the weight consists of folding the input vector followed by a multiplication by powers of the $N/2$-th primitive root of i modulo p, denoted $h \pmod{p}$. Despite the transform, the result of this step is assumed to be an N'-length vector.
2. By treating $\mathbf{a}$ as an $N_r \times N_c$-matrix, denoted $\mathbf{A}$, perform N_c simultaneous N_r-length transforms on each column of $\mathbf{A}$.
3. Apply the twisting factor g, which is the N'-th primitive root of unity modulo p, to $\mathbf{a}$ by multiplying each element $\mathbf{A}_{i,j}$ by $g^{i \cdot j} \pmod{p}$.
4. Finally, perform N_r simultaneous N_c-length transforms on each row of $\mathbf{A}$.

We summarize the basic steps of a forward hierarchical transform in Algorithm 2. Notice that $\mathbf{A}_j$ denotes the j-th column of the matrix $\mathbf{A}$. The inverse hierarchical transform is obtained by executing the above four steps in reversed order. This is done by replacing the forward transform in steps 2 and 4 with their inverse counterparts and substituting the primitive roots in steps 1 and 3 by their inverse modulo p.

Algorithm 2 Hierarchical forward transform

```
Require: An $N$-length integer vector $\mathbf{a}$.
Ensure: The operand $\mathbf{A}$ in the hierarchical transform domain.
  $\mathbf{a} = \mathsf{ApplyWeight}(\mathbf{a})$
  for $j = 0;\ j < N_c;\ j = j + 1$ do
      $\mathbf{A}_j = \mathsf{PerformForwardTransform}(\mathbf{A}_j)$
  end for
  $\mathbf{A} = \mathsf{ApplyTwiddleFactor}(\mathbf{A})$
  $\mathbf{A} = \mathsf{TransposeMatrix}(\mathbf{A})$
  for $i = 0;\ i < N_r;\ i = i + 1$ do
      $\mathbf{A}_i = \mathsf{PerformForwardTransform}(\mathbf{A}_i)$
  end for
  return $\mathbf{A}$
```

Consider that we want to perform the polynomial multiplication $c(x) = a(x) \cdot b(x) \in \mathbb{Z}[x]/(x^N + 1)$ by adopting the hierarchical transforms. We consider that $\mathbf{a}$ and $\mathbf{b}$ contain the coefficients of the integer polynomials $a(x)$ and $b(x)$, respectively. Thus, the polynomial multiplication is done by executing the four-step algorithm into both $\mathbf{a}$ and $\mathbf{b}$, by computing the component-wise multiplication on the operands, and by finally computing the inverse hierarchical transform. As a result, we obtain **c**, which holds the coefficients of the polynomial $c(x)$. Notice that, in the NTT domain, the component-wise multiplication is performed in $\mathbb{Z}_p$. On the other hand, in the DGT domain, the product is performed in $\mathbb{Z}_p[i]$ using arithmetic similar to performed over complex numbers.

3 Performance Evaluation of Hierarchical Transforms

This work investigates whether the DGT outperforms the NTT in its hierarchical form as a mechanism to accelerate the CKKS arithmetic in the CUDA architecture. The main differences between them are the input folding and the field arithmetic of the DGT, which is more expensive than in the NTT. However, doing arithmetic operations in $\mathbb{F}(p^2)$ instead of $\mathbb{F}(p)$ offers a higher computational density. These characteristics may improve performance if they use the processing hardware more efficiently. To examine this hypothesis, we conceived two CUDA-based implementations of CKKS using the hierarchical versions of both DGT and NTT, which we refer to as AOA-DGT and AOA-NTT. Both follow the same design decisions, except for their basic data type.

We represent a polynomial as a single array by concatenating its residues. In AOA-NTT, the coefficients are stored as 64-bit unsigned integers, and in AOA-DGT we use a structure composed of two of those. All other implementation decisions follow Alves et al.'s blueprint, as the use of the *double*-CRT representation, encapsulating data simultaneously in the RNS representation and within the transform domain; and the GPU-optimized state machine, avoiding data move between memories and the transform domain [4].

Let $\mathcal{C} = \{q_0, \ldots, q_L\}$ and $\mathcal{D} = \{q_{L+1}, \ldots, q_{L+k}\}$ be the main and secondary RNS basis, as defined in Section 2.3. All time measurements were obtained with a 63-bit prime q_0, and 52-bit primes q_i, for $i > 0$, fixing the scaling factor at 2^{52}. We selected this scaling factor aiming the required precision to execute the logistic regression inference, described at Section 4.

We collected time measurements of both implementations in two distinct scenarios, comparing the transforms as a standalone but also as part of more complex algorithms. The implementations were analyzed using NVIDIA's recommended profiling tool, namely NVIDIA Nsight Systems version 2021.2.1.2 [36]. All executions were performed on a Google cloud instance and the code was compiled using GCC and G++ 8.4.0, and CUDA 11.3. Experiments were executed on either NVIDIA Tesla V100 or Tesla A100 GPUs.

3.1 Direct Comparison: DGT versus NTT

Following the hierarchical strategy, presented in Section 2.4, N-degree polynomials are processed as N_r or N_c-degree, such that $N = N_r \cdot N_c$. Thus, we need N_c blocks of $\lceil N_r/2 \rceil$ threads each to compute step 2 of the algorithm, and then N_r blocks of $\lceil N_c/2 \rceil$ threads for step 4.

The effect of the hierarchical approach can be observed in rings with degrees 4096 and 8192. Table 1 shows execution times for computing the DGT and the NTT on these instances represented in different RNS bases, which offer the best and the worst performance for the DGT compared to the NTT, respectively. As it can be seen in the table, DGT exhibits a consistent slowdown in the smaller instance. Due to DGT's folding procedure, 4096-degree polynomials are folded into 2048-degree polynomials with Gaussian integer coefficients. Since 2048 can be written as $64 \cdot 32$, it means that there will be blocks with $32/2 = 16$ threads running in the GPU.

Modern GPUs' streaming multiprocessors (SMs) process groups of 32 threads at a time, called warps, which are the primary processing unit in a GPU. In this sense, 16-thread blocks are too small and do not reach the CUDA warp size. Since warps are only composed of threads contained in the same block, blocks smaller than 32 imply that SM resources are being wasted, explaining the performance observed on instances of the DGT with a size smaller or equal to 32. In comparison with NTT that does not use folding, the operands are processed as 4096-degree polynomials. Thus, $4096 = 64 \cdot 64$, and all blocks are set with 32 threads, fitting in a warp perfectly. For $N = 8192$, the opposite happens, and the DGT benefits from the SM processing. In this case, some NTT thread blocks have 64 elements, but DGT enters in its optimal setup with 32-thread blocks. Figure 1 expands this analysis for further

Table 1 Latencies for the computation of the DGT and NTT on their hierarchical formulation on 4096- and 8192-degree polynomials represented in RNS bases composed of r elements. Measurements in microseconds were computed as the average of 100 independent executions on an NVIDIA Tesla V100 GPU.

	$N = 4096$						
r	1	5	10	20	30	40	50
DGT	16.2	17.1	18.4	24.7	32.6	39.6	48.4
NTT	13.4	14.3	17.0	22.0	28.5	35.2	43.1
Ratio	1.21	1.20	1.08	1.12	1.14	1.13	1.12
	$N = 8192$						
r	1	5	10	20	30	40	50
DGT	17.7	18.9	23.0	35.0	47.8	60.4	77.0
NTT	14.2	17.2	23.5	36.9	52.2	64.3	76.4
Ratio	1.25	1.10	0.98	0.95	0.92	0.94	1.01

configurations. When $N = 4096$, a considerable slowdown of the DGT is observed, related to its inefficiency in exploiting the hardware's resources. The NTT starts to move from its optimal configuration when $N = 8192$, losing execution efficiency. At the same time, the DGT finds its best performance, in which a speedup in the interval between 10 and 45 residues is observed.

In larger instances, AOA-DGT and AOA-NTT suffer from the increasing consumption of shared memory, which rises bank conflicts, and thread block synchronization becomes more expensive, since blocks must be split among several warps.

3.2 Impact on CKKS Homomorphic Primitives

The NTT and DGT transforms are applicable to the CKKS context for reducing the complexity of polynomial multiplications. However, the consequences go beyond that. For instance, the basis extension methods ModUp and ModDown, described in Section 2.3, cannot be expressed in an arithmetic circuit that can be evaluated in the domains of the transforms. Thus, certain operations, such as homomorphic multiplication, require converting between distinct domains. One would prefer to keep the data structure associated with each transform to avoid conversion costs. In that case, the implementation of those methods would have to consider that data structure and would be affected by its particularities. Thus, Table 1 is not sufficient to decisively conclude about the suitability of each method regarding its employment to CKKS.

Table 2 compares the CKKS' homomorphic addition and multiplication using each transform. When the homomorphic addition is implemented in AOA-DGT, it presents a slowdown of around 10%. This is a straightforward procedure and, intuitively, one would expect the same performance in both implementations since the required number of modular additions is the same. However, this is an extremely memory-bound procedure. Apart from memory transactions, it only requires integer additions. Even the related reductions modulo p can be implemented with a single integer addition by constant-time selection of $a+b$ and $a+b-p$. Hence, the memory overhead outweighs the computational cost. Some of the kernel launch cost can be amortized by executing the entire homomorphic addition in a single CUDA kernel. However, the profiler still indicates that both implementations achieve very low occupancy and warps may stall waiting for load and store transactions to the device's memory. Notice that the implementation in AOA-DGT takes twice the number of input and output operands, thus being more affected by warps stalls.

Algorithm 3 Pseudo-code of CKKS' homomorphic multiplication

```
Require: $\mathsf{ct}_0 \equiv \text{CKKS.Encrypt}(m_0)$ and $\mathsf{ct}_1 \equiv \text{CKKS.Encrypt}(m_1)$, and $\mathsf{evk}$ as an evaluation key.
Ensure: $\mathsf{ct}_2$ such that $\mathsf{ct}_2 \equiv \text{CKKS.Encrypt}(m_0 \times m_1)$.
  $\widehat{\mathsf{ct}}_0 = \mathsf{Transform}(\mathsf{ct}_0)$
  $\widehat{\mathsf{ct}}_1 = \mathsf{Transform}(\mathsf{ct}_1)$
  $\hat{d} = \mathsf{CKKS.DR2}(\widehat{\mathsf{ct}}_0, \widehat{\mathsf{ct}}_1)$
  $d = \mathsf{InverseTransform}(\hat{d})$
  $e_2 := \mathsf{ModUp}_{\mathcal{C}_\ell \leftarrow \mathcal{D}_\ell}(d_2)$
  $\hat{e}_2 = \mathsf{Transform}(e_2)$
  $\widehat{\mathsf{ct}} := \hat{e}_2 \times \mathsf{evk}$
  $\mathsf{ct} = \mathsf{InverseTransform}(\widehat{\mathsf{ct}})$
  $a := \mathsf{ModDown}_{\mathcal{D}_\ell \leftarrow \mathcal{C}_\ell}(\mathsf{ct})$
  $\mathsf{ct}_2 := (a_0 + d_0, a_1 + d_1)$
  return $\mathsf{ct}_2$
```

In contrast, homomorphic multiplication is a much more complex operation, composed of arithmetic operations, basis extensions, and transformations between the polynomial and transform domain, as shown in Algorithm 3 and discussed

Fig. 1 Ratio (DGT/NTT) of the DGT and NTT execution time for different polynomial degrees and varying sizes of RNS basis. Measurements computed as the average of 100 independent executions on an NVIDIA Tesla V100 GPU.

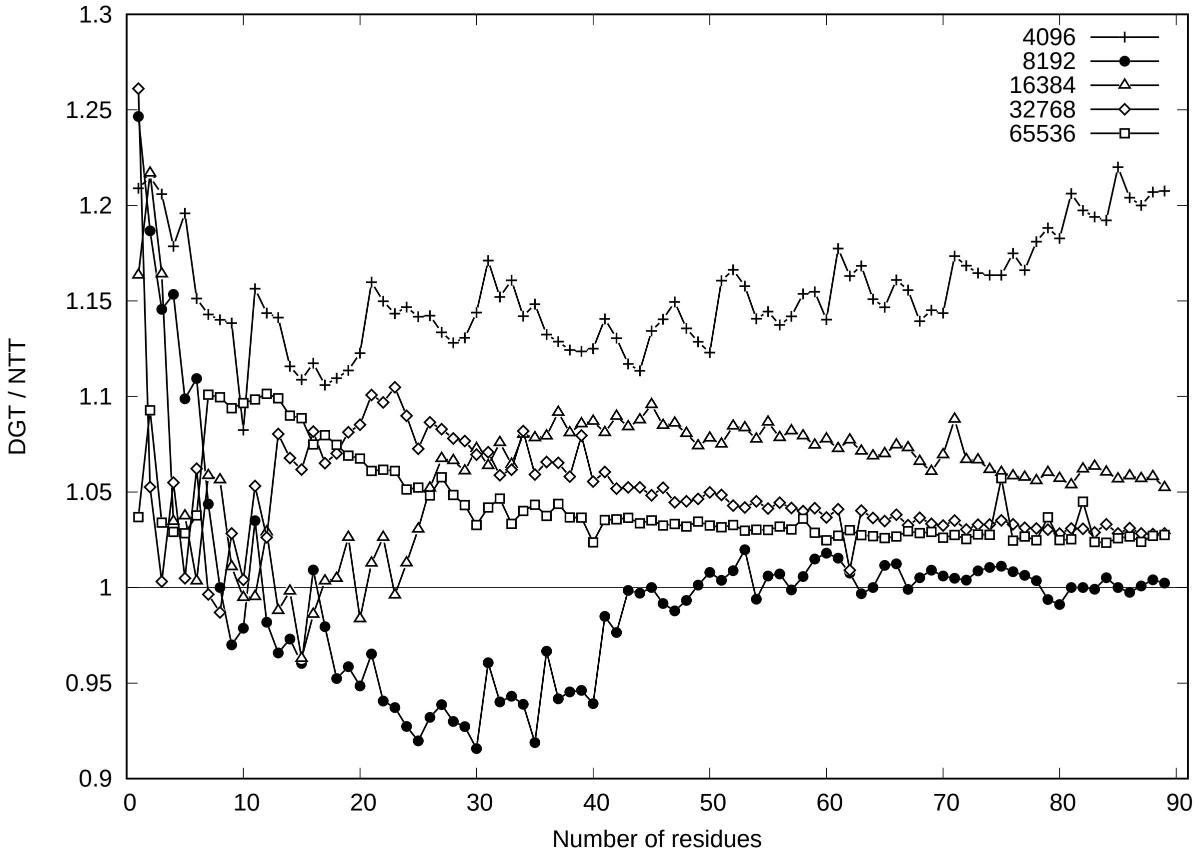

Table 2 Comparison of homomorphic operations using the DGT and the NTT to perform the multiplication of 2^n-degree polynomials with $\log q = 323$, providing at least 80-bit security level when $n \geq 13$. Rescaling is not considered for homomorphic multiplication. Measurements, in microseconds, computed as the average of 100 independent executions on an NVIDIA Tesla V100 GPU.

	Hom. Add.			Hom. Mul.		
n	**DGT** (μs)	**NTT** (μs)	**DGT/NTT**	**DGT** (μs)	**NTT** (μs)	**DGT/NTT**
12	4.4	4.1	1.07	188.6	175.8	1.07
13	5.7	5.5	1.04	237.9	251.6	0.95
14	8.8	8.3	1.06	380.7	395.1	0.96
15	15.9	14.2	1.12	652.4	725.4	0.90
16	30.8	25.5	1.21	1232.9	1402.8	0.88

in Section 2.3. In Table 2, AOA-NTT presents a slowdown that increases with the ring degree. This result appears to contradict the observation presented in Section 3.1, when we observed a better performance for the NTT on these parameters.

In Table 3, we provide experimental results for each component of the homomorphic multiplication, allowing them to be observed separately on a much bigger instance. NVIDIA's profiler tool reveals that the basis extension procedures occupy a critical role in homomorphic multiplication. In

Table 3 Comparison of the latency needed for each component of a homomorphic multiplication algorithm using the DGT and the NTT to perform polynomial multiplication. We also compare them with two distinct methods for basis conversion, a canonical and an optimized version applied to AOA-NTT to increase its arithmetic density. The canonical fast basis extension algorithms are implemented following Algorithm 4 whereas the optimized uses Algorithm 5. These values were obtained through NVIDIA's Nsight Systems 2021.2.1.2 on a machine with an NVIDIA Tesla V100 GPU. The results were evaluated on 2^{16}-degree polynomials with $\log q = 1831$, composed by 53- and 54-element main and secondary basis, respectively.

	DGT (μs)	**NTT** (μs)	**Ratio**	**Opt.** (μs)	**Ratio**
ModUp	2377.0	4928.9	0.48	2592.8	0.92
ModDown	1659.2	1854.7	0.89	1896.6	0.87
Transforms	1131.2	1054.8	1.07	1062.8	1.06
Integer Op.	354.3	227.3	1.56	203.2	1.74
Total	5521.7	8065.7	0.68	5755.5	0.96

Algorithm 4 Canonical basis extension

```
Require: $a_\mathcal{C}$ an $N$-degree polynomial represented in the main RNS basis $\mathcal{C} = \{q_0, \ldots, q_\ell\}$.
Ensure: $a_\mathcal{D}$ an $N$-degree polynomial represented in the secondary RNS basis $\mathcal{D} = \{p_0, \ldots, p_k\}$.
  for $i = 0;\ i \leq k;\ i = i + 1$ do
    $a_\mathcal{D}^{(i)} = \{0, \ldots, 0\}$
    for $j = 0;\ j \leq \ell;\ j = j + 1$ do
      for $z = 0;\ z < N;\ z = z + 1$ do
        $x = a_\mathcal{C}^{(j)}[z]$
        $\text{aux} = x \cdot \hat{q}_j^{-1}$
        $\text{aux} = \text{aux} \mod q_j$
        $\text{aux} = \text{aux} \cdot \hat{q}_j$
        $\text{aux} = \text{aux} \mod p_i$
        $\text{aux} = \text{aux} + a_\mathcal{D}^{(i)}[z]$
        $\text{aux} = \text{aux} \mod p_i$
        $a_\mathcal{D}^{(i)}[z] = \text{aux}$
      end for
    end for
  end for
  return $a_\mathcal{D}$
```

AOA-NTT, ModUp takes 61% of the primitive's computing time, while ModDown takes 23%. Also, the procedures ModUp and ModDown consumes 43% and 30% of the overall running time of AOA-DGT, respectively. For the results in the first two columns, both basis switching methods were implemented as in Algorithm 4. Specially, the profiling tool indicated that ModUp is roughly 2× slower in AOA-NTT in comparison with AOA-DGT, and that ModDown presents a non-negligible slowdown of 11%.

Our implementation of ModUp for the NTT issues 2.2× more instructions than the DGT approach, suggesting that the processor scheduler is being less efficient. We verified this hypothesis by refactoring our implementation according to Algorithm 5. Experimental results for this optimized version are referred to as "Opt." in Table 3. Now, each thread handles two coefficients, and same-instruction operations are called sequentially, inducing the processor into dual issue mode. This operation emulates the behavior of the DGT, which benefits from the input's folding. Table 3 shows this optimization partially solves the slowdown for ModUp, although not affecting ModDown. No benefit could be measured for this technique on DGT, suggesting that the previous approach already saturates the processor's dual-issue capability.

Table 4 Comparison of homomorphic operations using the DGT and the NTT to accelerate polynomial multiplication for 2^n-degree polynomials and $\log q = 323$, providing at least 80-bit security level when $n \geq 13$. Rescaling is not considered for homomorphic multiplication. This experiment uses the Algorithm 5 on AOA-NTT, which increases arithmetic density per thread for the basis extension algorithms. Measurements, in microseconds, computed as the average of 100 independent executions on an NVIDIA Tesla V100 GPU.

	Hom. Mul. (μs)		
n	**DGT**	**NTT**	**Ratio**
12	188.6	175.4	1.08
13	237.9	240.8	0.99
14	380.7	381.0	1.00
15	652.4	681.6	0.96
16	1232.9	1318.6	0.94

The effect of this optimized version of ModUp on homomorphic multiplication is summarized in Table 4. By replacing that method we can observe a considerable performance gain that matches DGT's implementation in most cases. This result goes towards Badawi et al.'s claim that the DGT better fits CUDA's processing paradigm [6]. The

Algorithm 5 Optimized basis extension

```
Require: $a_{\mathcal{C}}$ an $N$-degree polynomial represented in the main RNS basis $\mathcal{C} = \{q_0, \dots, q_\ell\}$, for $N \equiv 0 \mod 2$.
Ensure: $a_{\mathcal{D}}$ an $N$-degree polynomial represented in the secondary RNS basis $\mathcal{D} = \{p_0, \dots, p_k\}$, for $N \equiv 0 \mod 2$.
  $N_h \coloneqq N/2$
  for $i = 0;\ i \leq k;\ i = i + 1$ do
    $a_{\mathcal{D}}^{(i)} = \{0, \dots, 0\}$
    for $j = 0;\ j \leq \ell;\ j = j + 1$ do
      for $z = 0;\ z < N_h;\ z = z + 1$ do
        $x_0\ ,\ x_1 = a_{\mathcal{C}}^{(j)}[z]\ ,\ a_{\mathcal{C}}^{(j)}[z + N_h]$
        $\mathrm{aux}_0\ ,\ \mathrm{aux}_1 = x_0 \cdot \hat{q}_j^{-1}\ ,\ x_1 \cdot \hat{q}_j^{-1}$
        $\mathrm{aux}_0\ ,\ \mathrm{aux}_1 = \mathrm{aux}_0 \mod q_j\ ,\ \mathrm{aux}_1 \mod q_j$
        $\mathrm{aux}_0\ ,\ \mathrm{aux}_1 = \mathrm{aux}_0 \cdot \hat{q}_j\ ,\ \mathrm{aux}_1 \cdot \hat{q}_j$
        $\mathrm{aux}_0\ ,\ \mathrm{aux}_1 = \mathrm{aux}_0 \mod p_i\ ,\ \mathrm{aux}_1 \mod p_i$
        $\mathrm{aux}_0\ ,\ \mathrm{aux}_1 = \mathrm{aux}_0 + a_{\mathcal{D}}^{(i)}[z]\ ,\ \mathrm{aux}_1 + a_{\mathcal{D}}^{(i)}[z + N_h]$
        $\mathrm{aux}_0\ ,\ \mathrm{aux}_1 = \mathrm{aux}_0 \mod p_i\ ,\ \mathrm{aux}_1 \mod p_i$
        $a_{\mathcal{D}}^{(i)}[z]\ ,\ a_{\mathcal{D}}^{(i)}[z + N_h] = \mathrm{aux}_0\ ,\ \mathrm{aux}_1$
      end for
    end for
  end for
  return $a_{\mathcal{D}}$
```

folding property of DGT naturally increases the arithmetic density, which benefits the execution on GPUs.

4 Case Study: Homomorphic Logistic Regression

Logistic Regression (LR) is a learning algorithm widely used to solve classification problems. Basically, LR tries to model dependence between variables [19] For instance, in a binary classification problem, LR takes a dataset composed by n records of the form $(y_i, \mathbf{x}_i)$, with $y_i \in \{0, 1\}$ and $\mathbf{x}_i \in \mathbb{R}^d$. Its objective is to predict the value of y given $\mathbf{x}$. For that, it assumes that the distribution of y given $\mathbf{x}$ is

$$\Pr[y = 1 \mid \mathbf{x}] \coloneqq \sigma(-\mathbf{x}'^{\top} \mathbf{w}), \qquad (3)$$

for some fixed vector $\mathbf{w}$ of weights, $\mathbf{x}'_i = (1 \mid \mathbf{x}_i) \in \mathbb{R}^{d+1}$, and the Sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$. Thus, by having an approximation $\mathbf{w}_{\text{approx}}$ for $\mathbf{w}$, we can infer y with a predictable accuracy by evaluating $\sigma(-\mathbf{x}'^{\top} \mathbf{w}_{\text{approx}})$.

LR can be seen as a neural network composed by a single hidden unit that uses the Sigmoid as the activation function. Its implementation resembles some of the challenges of implementing more complex neural networks, as selecting homomorphic compatible approximations of those functions [11]. Hence, this is a relevant application for homomorphic encryption and, in particular, to evaluate using the DGT or the NTT in an implementation of CKKS.

The computation of $\mathbf{w}$ is done in the training phase. A training dataset is used to compute a $\mathbf{w}_{\text{approx}}$ that sufficiently approximates $\mathbf{w}$, and a test dataset is used to evaluate the quality of the approximation. The Gradient Descent method is a common strategy for training [42]. It selects an initial $\mathbf{w}^0_{\text{approx}}$ and repeatedly computes $\mathbf{w}^j_{\text{approx}} = \mathbf{w}^{j-1}_{\text{approx}} - s \cdot \Delta \cdot J(\mathbf{x}_i)$ until a certain objective is satisfied (e.g. a certain number of iterations is executed), for s an arbitrary step size and J the loss function related to the problem. By modifying $\mathbf{w}_{\text{approx}}$ in the negative direction of the gradient of J, we minimize this function, obtaining a better approximation for the weight vector. However, this is computationally costly and a delicate procedure that may require thousands of multiplications to achieve a suitable $\mathbf{w}_{\text{approx}}$. Moreover, it can also require human-supervised iterations to adjust the network topology if we consider more complex neural networks. Hence, since training is done much less frequently than inference, in this work, we focus only on the effects of the hierarchical transforms in the inference phase, when predictions are done by evaluating Equation 3 using $\mathbf{w}_{\text{approx}}$.

Lastly, it is important to notice that the Sigmoid function cannot be computed homomorphically. In

this sense, similar works that also implement LR inference on FHE schemes avoid its computation by just taking the outcome of $\mathbf{x}'^{\top}\mathbf{w}$ as the classification result [10]. An alternative approach is to approximate the computation of the Sigmoid function using the related Taylor series expansion [29]. The former strategy can retain the classification result but fails to compute the probability of a specific record belonging to a particular class. Conversely, approximating the Sigmoid function requires several additional multiplicative levels per ciphertext since such approximation is usually made using around 4 and 8-degree polynomials according to the required precision.

Implementation

We assume a scenario in which both the training and the model was already computed. It could be done over plaintexts through a standard library as `scikit-learn` or `PyTorch` [38, 39], or over ciphertexts [10, 11, 19]. Then, CUDA-enabled nodes need to classify encrypted data using the model given as input. This model could be made available as plaintext, contemplating cases in which the model owner is contracted to evaluate third-party data, or in encrypted form, when the computation is performed by an entity that should not have access to the model.

We follow other works in the literature that apply learning algorithms to the MNIST dataset [33]. The MNIST dataset is a classical data collection of handwritten digits, composed of black and white images representing digits between 0 and 9, each having $28 \times 28 = 784$ pixels. These images are split into a training and a test set with $60,000$ and $10,000$ records, respectively. Moreover, each image is unique regarding the handwritten style and the expected complexity for its interpretation. So, the digit recognition problem involves classifying m images among $d =$ 10 classes of digits, each image having its pixel columns serialized, composing $n = 784$-element arrays.

We trained a model using a simple Python script that applied the `scikit-learn` library's LR implementation to the training set of images. The outcome is a model $\mathcal{M}$ that indicates an accuracy of 0.9167 when evaluated over the test set. This model, as Equation 3 suggests, is simply a $d \times n$ matrix of real numbers, represented using the `float` data type, such that each row relates to a classification index. So, it follows that $d = 10$ and $n = 784$.

In this MNIST context, two ciphertext designs were evaluated, as follows.

Direct: In a simple implementation, we encrypt each row of the model in a ciphertext, having their columns distributed through the slots. So, a single ciphertext stores all the columns related to a particular digit. This implies that d ciphertexts are needed to fully encrypt the model $\mathcal{M}$. We perform a similar process to encrypt images, encrypting each image into a single ciphertext. Thus, a set of m images becomes a set of m ciphertexts, each one using n slots, as shown in Algorithm 7.

Transposed: The direct approach has memory consumption efficiency but requires a sequence of slots rotations to execute LR's inference. An alternative to that is the transposition of the operands. An n-pixel image dataset with m records becomes a matrix of n rows and m columns in which each row is encrypted to a single ciphertext. However, the model encryption requires each element to be encrypted in a single ciphertext, so we can compute the inner product. This implies a considerable increase in memory consumption. The model encryption in this case requires $n \cdot d$ m-slot ciphertexts. This design is presented in Algorithm 8.

In more detail, Algorithm 7 receives an encrypted set of images such that each image is encapsulated in a single ciphertext. Moreover, it also receives the trained model, which can be encrypted or *exposed* (in plaintext). Let $\mathcal{M} = \mathbb{R}^d \times \mathbb{R}^n$ be the model. If encrypted, the algorithm receives a vector $\mathbf{W} = \{\text{CKKS.Encrypt}(w_{i,*}) \mid w_{i,*} \in \mathcal{M}\}$. Otherwise, it receives $\mathbf{W} = \{w_{i,*} \mid w_{i,*} \in \mathcal{M}\}$.

The input of Algorithm 8 is transposed regarding Algorithm 7. This time, the set of images is encrypted such that each ciphertext stores a single pixel of all images. Thus, a m-sized set of n-pixel images becomes n ciphertexts of m slots. When executed with the exposed model, it takes as input a matrix $\mathbf{W} = \{w_{i,j} \mid w_{i,j} \in \mathcal{M}^\top\}$. Otherwise, it receives a matrix of ciphertexts such that $\mathbf{W} = \{\text{CKKS.Encrypt}(\{w_{j,i}, \ldots, w_{j,i}\}) \mid w_{i,j} \in \mathcal{M}^\top\}$.

Initially, we consider that the inference is executed without computing the Sigmoid function. In this case, the direct strategy requires $d+m$ ciphertexts with at least n available slots to encrypt images and the model. The transposed strategy requires $n \cdot (d+1)$ ciphertexts containing at least m slots. Hence, the direct and the transposed ciphertext designs require $d+m = 10+10,000 = 10,010$ and $n \cdot (d+1) = 784 \cdot 11 = 8624$ ciphertexts, respectively. Also, the direct method performs two homomorphic multiplications to compute the inner product whereas the transposed requires only one homomorphic multiplication.

In the transposed strategy, the number of multiplications and the encoding of the MNIST test set of images require 10,000 slots. Thus, since our implementation only supports power-of-2 polynomials, we need that $N = 32,768$ and $\log q = 115$, composed by a 2-element RNS basis. We use the LWE estimator of Albrecht, Player, and Scott [3] to estimate the hardness of the proposed parameter set against the fastest LWE solvers currently known. For that, we obtain that the cost of running lattice attacks against the underlying LWE instance is at least 2^{1343}, comprehending the cost to run the uSVP variant of the primal lattice attack.

Considering that a maximum number of 784 slots are needed per ciphertext in the direct design, we would require that $N = 2,048$ and $\log q = 167$, composed by a 3-element RNS basis. However, the estimated hardness of such a parameter set is equivalent to a 47-bit security level, not surpassing the 100-bit security level threshold. By choosing $N = 8,192$, we obtain a parameter set with 181-bit security. Consequently, given the high-degree of the polynomial rings, the memory consumption for executing the LR's inference over the entire test set of the MNIST dataset is 3.42 GB and 8.1 GB for direct and transposed versions, respectively.

Moreover, the computation of $p_{i,j}$, the probability of the image i being classified as the digit j, is partially solved by a single homomorphic multiplication between the i-th encrypted image and the j-th encrypted row of $\mathcal{M}$. After that, each slot of the resulting ciphertext will contain the multiplication between each slot of the operands. To conclude the inner product, we need to sum all the slots of a ciphertext. That can be done as shown in Algorithm 6, which depends on a slot rotation done homomorphically [14]. Let $c' = \mathsf{rotate}(c, i)$. If c' is a rotation of k slots of c, and s_i is the i-th slot of c, then $s'_{i+k \mod n} = s_i$, where s'_j is the j-th slot of c'.

Table 5 Execution times of our implementations of Algorithm 6 on AOA-DGT and AOA-NTT for cyclotomic polynomial rings with dimension $N = 2^{n+1}$. The optimized basis extension approach is used in AOA-NTT. We consider $\log q = 167$, that offers 80-bit security in all cases, and $\log q = 323$, that achieves this security level when $n \geq 13$. Measurements taken on a Google Cloud instance with an NVIDIA Tesla V100 GPU.

Sumslots						
$\log(q)$	323			167		
n	**DGT**	**NTT**	**Ratio**	**DGT**	**NTT**	**Ratio**
12	1.34	1.21	1.11	1.05	0.98	1.08
13	1.92	1.89	1.01	1.31	1.22	1.07
14	3.30	3.27	1.01	2.02	1.90	1.07
15	6.44	6.23	1.03	3.44	3.41	1.01
16	13.34	12.84	1.04	6.87	6.59	1.04

Algorithm 6 sumslots – Sum all slots of a ciphertext

```
Require: A ciphertext ct with $n$ slots.
Ensure: A ciphertext ct' with $n$ slots such that
    each slot is the summation of every ct slot.
  ct' = copy(ct)
  for $i = n/2$; $i \geq 1$; $i = i/2$ do
      aux = rotate(ct', $i$)
      ct' = ct' + aux
  end for
  return ct'
```

Algorithm 6, however, has a high inherent cost due to the rotation procedure. Table 5 shows the latencies measured in our implementations. When compared to the costs for homomorphic multiplication, presented in Table 3, it becomes clear that the slots summation is the most costly part of the LR inference with the direct approach in both AOA-DGT and AOA-NTT.

Let $s = \{s_0, s_1, \ldots, s_{n-1}\}$ be the slots of a ciphertext c. The outcome of the c-slot summation will be a ciphertext c' such that all its slots will be equal to $\sum_{i=0}^{n-1} s_i$. To improve space efficiency, we would like to store information about the suitability of an image i to all candidate digits in a single ciphertext, with d being the slot related to the digit d. We use $\mathsf{DiscardSlotsExcept}(c, d)$ for that. It returns the homomorphic multiplication of c by the encryption of $a = \{a_0, a_1, \ldots, a_{n-1}\}$ such that $a_i = 1$, if $i = d$, and 0, otherwise.

By following the direct approach, we can compute homomorphically a vector of ciphertexts, denoted **pred**, such that element with index i stores the encryption of the inner product between the image i and the weight vector related to each class, as shown in Algorithm 7. By design, **pred** supports n slots but only d will be possibly different than zero.

Algorithm 7 Direct version of an encrypted LR inference

```
Require: $\mathbf{W}$ as a $d$-element representation of the trained model; $\mathbf{X} \in \mathbb{R}^m \times \mathbb{R}^n$ as a set of images; and $X_i = \text{CKKS.Encrypt}(x_i)$ for $x_i \in \mathbf{X}$.
Ensure: $c \in \mathbb{Z}^n$ such that $c_i = d$, if $x_i$ is classified as the digit $d$.
  ———— Online phase ————
  for $j = 0;\ j < d;\ j = j + 1$ do
      for $i = 0;\ i < m;\ i = i + 1$ do
          $p = \mathsf{sumslots}(X_i \cdot W_j)$
          $p = \mathsf{DiscardSlotsExcept}(p, d)$
          $\text{pred}_i = \text{pred}_i + p$
      end for
  end for
  ———— Offline phase ————
  $\text{pred} = \text{CKKS.Decrypt}(\text{pred})$
  for $i = 0;\ i < m;\ i = i + 1$ do
      $c_i = \mathsf{argmax}(\text{pred}_i)$
  end for
  return $c$
```

On the other hand, the transposed design does not need the $\mathsf{sumslots}$ procedure to compute the inner product. Instead, the operands' transposition enables its replacement by a sequence of homomorphic multiplications and additions, as seen in Algorithm 8. As we avoid the expensive Algorithm 6, a considerable performance improvement of 70× is observed when the model is encrypted. When the model is given as plaintext, the performance is improved by 700×, as presented in Table 6.

Algorithm 8 Transposed version of an encrypted LR inference

```
Require: $\mathbf{W}$ as a $n \times d$ matrix representing the transposed trained model; $\mathbf{X}^\top \in \mathbb{R}^m \times \mathbb{R}^n$ as a set of transposed images; and $X_i = \text{CKKS.Encrypt}(x_i)$ for $x_i \in \mathbf{X}^\top$.
Ensure: $c \in \mathbb{Z}^n$ such that $c_i = d$, if $x_i$ is classified as the digit $d$.
  ———— Online phase ————
  for $j = 0;\ j < d;\ j = j + 1$ do
      $\text{pred}_j = 0$
      for $i = 0;\ i < n;\ i = i + 1$ do
          $\text{pred}_j = \text{pred}_j + X_i \cdot W_{j,i}$
      end for
  end for
  ———— Offline phase ————
  $\text{pred} = \text{CKKS.Decrypt}(\text{pred})$
  for $i = 0;\ i < m;\ i = i + 1$ do
      $c_i = \mathsf{argmax}(\text{pred}_{*,i})$
  end for
  return $c$
```

Algorithms 7 and 8 are composed of two phases. The first one, called *online phase*, is the more computationally intensive and, as aforementioned, can be executed homomorphically on a powerful device. The dataset is kept encrypted, and no knowledge of the decryption key is needed. The second, called *offline phase*, is the result delivery phase when the matrix of probabilities is decrypted, and the prediction is made by selecting the index of the maximum element of the array. This is a much less intensive step that can be executed even on a low-power device. However, as discussed by Bajard et al., the `sign` or `argmax` works also as filters that truncate data that can be used to retrieve confidential information. Thus, by not executing these functions homomorphically, an adversary may be able to leak the entire trained model [9]. Moreover, in some cases, as with the Support Vector Machine algorithm, this could also imply leakage of the input data. Thus, the solution presented in those algorithms assumes a secure executing environment for the decryption keys and the decrypted matrix of probabilities.

Table 6 presents the latencies for the minimum parameter set that offers at least a 128-bit security level and provides the required multiplicative depth for each approach. The accumulated slowdown for $\mathsf{sumslots}$ on AOA-DGT, presented in Table 5, impacts its latency on the direct approach. Nonetheless, the transpose approach does not depend on it and thus AOA-DGT and AOA-NTT show similar execution times.

No impact was detected for the AOA-NTT implementation done with the optimization discussed on Section 3.2. The ModUp function performance is directly impacted by the basis sizes, and in these instances, with 3 and 2-sized bases, that optimization does not seem relevant.

When we consider the canonical formulation of LR inference, which depends on the Sigmoid function, something different can be observed. We follow the literature and approximate this function using a polynomial obtained by the truncation at the 8-th term of its Taylor series approximation [1]. Evaluated through Horner's rule, it increases by 8 the number of homomorphic multiplications needed for inference, affecting performance and memory consumption. Table 7 presents our measurements. Motivated by the increased memory requirement, we executed this experiment on an NVIDIA Tesla A100. In the direct design, the additional multiplications imply a reduction of the security level of about 50 bits if we choose $N = 2^{13}$. Thus, by setting $N = 2^{14}$, we could obtain a security level of at least 80 bits.

In bigger instances, the higher arithmetic density of AOA-DGT implied in speedups in all cases when compared to the non-optimized AOA-NTT. However, when the optimization of the ModUp function takes place, AOA-NTT is capable of reversing that scenario. The transposed design, implemented with a plaintext model, presents a consistent similarity between both implementations. Homomorphic addition and multiplication between ciphertexts and plaintexts are coefficient-wise operations, as shown in Section 2.3.

The folding procedure used by the DGT does not imply the reduction of the number of required operations. However, the scalability of the DGT-based implementation highlights and reduces the execution time when more complex methods are considered as basis extension procedures rotation. This behavior agrees with the conclusion in Section 3.2, which suggests that the DGT implementation better fits CUDA's processing paradigm and that its characteristics have to be ported to AOA-NTT so we can observe a similar, or even superior, performance.

5 Conclusion

Most implementations based on the RLWE problem use the NTT to efficiently compute the polynomial multiplication in the ring $\mathbb{Z}_p[x]/(x^N + 1)$ for N a power of two and p a prime. However, recent works [2, 4, 6] claim that polynomial multiplication can be more efficiently implemented on GPUs using the DGT instead of the NTT.

In this context, we developed two implementations of the CKKS cryptosystem following the same blueprint but diverging on the transform, using either the DGT or the NTT for polynomial multiplication. We refer to them as AOA-DGT and AOA-NTT. We performed benchmarks on both implementations by first directly comparing the NTT and DGT transforms as standalone, and then their implementations of CKKS' homomorphic multiplication. After that, we extended our evaluation to the context of logistic regression inference.

Considering the latency for the DGT or NTT isolated, we observed an overall similarity between both implementations in bigger instances, i.e., the ring dimension ranging from 16384 to 65536. For smaller instances, the NTT outperforms the DGT but with a clear trend towards equalization in large rings. The exception occurs on 8192-degree polynomial rings when the DGT reaches its warp efficiency peak. In that case, we observed a speedup of up to 10% for AOA-DGT. Nevertheless, we could not observe the same behavior for the homomorphic multiplication in CKKS. In this case, AOA-DGT rapidly surpasses the AOA-NTT performance. A deep analysis through NVIDIA's profiler tool showed that the DGT data structure provides a higher arithmetic density, which efficiently explores the processing hardware, especially on the methods for basis extension. We were able to successfully match the performance of both implementations by porting the DGT data representation to AOA-NTT

Lastly, we evaluated the impact of both AOA-DGT and AOA-NTT on a logistic regression inference performed homomorphically over ciphertexts. We trained a model to classify handwritten digits in the MNIST dataset and measured the execution time per image in different configurations, exploring two ciphertext designs. We concluded that, in bigger instances, AOA-DGT presents a considerable speedup when compared

Table 6 Comparison of the latency per record required to compute the online phase of the LR inference over records in the test set of the MNIST database with the model encrypted and as plaintext. The basic design runs with $N = 2^{13}$ and $\log q = 167$ while the transposed version runs with $N = 2^{15}$ and $\log q = 115$, offering 172-bit and 1343-bit security level, respectively, according to Albrecht's estimator [3]. Measurements in microseconds were taken on a Google Cloud instance with an NVIDIA Tesla V100 GPU.

Model	**Encrypted**	**Exposed**	**Encrypted**$^\top$	**Exposed**$^\top$
DGT	15461.2	13885.9	226.8	15.6
NTT	14396.8	12827.6	238.5	15.2
DGT/NTT	1.07	1.08	0.95	1.03

Table 7 Comparison of the latency per record required to compute the online phase of the LR inference over records in the test set of the MNIST database with the model encrypted and as plaintext following the textbook algorithm, with the homomorphic computation of the Sigmoid function. The basic design runs with $N = 2^{14}$ and $\log q = 583$ while the transposed version runs with $N = 2^{15}$ and $\log q = 531$, offering 94-bit and 225-bit security level, respectively, according to Albrecht's estimator [3]. Measurements in microseconds were taken on a Google Cloud instance with an NVIDIA Tesla A100 GPU.

Model	**Encrypted**	**Exposed**	**Encrypted**$^\top$	**Exposed**$^\top$
DGT	60279.7	56013.2	475.7	33.1
NTT	65998.5	60968.6	562.6	33.4
DGT/NTT	0.91	0.92	0.85	0.98
NTT-Opt	58200.0	56053.7	517.6	32.5
DGT/NTT-Opt	1.04	1.00	0.92	1.02
NTT/NTT-Opt	1.13	1.09	1.09	1.03

with a standard implementation on the AOA-NTT, i.e. without methods that increase arithmetic density on CUDA kernels.

Hence, our results indicate that the data representation of the DGT on a CKKS implementation increases the implementation's overall performance by assisting the programmer to take implementation decisions that more efficiently explore the GPU hardware.

Acknowledgments. This work was supported in part by the Brazilian National Council for Scientific and Technological Development (CNPq), grants number 164489/2018-5 and 203175/2019-0; and the Brazilian Coordination for the Improvement of Higher Education Personnel Foundation (CAPES) grant number 1591123. We specially thank Google for GCP Research Credits Program under number 106101194491; the Concordium Blockchain Research Center at Aarhus University (COBRA), Denmark; and the European Research Council (ERC) under the European Unions's Horizon 2020 research and innovation programme under grant agreement No. 803096 (SPEC).

References

[1] Abramowitz, M., I.A. Stegun, and R.H. Romer. 1988. Handbook of mathematical functions with formulas, graphs, and mathematical tables.

[2] Al Badawi, A., B. Veeravalli, and K.M.M. Aung 2018. Efficient polynomial multiplication via modified discrete galois transform and negacyclic convolution. In *Future of Information and Communication Conference*, pp. 666–682. Springer.

[3] Albrecht, M.R., R. Player, and S. Scott. 2015. On the concrete hardness of learning with errors. *J. Math. Cryptol. 9*(3): 169–203 .

[4] Alves, P.G.M.R., J.N. Ortiz, and D.F. Aranha. 2020. Faster homomorphic encryption over gpgpus via hierarchical DGT. *IACR Cryptol. ePrint Arch.* 2020: 861 .

[5] Badawi, A.A., Y. Polyakov, K.M.M. Aung, B. Veeravalli, and K. Rohloff. 2021. Implementation and performance evaluation of RNS variants of the BFV homomorphic encryption

scheme. *IEEE Trans. Emerg. Top. Comput.* *9*(2): 941–956. https://doi.org/10.1109/TETC.2019.2902799 .

[6] Badawi, A.A., B. Veeravalli, C.F. Mun, and K.M.M. Aung. 2018. High-performance FV somewhat homomorphic encryption on gpus: An implementation using CUDA. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* *2018*(2): 70–95. https://doi.org/10.13154/tches.v2018.i2.70-95 .

[7] Bailey, D.H. 1990. FFTs in external or hierarchical memory. *J. Supercomput.* *4*(1): 23–35 .

[8] Bajard, J., J. Eynard, M.A. Hasan, and V. Zucca 2016. A full RNS variant of FV like somewhat homomorphic encryption schemes. In R. Avanzi and H. M. Heys (Eds.), *Selected Areas in Cryptography - SAC 2016 - 23rd International Conference, St. John's, NL, Canada, August 10-12, 2016, Revised Selected Papers*, Volume 10532 of *Lecture Notes in Computer Science*, pp. 423–442. Springer.

[9] Bajard, J., P. Martins, L. Sousa, and V. Zucca. 2020. Improving the efficiency of SVM classification with FHE. *IEEE Trans. Inf. Forensics Secur.* 15: 1709–1722. https://doi.org/10.1109/TIFS.2019.2946097 .

[10] Benaissa, A., B. Retiat, B. Cebere, and A.E. Belfedhal. 2021. Tenseal: A library for encrypted tensor operations using homomorphic encryption.

[11] Bergamaschi, F., S. Halevi, T.T. Halevi, and H. Hunt 2019. Homomorphic training of 30, 000 logistic regression models. In R. H. Deng, V. Gauthier-Umaña, M. Ochoa, and M. Yung (Eds.), *Applied Cryptography and Network Security - 17th International Conference, ACNS 2019, Bogota, Colombia, June 5-7, 2019, Proceedings*, Volume 11464 of *Lecture Notes in Computer Science*, pp. 592–611. Springer.

[12] Bos, J.W., K.E. Lauter, J. Loftus, and M. Naehrig 2013. Improved security for a ring-based fully homomorphic encryption scheme. In M. Stam (Ed.), *Cryptography and Coding - 14th IMA International Conference, IMACC 2013, Oxford, UK, December 17-19, 2013. Proceedings*, Volume 8308 of *Lecture Notes in Computer Science*, pp. 45–64. Springer.

[13] Brakerski, Z., C. Gentry, and V. Vaikuntanathan. 2011. Fully homomorphic encryption without bootstrapping. *Electron. Colloquium Comput. Complex.*: 111 .

[14] Cheon, J.H., K. Han, A. Kim, M. Kim, and Y. Song 2018. A full RNS variant of approximate homomorphic encryption. In C. Cid and M. J. J. Jr. (Eds.), *Selected Areas in Cryptography - SAC 2018 - 25th International Conference, Calgary, AB, Canada, August 15-17, 2018, Revised Selected Papers*, Volume 11349 of *Lecture Notes in Computer Science*, pp. 347–368. Springer.

[15] Chillotti, I., N. Gama, M. Georgieva, and M. Izabachène 2016. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In J. H. Cheon and T. Takagi (Eds.), *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I*, Volume 10031 of *Lecture Notes in Computer Science*, pp. 3–33.

[16] Chillotti, I., M. Joye, and P. Paillier 2021. Programmable bootstrapping enables efficient homomorphic inference of deep neural networks. In S. Dolev, O. Margalit, B. Pinkas, and A. A. Schwarzmann (Eds.), *Cyber Security Cryptography and Machine Learning - 5th International Symposium, CSCML 2021, Be'er Sheva, Israel, July 8-9, 2021, Proceedings*, Volume 12716 of *Lecture Notes in Computer Science*, pp. 1–19. Springer.

[17] Cooley, J. and J. Tukey. 1965. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation* *19*(90): 297–301 .

[18] Crandall, R.E. 1999. Integer convolution via split-radix fast Galois transform.

[19] Crawford, J.L.H., C. Gentry, S. Halevi, D. Platt, and V. Shoup 2018. Doing real

work with FHE: the case of logistic regression. In M. Brenner and K. Rohloff (Eds.), *Proceedings of the 6th Workshop on Encrypted Computing & Applied Homomorphic Cryptography, WAHC@CCS 2018, Toronto, ON, Canada, October 19, 2018*, pp. 1–12. ACM.

[20] Dai, W., Y. Doröz, Y. Polyakov, K. Rohloff, H. Sajjadpour, E. Savaş, and B. Sunar. 2018. Implementation and evaluation of a lattice-based key-policy abe scheme. *IEEE Transactions on Information Forensics and Security 13*(5): 1169–1184. https://doi.org/10.1109/TIFS.2017.2779427 .

[21] Dai, W. and B. Sunar 2016. cuhe: A homomorphic encryption accelerator library. In E. Pasalic and L. R. Knudsen (Eds.), *Cryptography and Information Security in the Balkans*, Cham, pp. 169–186. Springer International Publishing.

[22] Fan, J. and F. Vercauteren. 2012. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch.* 2012: 144 .

[23] Feldmann, A., N. Samardzic, A. Krastev, S. Devadas, R. Dreslinski, K. Eldefrawy, N. Genise, C. Peikert, and D. Sánchez. 2021. F1: A fast and programmable accelerator for fully homomorphic encryption (extended version). *CoRR* abs/2109.05371. https://arxiv.org/abs/2109.05371 .

[24] Gamal, T.E. 1984. A public key cryptosystem and a signature scheme based on discrete logarithms. In *CRYPTO*, Volume 196 of *Lecture Notes in Computer Science*, pp. 10–18. Springer.

[25] Gentleman, W.M. and G. Sande 1966. Fast fourier transforms: For fun and profit. In *Proceedings of the November 7-10, 1966, Fall Joint Computer Conference*, AFIPS '66 (Fall), New York, NY, USA, pp. 563–578. Association for Computing Machinery.

[26] Gentry, C. 2009a. Fully homomorphic encryption using ideal lattices. In M. Mitzenmacher (Ed.), *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*, pp. 169–178. ACM.

[27] Gentry, C. 2009b. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, STOC '09, New York, NY, USA, pp. 169–178. Association for Computing Machinery.

[28] Govindaraju, N.K., B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli 2008. High performance discrete fourier transforms on graphics processors. In *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2008, November 15-21, 2008, Austin, Texas, USA*, pp. 2. IEEE/ACM.

[29] Han, K., S. Hong, J.H. Cheon, and D. Park. 2018. Efficient logistic regression on large encrypted data. *IACR Cryptol. ePrint Arch.*: 662 .

[30] Harvey, D. 2014. Faster arithmetic for number-theoretic transforms. *Journal of Symbolic Computation* 60: 113–119 .

[31] Jung, W., S. Kim, J.H. Ahn, J.H. Cheon, and Y. Lee. 2021. Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with gpus. Cryptology ePrint Archive, Report 2021/508. https://ia.cr/2021/508.

[32] Kim, S., W. Jung, J. Park, and J.H. Ahn 2020. Accelerating number theoretic transformations for bootstrappable homomorphic encryption on gpus. In *IISWC*, pp. 264–275. IEEE.

[33] LeCun, Y., C. Cortes, and C. Burges. 2010. Mnist handwritten digit database.

[34] López-Alt, A., E. Tromer, and V. Vaikuntanathan 2012. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In H. J. Karloff and T. Pitassi (Eds.), *Proceedings of the 44th Symposium on Theory of Computing Conference, STOC 2012, New York, NY, USA, May 19 - 22, 2012*, pp. 1219–1234. ACM.

[35] Lyubashevsky, V., D. Micciancio, C. Peikert, and A. Rosen 2008. SWIFFT: A Modest Proposal for FFT Hashing. In K. Nyberg (Ed.), *Fast Software Encryption*, Berlin, Heidelberg, pp. 54–72. Springer Berlin Heidelberg.

[36] NVIDIA. 2021. NVIDIA Nsight Systems. https://developer.nvidia.com/nsight-systems. Last accessed: 2021-10-13.

[37] Paillier, P. 1999. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, Volume 1592 of *Lecture Notes in Computer Science*, pp. 223–238. Springer.

[38] Paszke, A., S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. 2019. Pytorch: An imperative style, high-performance deep learning library, In *Advances in Neural Information Processing Systems 32*, eds. Wallach, H., H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, 8024–8035. Curran Associates, Inc.

[39] Pedregosa, F., G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. 2011. Scikit-learn: Machine learning in python. *Journal of machine learning research 12*(Oct): 2825–2830 .

[40] Pollard, J.M. 1971. The fast Fourier transform in a finite field. *Mathematics of Computation* 25: 365–374 .

[41] Rivest, R.L., L. Adleman, M.L. Dertouzos, et al. 1978. On data banks and privacy homomorphisms. *Foundations of secure computation 4*(11): 169–180 .

[42] Ruder, S. 2016. An overview of gradient descent optimization algorithms. *CoRR* abs/1609.04747. https://arxiv.org/abs/1609.04747 .