

GPU-Accelerated Tensor Operations for CKKS-Based CNN Inference on Homomorphically Encrypted Data

Núria Moreno-Chamorro

morenon@uji.es

Universitat Jaume I

Leonel Sousa

Instituto Superior Técnico

Aleksandar Ilić

Instituto Superior Técnico

Maribel Castillo

Universitat Jaume I

Manuel F. Dolz

Universitat Jaume I

Research Article

Keywords: Homomorphic encryption, CKKS, GPU acceleration, convolutional neural networks

Posted Date: October 13th, 2026

DOI: <https://doi.org/10.21203/rs.3.rs-11162036/v1>

License:   This work is licensed under a Creative Commons Attribution 4.0 International License.

[Read Full License](#)

Additional Declarations: No competing interests reported.

GPU-Accelerated Tensor Operations for CKKS-Based CNN Inference on Homomorphically Encrypted Data

Núria Moreno-Chamorro^{1*}, Leonel Sousa², Aleksandar Ilić²,
Maribel Castillo¹, Manuel F. Dolz¹

^{1*} Department of Computer Science and Engineering,
Universitat Jaume I, Castelló de la Plana, Spain.

² INESC-ID, Instituto Superior Técnico, Universidade de Lisboa,
Lisbon, Portugal.

*Corresponding author(s). E-mail(s): morenon@uji.es;

Abstract

Homomorphic encryption (HE) enables privacy-preserving inference over encrypted data, but its high computational cost limits practical deployment. While GPUs accelerate HE operations, integrating them into high-level tensor frameworks remains challenging, especially for compact and edge platforms. We present a GPU-accelerated framework for CKKS-based CNN inference that bridges high-level encrypted tensor operations and efficient GPU execution. The proposed approach extends Troy, a GPU-accelerated CKKS library, with dedicated CUDA kernels for convolution, pooling, square activation, fully connected layers, and other CKKS operations, integrating this backend into TenSEAL. Rather than executing tensor layers as sequences of individual ciphertext operations, the proposed implementation operates directly on the RNS-CKKS representation and exposes parallelism across encrypted tensor elements and their polynomial coefficients at the layer level. Synthetic benchmarks on the compact NVIDIA DGX Spark show speedups of up to **7.55×**, **18.12×**, and **6.33×** for convolution, pooling, and fully connected layers, respectively, over a 96-core Intel Xeon Sapphire Rapids server, and up to **258.30×**, **12.25×**, **87.09×**, and **16.41×** for convolution, pooling, fully connected, and square activation, respectively, over an operation-level GPU baseline on the same device. End-to-end evaluations with **CryptoNets** and **Mini-LoLa** on NVIDIA DGX Spark and the NVIDIA Jetson AGX Thor confirm that tensor-level execution on compact and edge GPUs enables efficient CKKS inference outperforming the evaluated 96-core server-class CPU system.

Keywords: Homomorphic encryption, CKKS, GPU acceleration, convolutional neural networks

1 Introduction

Privacy-preserving machine learning has emerged as one of the most prominent applications of Homomorphic Encryption (HE) [1], enabling Deep Neural Network (DNN) inference directly on encrypted data without requiring decryption during computation [2–4]. HE can be categorized as leveled or Fully Homomorphic Encryption (FHE), where FHE employs bootstrapping to enable computations of arbitrary multiplicative depth. Among the available HE schemes, Cheon-Kim-Kim-Song (CKKS) [5] is widely adopted for encrypted machine learning because it efficiently supports approximate arithmetic over real-valued data. Its security relies on the Ring Learning With Errors (RLWE) problem [6], while its efficient implementation is enabled through the Residue Number System (RNS) representation [7]. Nevertheless, performing inference on encrypted data remains significantly more complex than plaintext execution due to the polynomial representation of ciphertexts and the high computational cost of operations such as polynomial multiplication, key switching, and modular transformations. Consequently, improving the efficiency of homomorphic Neural Network (NN) inference remains one of the main challenges limiting the practical adoption of privacy-preserving deep learning. Surveys on HE identify computational efficiency, hardware acceleration, and optimized software frameworks as key research directions for making encrypted inference practical [8–12].

Although substantial progress has been made in both HE software and hardware acceleration, efficiently executing complete encrypted Convolutional Neural Networks (CNNs) on modern Graphics Processing Units (GPUs) remains challenging due to the heterogeneous computational characteristics of convolution, pooling, activation, and fully connected layers. Even though individual HE primitives have been successfully accelerated, their efficient integration into high-level machine learning frameworks for complete encrypted inference remains an open challenge [8, 10]. Bridging the gap between user-friendly encrypted machine learning frameworks and efficient GPU execution is therefore essential for making privacy-preserving deep learning practical in real-world applications. Moreover, the increasing computational capabilities of compact and edge GPU platforms raise the question of whether encrypted inference can be efficiently executed beyond traditional server-class systems, potentially broadening the range of platforms on which privacy-preserving inference can be deployed.

In this work, we present a GPU-accelerated framework for encrypted CNN execution that bridges high-level tensor abstractions and GPU-accelerated homomorphic computation. Building upon the tensor-based encrypted CNN operations introduced in our previous work [13] for multicore Central Processing Unit (CPU) execution, we extend their execution model to GPUs. Our implementation builds upon Troy [14], a Compute Unified Device Architecture (CUDA)-based HE library that provides GPU implementations of the CKKS scheme, which we extend and integrate into the TenSEAL [15] execution pipeline. Rather than executing encrypted tensor layers as sequences of individual ciphertext operations, the proposed implementation adopts a layer-level execution strategy that processes complete tensor layers directly on the GPU, exposing an additional dimension of tensor-level parallelism across encrypted tensor elements. Based on this execution strategy, we develop dedicated CUDA kernels covering both linear and nonlinear operators required to compose convolutional,

pooling, activation, and fully connected layers. These operators execute directly over CKKS ciphertexts while preserving the high-level programming interface provided by TenSEAL, enabling encrypted NNs to be efficiently executed on modern GPUs without requiring modifications to the application workflow. We evaluate the proposed framework on the compact NVIDIA DGX Spark and the edge NVIDIA Jetson AGX Thor, comparing their performance against a server-class 96-core Intel Xeon Sapphire Rapids system used as the CPU baseline. The main contributions of this work are summarized as follows:

- We design a layer-level GPU execution strategy for encrypted tensor operations that extends the operation-level execution model of the underlying HE backend by exposing parallelism across encrypted tensor elements and their polynomial coefficients. Based on this strategy, we redesign the tensor operations for GPU execution and implement dedicated CUDA kernels for convolution, pooling, square activation, and fully connected layers, together with the supporting CKKS operations required for their execution, including relinearization, Number Theoretic Transform (NTT)/Inverse Number Theoretic Transform (INTT), and rescaling.
- We extend the Troy HE library with these layer-level CUDA kernels and integrate the resulting backend into the TenSEAL execution pipeline, enabling transparent GPU execution of leveled CKKS while preserving the original high-level programming interface.
- We conduct a comprehensive performance evaluation campaign using synthetic benchmarks for each encrypted neural-network layer, comparing the proposed layer-level execution on the compact DGX against an operation-level GPU baseline on the same device and the 96-core Sapphire server-class system.
- We validate the proposed framework using the CryptoNets and Mini-LoLa architectures, evaluating complete encrypted inference on the compact DGX and edge Thor platforms against the 96-core Sapphire server-class system.
- We analyze the numerical error introduced by encrypted inference on the compact DGX and edge Thor platforms by comparing encrypted and plaintext outputs in terms of classification accuracy, mean absolute error (MAE), root mean squared error (RMSE), and maximum absolute error (MaxAE), assessing the impact of numerical deviations on the final predictions.

The remainder of this paper is organized as follows. Section 2 reviews related work on encrypted inference and GPU acceleration. Section 3 introduces the RNS-CKKS representation and describes the main homomorphic operations required for encrypted computation. Section 4 presents the proposed GPU-accelerated tensorial layers for homomorphic inference, including convolution, pooling, fully connected, and square activation layers. Section 5 evaluates their performance through synthetic layer benchmarks and complete encrypted inference using CryptoNets and Mini-LoLa. Finally, Section 6 summarizes the main conclusions and outlines future work.

2 Related Work

The application of HE to DNN inference has evolved considerably over the last decade. CryptoNets [16] was the first practical framework to demonstrate that NN could be evaluated directly over encrypted data, establishing the feasibility of privacy-preserving inference. Subsequent research focused on reducing the computational overhead introduced by homomorphic operations through improved ciphertext packing strategies, optimized convolution algorithms, and compiler-assisted execution. LoLa [17] significantly reduced the latency of encrypted NN inference by exploiting efficient ciphertext packing and data layout transformations. More recently, HyPHEN [18] introduced hybrid ciphertext packing and optimized convolution algorithms to reduce the number of expensive homomorphic operations, while UniHENN [19] eliminated the dependency on the *im2col* transformation, enabling more versatile and efficient encrypted CNN inference across different network architectures. In parallel, compiler-based approaches such as nGraph-HE [20], CHET [21], and EVA [22] automated the translation of deep learning models into homomorphic computations while introducing graph-level and circuit-level optimizations to improve execution efficiency. These works established the foundations for practical encrypted inference but remain primarily focused on CPU-based execution.

Complementing these DNN inference-oriented systems, several general-purpose HE libraries have been developed to support the implementation of encrypted applications. Microsoft Encrypted Arithmetic Library (SEAL) [23] provides one of the most mature implementations of the Brakerski-Fan-Vercautere (BFV), Brakerski-Gentry-Vaikuntanathan (BGV), and CKKS schemes and has become the reference library for both research and industrial applications. PALISADE [24] and its successor OpenFHE [25] provide comprehensive open-source frameworks supporting multiple HE schemes and advanced cryptographic primitives, while Lattigo [26] offers an efficient implementation of HE, also including Multiparty-HE-based secure protocols, in the Go programming language. Building upon SEAL, TenSEAL [15] introduces high-level tensor abstractions together with a Python interface specifically designed for encrypted machine learning, considerably simplifying the implementation of privacy-preserving NNs. Although these libraries provide robust software infrastructures for homomorphic computation, they primarily target CPU execution, making ciphertext arithmetic a significant bottleneck for large-scale NN inference.

To address this limitation, recent research has focused on exploiting hardware accelerators for HE. Early efforts such as cuHE [27] demonstrated that massively parallel GPUs can substantially reduce the execution time of homomorphic arithmetic by accelerating fundamental cryptographic operations. More recently, Zhai *et al.* [28] investigated the acceleration of encrypted computing on Intel GPUs, while hardware-oriented proposals such as HEAX [29] explored dedicated architectures for accelerating HE. Existing acceleration techniques primarily target the computational bottlenecks of HE, including the NTT, modular arithmetic, polynomial multiplication, key switching, and bootstrapping, as these kernels dominate the execution time of modern CKKS implementations [10]. Consequently, GPU-based acceleration has emerged as one of the most promising directions for improving the practicality of HE [8].

Representative GPU-enabled HE libraries include Phantom [30], Cheddar [31], HEonGPU [32], and Troy [14]. These libraries provide CUDA implementations of computationally intensive HE primitives and demonstrate substantial performance improvements over multicore CPU implementations, as confirmed by recent benchmarking studies [33]. However, these libraries primarily focus on the HE backend rather than integrating complete encrypted tensor layers into high-level machine-learning frameworks.

More recently, GPU acceleration has been extended beyond individual cryptographic kernels towards complete encrypted inference pipelines. TensorFHE [34] accelerates FHE primitives on GPUs and introduces operation-level batching to increase throughput by executing multiple homomorphic operations concurrently, while Pencil [35] focuses on privacy-preserving collaborative learning. Similarly, D’Agata *et al.* [36] investigate GPU acceleration for sparse fully homomorphic encrypted NNs to reduce the computational cost of encrypted inference. Zephyr [37] proposes a GPU-oriented framework for privacy-preserving transformer inference, introducing GPU-aware optimizations for homomorphic kernels, memory management, and execution scheduling. Although these studies demonstrate the benefits of combining GPU computing with HE, they focus on accelerating homomorphic operations or specific encrypted applications rather than integrating layer-level GPU execution into a high-level tensor-based inference framework.

Our work extends to the GPU the tensor-based inference framework introduced in our previous work [13], which extended TenSEAL with tensor-based CNN layers for multicore CPU execution. We integrate Troy as the GPU backend and further extend the framework with dedicated layer-level CUDA kernels. This enables complete HE inference pipelines to execute directly on the GPU, extending the execution granularity from individual homomorphic operations to complete tensor layers.

3 RNS-CKKS Scheme for GPU Execution

The CKKS HE scheme enables approximate arithmetic over encrypted real or complex values while preserving data confidentiality [5]. A subsequent work introduced a full RNS variant of CKKS [7], replacing large-integer arithmetic with independent modular computations over multiple residues while preserving the semantics of the original scheme. This representation has been adopted by modern libraries such as SEAL [23] and TenSEAL [15]. Moreover, the independence of the residue computations naturally exposes massive fine-grained parallelism, making the scheme particularly suitable for acceleration on GPUs.

The implementation of the encrypted CNN layers presented in this work relies on the arithmetic foundations of the RNS-CKKS scheme as implemented in the Troy [14] HE library. We therefore begin by introducing the internal representation of ciphertexts and their decomposition into RNS components, followed by the sequence of arithmetic transformations involved in homomorphic computation. These include coefficient-wise modular additions, polynomial multiplications accelerated by the NTT, and the post-processing operations required to preserve the ciphertext structure and encryption parameters, namely key switching, modulus switching, and

rescaling. Together, these arithmetic primitives constitute the computational basis of the algorithms presented in the remainder of the paper.

3.1 CKKS Ciphertext Representation

A CKKS ciphertext consists of two high-degree polynomials $ct = (c_0, c_1)$, defined over the polynomial ring

$$R_q = \mathbb{Z}_q[X]/(X^N + 1),$$

where N denotes the polynomial modulus degree and

$$q = \prod_{\ell=0}^{L-1} q_\ell$$

is the coefficient modulus, expressed as the product of L distinct primes q_ℓ in the full RNS variant of the CKKS scheme [7]. Each polynomial

$$c_j(X), \quad j \in \{0, 1\},$$

contains N coefficients. Instead of storing these coefficients modulo the large modulus q , the full RNS representation stores every coefficient independently modulo each coprime number q_ℓ . As a result, each polynomial is represented by L residue polynomials, one for each modulus.

Since a ciphertext consists of two polynomials, its internal representation contains $2 \times L$ residue polynomials. Each residue polynomial contains the same number of N coefficients, so every ciphertext is ultimately stored as $2 \times L \times N$ modular coefficients.

3.2 GPU Memory Layout and Parallelization

To maximize effective memory bandwidth, ciphertext coefficients are stored contiguously in GPU global memory. Considering M ciphertexts, indexed by $m \in \{0, \dots, M-1\}$, each ciphertext $ct_m = (c_{m,0}, c_{m,1})$ contains two polynomial components, each decomposed into L RNS polynomials of N coefficients. For each ciphertext, the RNS polynomials of $c_{m,0}$ are stored consecutively, followed by those of $c_{m,1}$, as illustrated in Fig. 1.

The hierarchical structure of the RNS-CKKS representation exposes four independent dimensions of parallelism that can be exploited individually or combined depending on the operation and its GPU mapping:

1. **Ciphertext-level parallelism**, where different ciphertexts are processed concurrently.
2. **Component-level parallelism**, where the ciphertext components c_0 and c_1 are processed independently.
3. **RNS-level parallelism**, where the L residue polynomials of each component are processed independently.
4. **Coefficient-level parallelism**, where the N polynomial coefficients are processed independently.

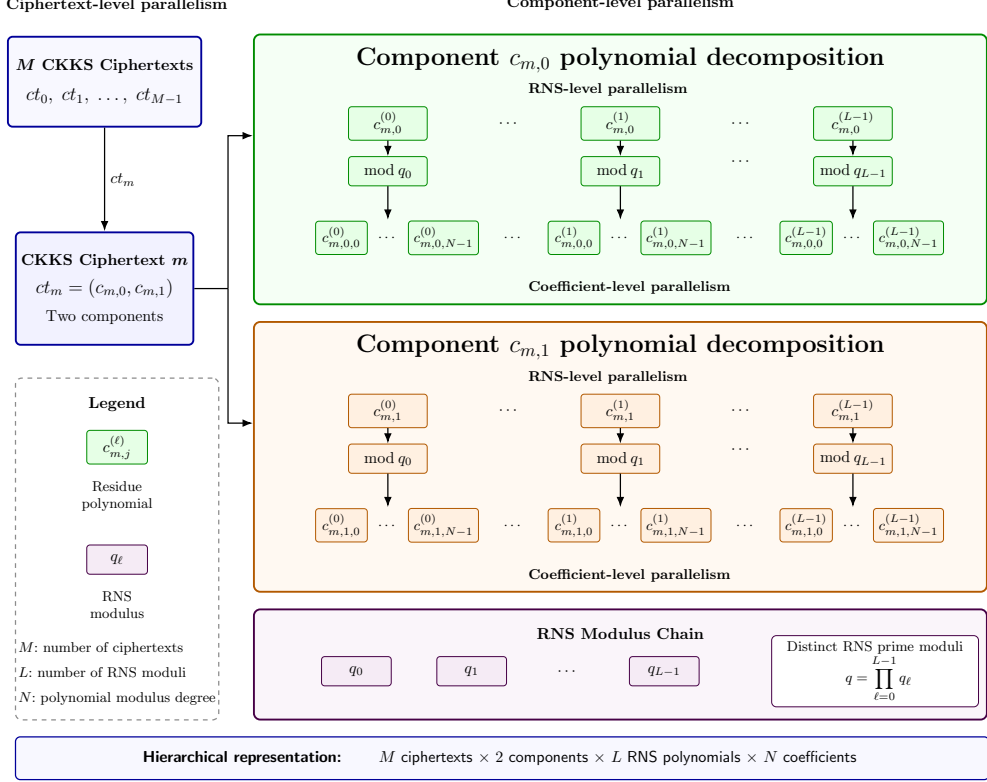


Fig. 1 Hierarchical memory organization of M CKKS ciphertexts under the full RNS representation. Each ciphertext $ct_m = (c_{m,0}, c_{m,1})$ contains two polynomial components, each decomposed into L RNS polynomials of N coefficients. The organization exposes parallelism across ciphertexts, components, RNS polynomials, and polynomial coefficients.

This hierarchical organization is preserved throughout all homomorphic operations, providing the necessary flexibility to target different granularities of execution depending on the computational workload. At the tensor-operation level, processing the ciphertexts associated with different tensor elements concurrently extends ciphertext-level parallelism into tensor-level parallelism, which forms the basis of the layer-level GPU execution strategy presented in Section 4.

3.3 RNS-CKKS Operations

The memory layout introduced in the previous subsection directly determines the GPU execution model adopted by homomorphic operations. Each ciphertext is stored as a collection of contiguous RNS polynomials, where every polynomial consists of N consecutive coefficients, enabling coalesced global-memory accesses,

$$c_j^{(\ell)} = (c_{j,0}^{(\ell)}, c_{j,1}^{(\ell)}, \dots, c_{j,N-1}^{(\ell)}),$$

with $j \in \{0, 1\}$ denoting the ciphertext component, $\ell \in \{0, \dots, L - 1\}$ the RNS modulus, and $n \in \{0, \dots, N - 1\}$ the coefficient index.

3.3.1 Encoding and Decoding

Before encryption, the input data are encoded as CKKS plaintexts. Vector encoding packs multiple values into a single plaintext by applying the inverse canonical embedding, efficiently implemented through an Inverse Fast Fourier Transform (IFFT), which maps the input vector to a polynomial whose evaluations at the primitive $2N$ -th roots of unity correspond to the plaintext slots. In contrast, scalar values are encoded as constant plaintexts by replicating the scalar throughout the plaintext slots. Once the plaintext polynomial has been obtained, its coefficients p_n , with $n \in \{0, \dots, N - 1\}$, are multiplied by the encoding scale Δ and rounded to the nearest integers,

$$\hat{p}_n = \lfloor \Delta p_n \rfloor.$$

Each coefficient is then represented independently under every modulus of the RNS base,

$$\hat{p}_n \longrightarrow (\hat{p}_n \bmod q_0, \hat{p}_n \bmod q_1, \dots, \hat{p}_n \bmod q_{L-1}),$$

producing the plaintext used by the subsequent homomorphic operations.

Decoding recovers the approximate values represented by a decrypted CKKS plaintext. The polynomial coefficients are first reconstructed from their RNS representation and scaled by Δ^{-1} to compensate for the scale introduced during encoding. The canonical embedding is then applied, efficiently implemented through a Fast Fourier Transform (FFT), to recover the approximate real or complex values stored in the plaintext slots.

3.3.2 Encryption and Decryption

Encryption transforms an encoded plaintext polynomial into a ciphertext using the secret key polynomial s . To introduce the randomness required for encryption, a randomized ciphertext encrypting zero, $\text{Enc}(0)$, is first generated. In the considered implementation, encoded plaintexts are represented in the NTT domain, so the encoded plaintext p is incorporated by adding it coefficient-wise to the first ciphertext component,

$$(c_0, c_1) = \text{Enc}(0) + (p, 0) = (\tilde{c}_0, \tilde{c}_1) + (p, 0) = (\tilde{c}_0 + p, \tilde{c}_1),$$

where p denotes the encoded plaintext polynomial. The resulting ciphertext therefore contains the encoded plaintext together with the randomness and noise introduced during the encryption of zero.

Decryption uses the secret key polynomial s to recover the encoded plaintext from a ciphertext $ct = (c_0, c_1)$. The decrypted polynomial is computed as

$$p' = c_0 + c_1 s \pmod{q},$$

where p' approximates the original encoded plaintext polynomial due to the noise introduced during encryption and accumulated through homomorphic computation. The resulting polynomial can subsequently be decoded to recover the approximate values stored in the CKKS slots.

3.3.3 Homomorphic Addition

Let $ct_A = (c_{0,0}, c_{0,1})$ and $ct_B = (c_{1,0}, c_{1,1})$ be two ciphertexts at the same modulus level and with compatible scales. Homomorphic addition produces the ciphertext $ct_{\text{sum}} = (c_{\text{sum},0}, c_{\text{sum},1})$. Under each RNS modulus, the corresponding coefficients of the two ciphertexts are added independently. Thus, for each component $j \in \{0, 1\}$,

$$c_{\text{sum},j,n}^{(\ell)} = c_{0,j,n}^{(\ell)} + c_{1,j,n}^{(\ell)} \pmod{q_\ell},$$

where $\ell \in \{0, \dots, L-1\}$ identifies the RNS modulus and $n \in \{0, \dots, N-1\}$ the polynomial coefficient.

Ciphertext–plaintext addition is performed similarly. Let $ct_0 = (c_{0,0}, c_{0,1})$ be a ciphertext and $pt_0 = p_0$ a plaintext. Since the plaintext is represented by a single polynomial, only the first ciphertext component is modified, while the second component remains unchanged. The resulting ciphertext $ct_{\text{sum}} = (c_{\text{sum},0}, c_{\text{sum},1})$ is given by

$$c_{\text{sum},0,n}^{(\ell)} = c_{0,0,n}^{(\ell)} + p_{0,n}^{(\ell)} \pmod{q_\ell}, \quad c_{\text{sum},1,n}^{(\ell)} = c_{0,1,n}^{(\ell)}.$$

3.3.4 Homomorphic Multiplication

Let $ct_A = (c_{0,0}, c_{0,1})$ and $ct_B = (c_{1,0}, c_{1,1})$ be two ciphertexts at the same modulus level and with compatible scales. In the NTT domain, homomorphic multiplication yields the three-component ciphertext

$$ct_{\text{mult}} = (d_0, d_1, d_2),$$

where the output coefficients are computed independently under each RNS modulus. Thus, for every $\ell \in \{0, \dots, L-1\}$ and $n \in \{0, \dots, N-1\}$,

$$\begin{aligned} d_{0,n}^{(\ell)} &= c_{0,0,n}^{(\ell)} c_{1,0,n}^{(\ell)} \pmod{q_\ell}, \\ d_{1,n}^{(\ell)} &= \left(c_{0,0,n}^{(\ell)} c_{1,1,n}^{(\ell)} + c_{0,1,n}^{(\ell)} c_{1,0,n}^{(\ell)} \right) \pmod{q_\ell}, \\ d_{2,n}^{(\ell)} &= c_{0,1,n}^{(\ell)} c_{1,1,n}^{(\ell)} \pmod{q_\ell}. \end{aligned}$$

To restore the standard two-component ciphertext representation, the contribution of the third component is incorporated into the first two components through the relinearization procedure described in the following subsection.

Ciphertext–plaintext multiplication follows the same principle. Let $ct_0 = (c_{0,0}, c_{0,1})$ be a ciphertext and $pt_0 = p_0$ a plaintext. Since a plaintext is represented by a single polynomial component, the resulting ciphertext preserves the standard two-component

representation,

$$ct_{\text{mult}} = (c_{\text{mult},0}, c_{\text{mult},1}),$$

whose coefficients are computed as

$$c_{\text{mult},j,n}^{(\ell)} = c_{0,j,n}^{(\ell)} p_{0,n}^{(\ell)} \pmod{q_\ell},$$

for $j \in \{0, 1\}$, $\ell \in \{0, \dots, L-1\}$, and $n \in \{0, \dots, N-1\}$.

3.3.5 Relinearization

A ciphertext multiplication produces an intermediate ciphertext

$$ct_{\text{mult}} = (d_0, d_1, d_2).$$

Keeping this three-component representation would introduce an additional ciphertext component after every homomorphic multiplication, dramatically increasing the cost of subsequent operations. Relinearization removes d_2 by transferring its contribution into (d_0, d_1) through a specific form of key switching.

The relinearization key RZ is stored in the NTT domain and follows the same two-component polynomial organization as a ciphertext, with components indexed by $j \in \{0, 1\}$. For relinearization, RZ contains one key element for each source modulus q_ℓ , with $\ell \in \{0, \dots, L-1\}$, and each key element is represented over the target moduli q_t , with $t \in \{0, \dots, L\}$, including the additional special modulus q_L . Thus, each component of RZ can be expressed as

$$RZ_j^{(t)} = [RZ_{0,j}^{(t)}, \dots, RZ_{L-1,j}^{(t)}], \quad j \in \{0, 1\}, \quad t \in \{0, \dots, L\},$$

where each $RZ_{\ell,j}^{(t)}$, with $\ell \in \{0, \dots, L-1\}$, is a polynomial with N coefficients.

For a given source modulus q_ℓ and target modulus q_t , the corresponding contribution of $d_2^{(\ell)}$ is converted to the target modulus q_t , yielding $D_\ell^{(t)}$. The corresponding coefficient-wise product is

$$D_{\ell,n}^{(t)} RZ_{\ell,j,n}^{(t)} \pmod{q_t}, \quad j \in \{0, 1\}, \quad \ell \in \{0, \dots, L-1\}, \quad t \in \{0, \dots, L\}.$$

The contributions from all source moduli are then accumulated. At coefficient level, the resulting correction is expressed as

$$\text{Relin}_{RZ,j,n}^{(t)}(d_2) = \sum_{\ell=0}^{L-1} D_{\ell,n}^{(t)} RZ_{\ell,j,n}^{(t)} \pmod{q_t}, \quad j \in \{0, 1\}, \quad t \in \{0, \dots, L\}.$$

The resulting values are represented over the extended RNS base of $L+1$ moduli. The special modulus q_L is then removed through modulus switching [7], returning the

corrections to the original L RNS moduli. Finally, these corrections are incorporated into d_0 and d_1 , yielding the relinearized ciphertext

$$(c'_0, c'_1) = (d_0, d_1) + \text{Relin}_{RZ}(d_2).$$

3.3.6 Number Theoretic Transform

For each ciphertext component and each RNS modulus, the NTT transforms a polynomial from the coefficient domain to the evaluation domain. The transform is computed through $\log_2 N$ butterfly layers, where N denotes the polynomial modulus degree. A butterfly is the basic operation of the NTT and combines two coefficients separated by a distance, or gap, within the same RNS polynomial. Different algorithmic strategies can be used to organize these butterfly operations, including the classical Cooley–Tukey algorithm [38], radix-4 approaches [39], and the Four-Step algorithm [40].

For the ciphertext $ct_0 = (c_{0,0}, c_{0,1})$, a butterfly operating on component j under modulus q_ℓ processes the coefficient pair

$$u = c_{0,j,n}^{(\ell)}, \quad v = c_{0,j,n+\text{gap}}^{(\ell)},$$

and updates them using the corresponding twiddle factor [41] $\omega_{\ell,r}$ as

$$c_{0,j,n}^{(\ell)} = u + \omega_{\ell,r} v \pmod{q_\ell}, \quad c_{0,j,n+\text{gap}}^{(\ell)} = u - \omega_{\ell,r} v \pmod{q_\ell}$$

where $j \in \{0, 1\}$ denotes the ciphertext component, $n \in \{0, \dots, N-1\}$ denotes the coefficient index within the RNS polynomial, and r denotes the index of the twiddle factor used at the current NTT layer.

Following the radix-2 Harvey decomposition adopted by modern HE libraries [42], the transform is recursively computed over groups of coefficients. Initially, the complete polynomial forms a single group of size N , containing $N/2$ independent butterflies with $\text{gap} = N/2$. After each layer, every group is divided into two smaller groups while the gap is halved. Consequently, the sequence of group sizes is

$$N, \frac{N}{2}, \frac{N}{4}, \dots, 2,$$

whereas the corresponding gap values are

$$\frac{N}{2}, \frac{N}{4}, \frac{N}{8}, \dots, 1.$$

The INTT follows an analogous butterfly structure, using the corresponding inverse twiddle factors and the normalization factor $N^{-1} \bmod q_\ell$, where N^{-1} denotes the multiplicative inverse of N modulo q_ℓ , to recover the coefficient-domain representation.

3.3.7 Modulus Switching and Rescaling

Modulus switching reduces the RNS base by removing one modulus, being used during relinearization to remove the special modulus q_L introduced by RZ . In contrast, rescaling is required after multiplication because, when both operands have scale Δ , the resulting scale becomes Δ^2 , removing the last modulus q_{L-1} from the ciphertext modulus chain to reduce the scale to Δ^2/q_{L-1} . Therefore, ciphertext–ciphertext multiplication requires relinearization followed by rescaling, whereas ciphertext–plaintext multiplication only requires rescaling.

- **Modulus switching:** Let the extended modulus used during relinearization be $q = \prod_{\ell=0}^L q_\ell$. Modulus switching removes the special modulus q_L , producing the reduced modulus $q' = \prod_{\ell=0}^{L-1} q_\ell$.
- **Rescaling:** Let the current ciphertext modulus after multiplication be $q = \prod_{\ell=0}^{L-1} q_\ell$. Rescaling removes q_{L-1} , producing the reduced modulus $q' = \prod_{\ell=0}^{L-2} q_\ell$.

The rescaling operation is performed using a divide-and-round procedure. Consider a coefficient $c_{j,n}$, represented by the residues $c_{j,n}^{(0)}, \dots, c_{j,n}^{(L-1)}$. The INTT is first applied to the polynomial associated with q_{L-1} to recover its coefficient-domain representation. These coefficients are then reduced modulo each remaining prime q_ℓ and transformed back to the NTT domain. The resulting residues $\tilde{c}_{j,n}^{(\ell)}$ are used to update each remaining polynomial as

$$\tilde{c}_{j,n}^{(\ell)} = \left(c_{j,n}^{(\ell)} - \hat{c}_{j,n}^{(\ell)} \right) q_{L-1}^{-1} \pmod{q_\ell}, \quad \ell \in \{0, \dots, L-2\}.$$

Finally, the residues associated with q_{L-1} are discarded, leaving the ciphertext represented over the reduced modulus chain q' .

4 GPU-Accelerated Tensor Layers for HE

The homomorphic operations described in the previous section provide the computational building blocks required to evaluate CNNs over encrypted data. Building upon these primitives, we implement the main tensor operations required for encrypted inference, namely convolution, pooling, fully connected, and square activation layers. The tensor-based implementation of these operations was originally introduced in our previous work [13] for multicore CPU execution, where parallelism is primarily exploited across ciphertexts by distributing independent output ciphertexts among the available cores.

The tensor representation also supports batched inputs by exploiting the packing capability of CKKS. During encoding, multiple values are packed into the slots of a single plaintext using vector encoding, which is subsequently encrypted into a ciphertext. Homomorphic operations applied to the ciphertext are then evaluated simultaneously over these slots, providing an intrinsic form of Single Instruction Multiple Data (SIMD) parallelism. In our tensor representation, this property is used to pack the same tensor coordinate from different input samples into the slots of a single plaintext.

Let $\mathcal{I}$ denote an input tensor and B the batch size. For each input tensor coordinate (c_i, h_i, w_i) , the values $\mathcal{I}^{(b)} c_i, h_i, w_i$ corresponding to the same coordinate across the

B input samples are first encoded into a plaintext and then encrypted into a single CKKS ciphertext,

$$EI_{c_i, h_i, w_i} = \text{Enc} \left(\text{Encode} \left(\mathcal{I}^{(0)} c_i, h_i, w_i, \mathcal{I}^{(1)} c_i, h_i, w_i, \dots, \mathcal{I}^{(B-1)} c_i, h_i, w_i \right) \right).$$

Each slot of EI_{c_i, h_i, w_i} therefore contains the value at coordinate (c_i, h_i, w_i) from a different input sample. Consequently, a single homomorphic operation on this ciphertext performs the corresponding computation on that tensor coordinate for all B samples simultaneously. Thus, the complete batch is processed in parallel through the CKKS slots.

In this work, we extend the TenSEAL tensor operations for GPU execution and enhance the underlying Troy backend with dedicated CUDA kernels operating directly on the RNS-CKKS representation. This GPU parallelism is complementary to the slot-level SIMD parallelism described above. Whereas CKKS processes the B packed samples simultaneously within each ciphertext, the proposed layer-level GPU execution strategy exposes tensor-level parallelism across encrypted tensor elements, while also exploiting coefficient-level parallelism within their polynomial representations. For the layer-level kernels, the proposed GPU mapping exploits the independence of the output-coefficient computations by assigning each output coefficient to an independent CUDA thread, which accumulates the contributions required to compute its assigned coefficient without requiring synchronization with other threads. The kernels involved in relinearization also exploit coefficient-level parallelism, except for the NTT/INTT kernels, which use a cooperative strategy in which the threads within a CUDA block jointly compute the transformation. This allows intermediate results to remain in shared memory and avoids unnecessary writes to global memory. The following subsections describe these mappings in detail.

4.1 Convolution

Let EI denote an encrypted input tensor with C_i input channels and spatial dimensions $H_i \times W_i$, where each element of EI is a CKKS ciphertext $ct = (c_0, c_1)$. The convolutional layer is defined by plaintext weights PW with C_o output channels and filter dimensions $H_f \times W_f$, and a plaintext bias PB for each output channel, in which each weight PW_{c_o, c_i, h_f, w_f} and bias PB_{c_o} is encoded as a CKKS plaintext consisting of a single polynomial.

The encrypted output tensor EO consists of C_o output channels with spatial dimensions $H_o \times W_o$. Since the ciphertexts are represented in the NTT domain, the convolution is evaluated coefficient-wise for each coefficient n of each RNS modulus q_ℓ . The two ciphertext components of each output element are therefore computed as

$$EO_{c_0}^{(\ell)}[c_o, h_o, w_o, n] = \left(\sum_{c_i=0}^{C_i-1} \sum_{h_f=0}^{H_f-1} \sum_{w_f=0}^{W_f-1} EI_{c_0}^{(\ell)}[c_i, h_i, w_i, n] PW_{c_o, c_i, h_f, w_f, n}^{(\ell)} + PB_{c_o, n}^{(\ell)} \right) \pmod{q_\ell},$$

$$EO_{c_1}^{(\ell)}[c_o, h_o, w_o, n] = \left(\sum_{c_i=0}^{C_i-1} \sum_{h_f=0}^{H_f-1} \sum_{w_f=0}^{W_f-1} EI_{c_1}^{(\ell)}[c_i, h_i, w_i, n] PW_{c_o, c_i, h_f, w_f, n}^{(\ell)} \right) \pmod{q_\ell}.$$

Algorithm 1: GPU implementation of the convolution.

Input: Encrypted input EI , plaintext weights PW , plaintext biases PB
Output: Encrypted output EO , initialized to plaintext zero

```

1  _global_ ConvKernel one thread per output coefficient  $(c_o, h_o, w_o, n)$ ,
    $0 \leq c_o < C_o, 0 \leq h_o < H_o, 0 \leq w_o < W_o, 0 \leq n < N$  :
2      for  $c_i \leftarrow 0$  to  $C_i - 1$  do
3          for  $h_f \leftarrow 0$  to  $H_f - 1$  do
4              for  $w_f \leftarrow 0$  to  $W_f - 1$  do
5                   $h_i \leftarrow h_o s + h_f - P$ ;
6                   $w_i \leftarrow w_o s + w_f - P$ ;
7                  if  $0 \leq h_i < H_i$  and  $0 \leq w_i < W_i$  then
8                      for  $\ell \leftarrow 0$  to  $L - 1$  do
9                           $pw \leftarrow PW_{c_o, c_i, h_f, w_f, n}^{(\ell)}$ ;
10                          $r_0 \leftarrow \text{BarrettReduce} \left( EI_{c_0}^{(\ell)}[c_i, h_i, w_i, n] \cdot pw, q_\ell \right)$ ;
11                          $EO_{c_0}^{(\ell)}[c_o, h_o, w_o, n] \leftarrow \left( EO_{c_0}^{(\ell)}[c_o, h_o, w_o, n] + r_0 \right) \bmod q_\ell$ ;
12                          $r_1 \leftarrow \text{BarrettReduce} \left( EI_{c_1}^{(\ell)}[c_i, h_i, w_i, n] \cdot pw, q_\ell \right)$ ;
13                          $EO_{c_1}^{(\ell)}[c_o, h_o, w_o, n] \leftarrow \left( EO_{c_1}^{(\ell)}[c_o, h_o, w_o, n] + r_1 \right) \bmod q_\ell$ ;
14     for  $\ell \leftarrow 0$  to  $L - 1$  do
15          $EO_{c_0}^{(\ell)}[c_o, h_o, w_o, n] \leftarrow \left( EO_{c_0}^{(\ell)}[c_o, h_o, w_o, n] + PB_{c_o, n}^{(\ell)} \right) \bmod q_\ell$ ;

```

Here, $\ell \in \{0, \dots, L - 1\}$ denotes the RNS modulus index and $n \in \{0, \dots, N - 1\}$ the polynomial coefficient. For each output position (h_o, w_o) , the filter coordinates $h_f \in \{0, \dots, H_f - 1\}$ and $w_f \in \{0, \dots, W_f - 1\}$ determine the corresponding input position as $h_i = h_o s + h_f - P$ and $w_i = w_o s + w_f - P$, where s and P denote the stride and padding, respectively. Input positions outside the $H_i \times W_i$ boundaries are treated as zero and are therefore omitted from the computation. Finally, note that the bias $PB_{c_o, n}^{(\ell)}$ is added only to the c_0 component of the output ciphertext.

The proposed GPU implementation assigns each CUDA thread to a polynomial coefficient n of a specific output position (c_o, h_o, w_o) in EO . Considering all output channels and spatial positions, the convolution is therefore performed by $C_o H_o W_o N$ threads in parallel, with each thread processing its assigned coefficient across the L RNS moduli.

As shown in Algorithm 1, each thread traverses the C_i input channels and the $H_f \times W_f$ filter positions (Lines 2–4), computes the corresponding input coordinates (h_i, w_i) , and skips positions outside the input boundaries. For each valid position, the thread iterates over the L RNS moduli and processes its assigned coefficient n independently for each q_ℓ . The corresponding plaintext weight is reused for both ciphertext components, c_0 and c_1 (Lines 8–13). The product between each ciphertext coefficient and the plaintext weight is computed and reduced modulo q_ℓ using Barrett reduction [43], and then accumulated into the corresponding output coefficient, followed by the final modular reduction. Finally, the plaintext bias is added only to the c_0 component for each RNS modulus (Lines 14–15).

4.2 Pooling

The pooling layer follows the same GPU mapping as the convolution, assigning each thread to a polynomial coefficient n of a specific output position (c_o, h_o, w_o) in EO . Pooling processes each input channel independently, preserving it as the corresponding output channel. Therefore, the number of output channels is equal to the number of input channels, i.e., $C_o = C_i$. Each thread traverses the $H_f \times W_f$ pooling window associated with (h_o, w_o) and, for all L RNS moduli, accumulates the corresponding encrypted input coefficients for both ciphertext components c_0 and c_1 . Thus, compared with Algorithm 1, the plaintext multiplication in Lines 10 and 13 is omitted and only the corresponding addition is performed. Moreover, Lines 14–15 are not required, since there is no bias to add. Positions that fall outside the input boundaries due to padding are skipped following the same strategy as in the convolution. Rather than computing the average by dividing the accumulated value by the number of elements in the pooling window, we retain the unscaled sum. This avoids an additional ciphertext–plaintext multiplication and the subsequent rescaling operation, thereby preserving the same level of multiplicative depth. Consequently, the pooling kernel only requires modular additions.

4.3 Fully Connected

Let EI denote an encrypted input matrix of dimensions $M \times I$. The fully connected layer is defined by a plaintext weight matrix PW of dimensions $I \times K$, together with a plaintext bias vector PB of length K . The result is an encrypted output matrix EO of dimensions $M \times K$. Each element of EI is a CKKS ciphertext $ct = (c_0, c_1)$, whereas each weight $PW_{i,k}$ and bias PB_k is encoded as a CKKS plaintext consisting of a single polynomial.

Following the RNS-CKKS representation introduced in the previous section, the operands are represented in the NTT domain and the fully connected layer is performed coefficient-wise for each RNS modulus q_ℓ . Thus, the two ciphertext components of EO are computed as

$$\begin{aligned} EO_{c_0}^{(\ell)}[m, k, n] &= \left(\sum_{i=0}^{I-1} EI_{c_0}^{(\ell)}[m, i, n] PW_{i,k,n}^{(\ell)} + PB_{k,n}^{(\ell)} \right) \pmod{q_\ell}, \\ EO_{c_1}^{(\ell)}[m, k, n] &= \left(\sum_{i=0}^{I-1} EI_{c_1}^{(\ell)}[m, i, n] PW_{i,k,n}^{(\ell)} \right) \pmod{q_\ell}. \end{aligned}$$

Here, $m \in \{0, \dots, M-1\}$ and $k \in \{0, \dots, K-1\}$ identify the row and column of the output matrix, respectively, while $i \in \{0, \dots, I-1\}$ traverses the inner dimension of the matrix multiplication. Moreover, $\ell \in \{0, \dots, L-1\}$ denotes the RNS modulus index and $n \in \{0, \dots, N-1\}$ the polynomial coefficient. As shown in the equations, note that the bias term $PB_{k,n}^{(\ell)}$ is added only to the c_0 component of the output.

Algorithm 2: GPU implementation of the fully connected layer.

Input: Encrypted input EI , plaintext weights PW , plaintext biases PB
Output: Encrypted output EO , initialized to plaintext zero
1 **_global_** FCKernel *one thread per output coefficient* (m, k, n) ,
 $0 \leq m < M, 0 \leq k < K, 0 \leq n < N$:
2 **for** $i \leftarrow 0$ **to** $I - 1$ **do**
3 **for** $\ell \leftarrow 0$ **to** $L - 1$ **do**
4 $pw \leftarrow PW_{i,k,n}^{(\ell)}$;
5 $r_0 \leftarrow \text{BarrettReduce} \left(EI_{c_0}^{(\ell)}[m, i, n] \cdot pw, q_\ell \right)$;
6 $EO_{c_0}^{(\ell)}[m, k, n] \leftarrow \left(EO_{c_0}^{(\ell)}[m, k, n] + r_0 \right) \bmod q_\ell$;
7 $r_1 \leftarrow \text{BarrettReduce} \left(EI_{c_1}^{(\ell)}[m, i, n] \cdot pw, q_\ell \right)$;
8 $EO_{c_1}^{(\ell)}[m, k, n] \leftarrow \left(EO_{c_1}^{(\ell)}[m, k, n] + r_1 \right) \bmod q_\ell$;
9 **for** $\ell \leftarrow 0$ **to** $L - 1$ **do**
10 $EO_{c_0}^{(\ell)}[m, k, n] \leftarrow \left(EO_{c_0}^{(\ell)}[m, k, n] + PB_{k,n}^{(\ell)} \right) \bmod q_\ell$;

The proposed GPU implementation assigns each CUDA thread to a polynomial coefficient n of a specific output position (m, k) in EO . Considering the $M \times K$ ciphertexts of the output matrix, the fully connected layer is therefore performed by NMK threads in parallel.

As shown in Algorithm 2, each thread traverses the I input ciphertexts required to compute its corresponding output coefficient n (Line 2). For a given input ciphertext, coefficient n is processed independently for all L RNS moduli. Within each modulus q_ℓ , the corresponding plaintext weight is loaded and reused for both ciphertext components, c_0 and c_1 (Lines 3–8). Ciphertext coefficients are multiplied by their corresponding plaintext weights using Barrett reduction [43], and the resulting values are then accumulated into the corresponding output coefficient modulo q_ℓ . Once all I input ciphertexts have been processed, the plaintext bias associated with the output column is added only to the c_0 component for each of the L RNS moduli (Lines 9–10).

4.4 Square Activation

The square activation provides a simple polynomial activation function suitable for HE neural-network inference. Conventional activation functions such as ReLU cannot be directly evaluated over encrypted data using the arithmetic operations supported by CKKS. Therefore, they are commonly replaced by polynomial functions that can be expressed using additions and multiplications. Among these alternatives, the square function, $f(x) = x^2$, is particularly convenient because it has degree two and can be evaluated with a single ciphertext–ciphertext multiplication, limiting the multiplicative depth and the associated noise growth. For this reason, square activations are employed in the CryptoNets and Mini-LoLa architectures considered in this work.

Algorithm 3: GPU implementation of the square activation.

Input: Encrypted input EI
Output: Encrypted output EO , auxiliary buffers T_{NTT} and T_{INTT}
1 **__global__ SquareKernel one thread per ciphertext coefficient** (m, n) ,
 $0 \leq m < M, 0 \leq n < N$:
2 **for** $\ell \leftarrow 0$ **to** $L - 1$ **do**
3 $T_{\text{NTT}}^{(\ell)}[m, n] \leftarrow \text{BarrettReduce} \left(EI_{c_1}^{(\ell)}[m, n] \cdot EI_{c_1}^{(\ell)}[m, n], q_\ell \right)$;
4 $T_{\text{INTT}}^{(\ell)}[m, n] \leftarrow T_{\text{NTT}}^{(\ell)}[m, n]$;
5 $r \leftarrow \text{BarrettReduce} \left(EI_{c_0}^{(\ell)}[m, n] \cdot EI_{c_1}^{(\ell)}[m, n], q_\ell \right)$;
6 $EO_{c_1}^{(\ell)}[m, n] \leftarrow 2r \bmod q_\ell$;
7 $EO_{c_0}^{(\ell)}[m, n] \leftarrow \text{BarrettReduce} \left(EI_{c_0}^{(\ell)}[m, n] \cdot EI_{c_0}^{(\ell)}[m, n], q_\ell \right)$;

Given an input ciphertext $ct = (c_0, c_1)$, the square operation corresponds to a ciphertext–ciphertext multiplication and produces a three-component ciphertext as

$$ct^2 = (d_0, d_1, d_2) = (c_0^2, 2c_0c_1, c_1^2).$$

In contrast to the ciphertext–plaintext multiplications employed in the convolutional and fully connected layers, this operation involves products between the two ciphertext components. Computing each coefficient of the squared ciphertext therefore requires the evaluation of the three component products shown above. The resulting ciphertext should subsequently be relinearized to reduce its size from three components to two and rescaled to the next level of the modulus chain.

Algorithm 3 shows the proposed GPU implementation of the square operation over the encrypted input EI . Following the parallelization strategy adopted for the previous layers, one CUDA thread is assigned to each pair (m, n) , where m identifies an input ciphertext in EI and n a polynomial coefficient (Line 1). Each thread processes coefficient n independently for all L RNS moduli (Line 2), computing d_0 , d_1 , and d_2 modulo the corresponding modulus q_ℓ (Lines 3–7). In this way, the N coefficients of multiple ciphertexts are processed concurrently on the GPU.

The d_0 and d_1 components are directly stored as the two components of EO (Lines 5–7), while d_2 , which is required by the subsequent relinearization, is stored across two sections of an auxiliary buffer, T_{NTT} and T_{INTT} (Lines 3–4). Both sections initially contain the same coefficients of d_2 , but are maintained separately to support the subsequent transformations required during relinearization. This organization introduces an additional write during the square operation, but avoids an additional copy of the complete third polynomial before relinearization.

4.4.1 Relinearization

The ciphertext–ciphertext multiplication performed by the square activation increases the ciphertext size from two to three components. For each ciphertext m , the square operation produces

$$ct_m^2 = (d_{m,0}, d_{m,1}, d_{m,2}).$$

The components $d_{m,0}$ and $d_{m,1}$ are stored in EO , while $d_{m,2}$ is maintained separately for relinearization. The third component cannot be directly discarded, since it contains a contribution associated with the square of the secret key. Relinearization transforms the three-component ciphertext back into the standard two-component representation by applying a key-switching procedure with a relinearization key RZ . In this way, the contribution of $d_{m,2}$ is converted into two correction polynomials that are incorporated into $d_{m,0}$ and $d_{m,1}$ without decrypting the ciphertext.

Algorithm 4 summarizes the proposed GPU implementation for the M ciphertexts. The relinearization procedure is decomposed into sequential stages, since each stage operates on the intermediate representation produced by the preceding one. Individual stages are implemented by one or more CUDA kernels that process the corresponding ciphertexts, ciphertext components, RNS moduli, and polynomial coefficients in parallel. The relinearization key (RZ) contains L key elements, one per source RNS modulus q_ℓ . Key elements contain two polynomial components, indexed by $j \in \{0, 1\}$, represented over $L + 1$ target moduli q_t , with $t \in \{0, \dots, L\}$, where q_L is the special modulus used during key switching. Finally, each $RZ_{\ell,j}^{(t)}$ is a polynomial with N coefficients.

First, $d_{m,2}$ is transformed to the coefficient domain using the INTT for each source modulus (Lines 1–2), allowing its coefficients to be reduced modulo each target modulus q_t . The resulting coefficients are then transformed back to the NTT domain (Lines 3–5), producing $D_{m,\ell}^{(t)}$. This representation matches the relinearization key and enables coefficient-wise polynomial multiplication. For each relinearization-key component j , the transformed polynomials are multiplied by the corresponding key polynomials $RZ_{\ell,j}^{(t)}$ and accumulated over the L source moduli (Lines 6–7). The resulting $\text{Partial}_{m,j}^{(t)}$ is the accumulated contribution of the L source moduli for component j under target modulus q_t .

After processing all target moduli, $\text{Partial}_{m,j}^{(L)}$ associated with the special modulus q_L is transformed to the coefficient domain (Lines 8–9). The `CalculateCorrection` operation then computes, within each thread, the modulus-switching correction for each of the L source moduli from these coefficients (Lines 10–12). The resulting correction polynomials are transformed back to the NTT domain (Lines 13–14) and combined with $\text{Partial}_{m,j}^{(\ell)}$ and $d_{m,j}^{(\ell)}$ for each source modulus q_ℓ (Lines 15–17), producing the two components of the relinearized ciphertext.

4.4.2 NTT Implementation

Relinearization requires multiple transformations between the coefficient and NTT domains, making the efficiency of the NTT and INTT critical to the overall performance of the key-switching procedure. Since both transformations share a similar computational structure, this section describes the proposed GPU implementation using the NTT as reference.

The NTT provides parallelism across the same dimensions exploited by the previously proposed tensor kernels. However, the thread-per-coefficient mapping adopted by those kernels is not suitable for the NTT, as the dependencies between consecutive

Algorithm 4: GPU implementation of relinearization.

Input: M squared ciphertexts $(d_{m,0}, d_{m,1}, d_{m,2})$, relinearization key (RZ)
Output: M relinearized ciphertexts $(c'_{m,0}, c'_{m,1})$

- 1 **GPUKernel one block per** (m, ℓ) , $0 \leq m < M$, $0 \leq \ell < L$:
 - 2 $\text{Coeff}_m^{(\ell)} \leftarrow \text{INTT}(d_{m,2}^{(\ell)})$;
- 3 **for** $t \leftarrow 0$ **to** L **do**
 - 4 **GPUKernel one block per** (m, ℓ) , $0 \leq m < M$, $0 \leq \ell < L$:
 - 5 $D_{m,\ell}^{(t)} \leftarrow \text{NTT}(\text{Coeff}_m^{(\ell)} \bmod q_t)$;
 - 6 **GPUKernel one thread per coefficient** (m, j, n) ,
 $0 \leq m < M$, $0 \leq j < 2$, $0 \leq n < N$:
 - 7 $\text{Partial}_{m,j,n}^{(t)} \leftarrow \sum_{\ell=0}^{L-1} D_{m,\ell,n}^{(t)} RZ_{\ell,j,n}^{(t)} \bmod q_t$;
 - 8 **GPUKernel one block per** (m, j) , $0 \leq m < M$, $0 \leq j < 2$:
 - 9 $\text{Partial}_{m,j}^{(L)} \leftarrow \text{INTT}(\text{Partial}_{m,j}^{(L)})$;
 - 10 **GPUKernel one thread per coefficient** (m, j, n) ,
 $0 \leq m < M$, $0 \leq j < 2$, $0 \leq n < N$:
 - 11 **for** $\ell \leftarrow 0$ **to** $L-1$ **do**
 - 12 $\text{Correct}_{m,j,n}^{(\ell)} \leftarrow \text{CalculateCorrection}(\text{Partial}_{m,j,n}^{(L)}, q_\ell)$;
 - 13 **GPUKernel one block per** (m, j, ℓ) , $0 \leq m < M$, $0 \leq j < 2$, $0 \leq \ell < L$:
 - 14 $\text{Correct}_{m,j}^{(\ell)} \leftarrow \text{NTT}(\text{Correct}_{m,j}^{(\ell)})$;
 - 15 **GPUKernel one thread per coefficient** (m, j, n) ,
 $0 \leq m < M$, $0 \leq j < 2$, $0 \leq n < N$:
 - 16 **for** $\ell \leftarrow 0$ **to** $L-1$ **do**
 - 17 $c'_{m,j,n}^{(\ell)} \leftarrow \text{ApplyCorrection}(d_{m,j,n}^{(\ell)}, \text{Partial}_{m,j,n}^{(\ell)}, \text{Correct}_{m,j,n}^{(\ell)})$;

butterfly stages would require a separate kernel launch for each stage, with intermediate results written to and read from global memory between launches. To avoid this overhead, we adopt a block-level strategy in which each independent RNS polynomial is assigned to a CUDA block. The threads within the block cooperatively execute the butterfly operations while keeping the polynomial in shared memory throughout the transformation, thereby avoiding global-memory accesses for intermediate results.

For M ciphertexts and L RNS moduli, Algorithm 5 maps the resulting $2ML$ independent RNS polynomials to $2ML$ CUDA blocks. Each block processes one polynomial $EI_{m,j}^{(\ell)}$, where m identifies the ciphertext, $j \in \{0, 1\}$ its component (c_0 or c_1), and $\ell \in \{0, \dots, L-1\}$ the RNS modulus. For the polynomial sizes considered in our experimental evaluation, the N coefficients fit within the shared-memory capacity of a block and are cooperatively loaded and kept there throughout the transformation (Lines 1–3).

Thread blocks perform all $\log_2 N$ layers of the NTT (Lines 4–17). Within a given layer, the $N/2$ independent butterfly operations are distributed among the threads of the block. Threads start from the butterfly identified by their thread index and process additional butterflies using a stride equal to the block size until all $N/2$ butterflies are computed (Lines 6–16). The gap between the two coefficients of each butterfly

Algorithm 5: GPU implementation of the NTT.

Input: Encrypted input EI , NTT tables T
Output: NTT-transformed encrypted input EI_{NTT}

```

1  _global_ NTTKernel one block per  $(m, j, \ell)$ ,  $0 \leq m < M$ ,  $0 \leq j < 2$ ,  $0 \leq \ell < L$  :
2    Shared  $\leftarrow EI_{m,j}^{(\ell)}$ ;
3    SyncThreads();
4    for layer  $\leftarrow 0$  to  $\log_2 N - 1$  do
5      gap  $\leftarrow 2^{\log_2 N - \text{layer} - 1}$ ;
6      butterfly  $\leftarrow \text{threadIdx.x}$ ;
7      while butterfly  $< N/2$  do
8        group  $\leftarrow \lfloor \text{butterfly} / \text{gap} \rfloor$ ;
9         $n' \leftarrow \text{group}(2 \text{ gap}) + (\text{butterfly} \bmod \text{gap})$ ;
10        $r \leftarrow 2^{\text{layer}} + \text{group}$ ;
11        $u \leftarrow \text{Shared}[n']$ ;
12        $v \leftarrow \text{Shared}[n' + \text{gap}]$ ;
13        $t \leftarrow \text{MulMod}(v, T_\ell[r], q_\ell)$ ;
14        $\text{Shared}[n'] \leftarrow (u + t) \bmod q_\ell$ ;
15        $\text{Shared}[n' + \text{gap}] \leftarrow (u - t) \bmod q_\ell$ ;
16       butterfly  $\leftarrow \text{butterfly} + \text{blockDim.x}$ ;
17     SyncThreads();
18    $EI_{\text{NTT},m,j}^{(\ell)} \leftarrow \text{Shared}$ ;

```

decreases from $N/2$ to 1 across the layers (Line 5). For a given butterfly, its two coefficient positions are determined from the butterfly index and the current gap (Lines 8–9), and the corresponding precomputed twiddle factor is selected from the NTT table T_ℓ (Lines 10–13). The butterfly outputs are stored back in shared memory (Lines 14–15). Once all butterflies of a layer are completed, threads are synchronized before proceeding to the next layer (Line 17).

Finally, the transformed polynomial $EI_{\text{NTT},m,j}^{(\ell)}$ is written back to global memory (Line 18). Keeping the polynomial in shared memory throughout all $\log_2 N$ layers avoids writing and reading the partial results generated between consecutive layers to and from global memory, as well as additional kernel launches. The INTT follows the same GPU mapping, using the corresponding inverse butterfly ordering and inverse twiddle factors, together with the normalization factor $N^{-1} \bmod q_\ell$.

4.5 Rescaling

Rescaling is applied after ciphertext multiplication to reduce the ciphertext scale and remove the last modulus q_{L-1} from the current RNS base $\{q_0, \dots, q_{L-1}\}$. Thus, a ciphertext represented over L moduli is reduced to the $L - 1$ remaining moduli $\{q_0, \dots, q_{L-2}\}$.

Algorithm 6 summarizes the proposed GPU implementation. First, an INTT is applied to the polynomial associated with the modulus to be removed, q_{L-1} , to obtain its coefficients (Lines 1–2). The first coefficient-wise kernel (Lines 3–6) takes these coefficients and represents them under each of the remaining RNS moduli $\{q_0, \dots, q_{L-2}\}$,

Algorithm 6: GPU implementation of rescaling.

Input: M ciphertexts $(c_{m,0}, c_{m,1})$ represented over L RNS moduli
Output: M rescaled ciphertexts $(\tilde{c}_{m,0}, \tilde{c}_{m,1})$ over $L - 1$ RNS moduli

```
1 GPUKernel one block per  $(m, j), 0 \leq m < M, 0 \leq j < 2$  :  
2   Coeff $_{m,j} \leftarrow \text{INTT} \left( c_{m,j}^{(L-1)} \right)$ ;  
3 GPUKernel one thread per  $(m, n), 0 \leq m < M, 0 \leq n < N$  :  
4   for  $j \leftarrow 0$  to 1 do  
5     for  $\ell \leftarrow 0$  to  $L - 2$  do  
6        $\hat{c}_{m,j,n}^{(\ell)} \leftarrow \text{Coeff}_{m,j,n} \bmod q_\ell$ ;  
7 GPUKernel one block per  $(m, j, \ell), 0 \leq m < M, 0 \leq j < 2, 0 \leq \ell < L - 1$  :  
8    $\hat{c}_{m,j}^{(\ell)} \leftarrow \text{NTT} \left( \hat{c}_{m,j}^{(\ell)} \right)$ ;  
9 GPUKernel one thread per  $(m, n), 0 \leq m < M, 0 \leq n < N$  :  
10  for  $j \leftarrow 0$  to 1 do  
11    for  $\ell \leftarrow 0$  to  $L - 2$  do  
12       $\tilde{c}_{m,j,n}^{(\ell)} \leftarrow \left( c_{m,j,n}^{(\ell)} - \hat{c}_{m,j,n}^{(\ell)} \right) q_{L-1}^{-1} \bmod q_\ell$ ;
```

producing the residues $\hat{c}$. This computation is parallelized across the coefficients of all ciphertexts, while both ciphertext components are processed within each thread.

The resulting residues $\hat{c}$ are then transformed back to the NTT domain (Lines 7–8). Finally, the second coefficient-wise kernel (Lines 9–12) uses these residues to update the ciphertext coefficients associated with the remaining moduli $\{q_0, \dots, q_{L-2}\}$, producing the rescaled ciphertext $\tilde{c}$.

5 Experimental Results

This section presents the experimental evaluation of the proposed layer-level GPU-accelerated tensor operations for CKKS-based CNN inference. The experimental evaluation is divided into two parts: synthetic benchmarks of the individual tensor layers and complete encrypted CNN inference.

The synthetic benchmarks evaluate the proposed convolution, pooling, fully connected, and square activation layers under different configurations. These experiments compare two GPU implementations on the compact DGX platform against a multicore CPU implementation on the server-class Sapphire system. The GPU implementations use Troy as the underlying HE library, whereas the CPU implementation uses SEAL.

The GPU evaluation compares our proposed layer-level implementation against an operation-level baseline, both using Troy as the underlying execution backend. The proposed approach extends the tensor layers developed in our previous work [13] with dedicated CUDA kernels that jointly process the ciphertexts involved in each layer, exploiting parallelism across both ciphertexts and their polynomial coefficients. In contrast, the baseline combines the original TenSEAL-based tensor layers from our previous work [13] with Troy without these dedicated kernels. Executing a layer thus

requires decomposing it into sequences of individual ciphertext operations and invoking separate Troy GPU kernels for each primitive, including addition, multiplication, rescaling, and relinearization. Finally, for the server-class comparison, the CPU implementation corresponds to our previous multicore implementation [13], executed on the 96-core **Sapphire** server.

The complete encrypted inference experiments evaluate the **CryptoNets** and **MiniLoLa** architectures on the two GPU platforms considered in this work, the compact DGX and the edge **Thor**, and on the server-class 96-core **Sapphire** system. We report both the execution time of the individual network layers and the overall encrypted inference time, allowing the proposed GPU kernels to be evaluated across two different GPU architectures and compared with the multicore CPU implementation.

5.1 Experimental Setup

Experiments were conducted on three platforms: a multicore CPU server and two GPU-based platforms:

- **Sapphire**: a server with two 48-core Intel Xeon Platinum Sapphire Rapids 8468 processors (96 cores total, 2.10 GHz) and 2 TiB of DDR5 memory (4800 MHz).
- **DGX**: an NVIDIA DGX Spark platform featuring an NVIDIA GB10 Grace Blackwell Superchip, comprising a Blackwell-architecture GPU and a 20-core Arm CPU (10 Cortex-X925 cores at up to 3.90 GHz and 10 Cortex-A725 cores at up to 2.81 GHz) coupled via NVLink-C2C, with 121 GiB of unified memory.
- **Thor**: an NVIDIA Jetson AGX Thor Developer Kit featuring a Blackwell-architecture GPU and a 14-core Arm CPU (Neoverse-V3AE, operating at up to 2.60 GHz), with 122 GiB of unified memory.

Regarding the software environment, the CPU implementation is based on TenSEAL v0.3.15 [15] and SEAL v4.1.1 [44]. On **Sapphire**, TenSEAL was linked with Intel HEXL v1.2.6 to accelerate polynomial arithmetic using AVX-512 instructions. For GPU execution, we extended Troy [14], a CUDA-based implementation of the BFV, BGV, and CKKS HE schemes based on SEAL, and integrated it into TenSEAL. This extended framework provides the basis for the GPU-accelerated tensor operations proposed in this work.

All experiments use a polynomial modulus degree of $N = 8192$, a global scale of $\Delta = 2^{26}$, and a coefficient modulus chain with bit sizes [31, 26, 26, 26, 26, 26, 26, 31], resulting in a total coefficient modulus size of 218 bits. These parameters were selected to provide the modulus levels required by the multiplicative depth of the evaluated networks while satisfying the 128-bit security recommendation of the Homomorphic Encryption Standard [9] for the ternary secret-key distribution used by both Troy and SEAL. The batch size B is fixed to $N/2 = 4096$ in all experiments, corresponding to the maximum number of slots available in a CKKS ciphertext for this polynomial modulus degree and thus fully exploiting the SIMD parallelism provided by CKKS.

All reported execution times correspond to the mean of three independent iterations. On both GPU platforms, CUDA kernels are configured with 256 threads per block, corresponding to the default block size used by Troy. Since Troy is also employed as the HE backend of our GPU baseline, we retain the same default configuration

in our kernels to ensure a consistent execution setup and a fair comparison with the baseline.

5.2 Synthetic Layer Benchmarks

To evaluate each tensor layer independently, we benchmark the proposed convolution, pooling, fully connected, and square activation layers under workloads of increasing computational complexity. The proposed layer-level GPU implementation on DGX is compared on the same platform against the operation-level GPU baseline, as well as against the multithreaded CPU implementation using all 96 cores of **Sapphire**.

The three implementations exploit parallelism at different granularities. On **Sapphire**, ciphertexts are distributed among the available CPU cores, with each core computing one or more complete output ciphertexts. The operation-level GPU baseline preserves the original layer execution structure, confining CUDA parallelism to the individual homomorphic operations provided by **Troy**, such as addition, multiplication, relinearization, and rescaling. In contrast, the proposed layer-level GPU implementation exposes an additional dimension of tensor-level parallelism across encrypted tensor elements. Exploiting this parallelism requires implementing the tensor layers directly on the GPU through the dedicated CUDA kernels described in Section 4 (Algorithms 1–6), extending the operation-level baseline model to support the parallel execution of complete tensor layers rather than invoking individual ciphertext operations.

This difference in execution granularity also affects the handling of plaintext weights. In the multicore CPU implementation, each core independently encodes the plaintext weights required for its assigned output ciphertexts, which may result in the same weight being encoded by multiple cores. In the proposed GPU implementation, plaintext representations of multiple weights are constructed in parallel by distributing the N polynomial coefficients of each weight across CUDA threads. The resulting plaintexts are encoded once and reused by the layer-level kernels, enabling concurrent operations on the polynomial representations of multiple tensor elements.

To analyze performance bottlenecks, execution times are decomposed into their constituent stages for each layer. Convolution and fully connected layers include plaintext encoding, tensor computation, and rescaling; square activation includes square computation, relinearization, and rescaling; and pooling involves tensor computation only.

5.2.1 Convolution

Figure 2 reports the execution time of the convolution layer for different numbers of input and output channels and filter sizes. The number of input channels (C_i) ranges from 1 to 3, the number of output channels (C_o) from 3 to 7, and the filter size ($H_f \times W_f$) from 1×1 to 7×7 , with the padding (P) adjusted accordingly. As expected, the execution time increases with the computational workload, particularly as the filter size and the number of channels increase.

The proposed layer-level GPU implementation consistently achieves the lowest execution time across all evaluated configurations. Compared with the 96-core **Sapphire**

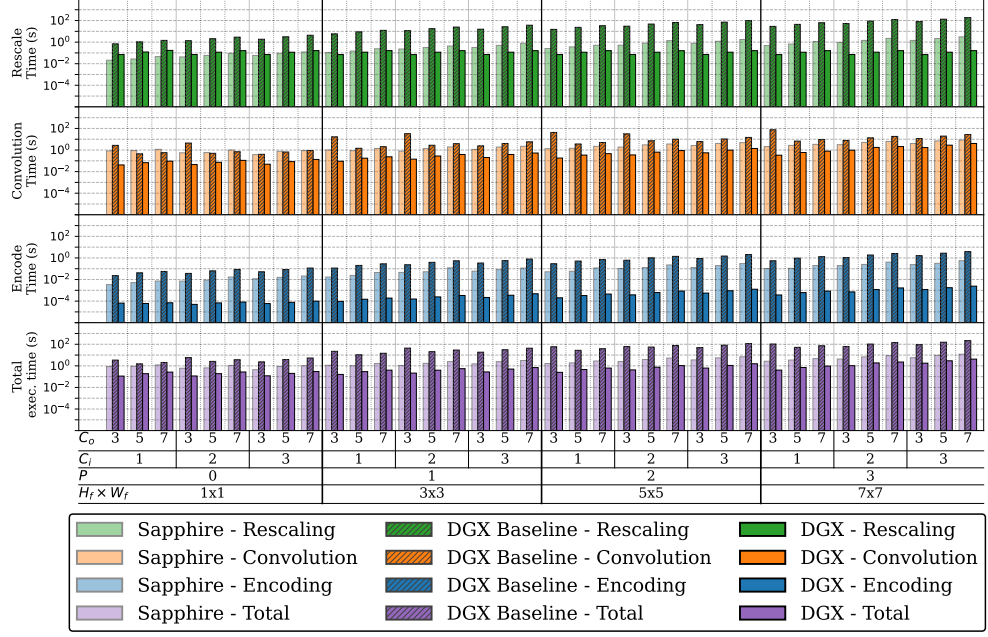


Fig. 2 Execution time of the convolution layer for different input channels (C_i), output channels (C_o), padding (P), and filter sizes ($H_f \times W_f$).

implementation, it achieves speedups ranging from $2.87\times$ to $7.55\times$, with an average of $4.67\times$. More significantly, compared with the operation-level baseline running on the same DGX platform, the proposed implementation achieves speedups ranging from $7.82\times$ to $258.30\times$, with an average of $69.52\times$. Since both GPU implementations run on the same hardware, these improvements isolate the impact of the proposed execution strategy from the underlying device. In particular, they demonstrate the benefit of extending operation-level GPU execution with layer-level parallelism, rather than executing the layer as a sequence of independent ciphertext operations.

The relative advantage over **Sapphire** does not increase monotonically with the filter size and tends to decrease for the largest filters. As the filter size increases, the convolution workload grows substantially; for instance, increasing the filter size from 3×3 to 7×7 increases the number of window elements processed per input channel from 9 to 49. The relative performance of the GPU and CPU implementations therefore varies with the computational characteristics of each convolution configuration rather than scaling directly with the filter size. Nevertheless, the layer-level implementation remains faster than **Sapphire** across all evaluated convolution configurations.

Finally, the execution-time breakdown further illustrates the impact of the proposed parallelization strategy. In the layer-level implementation on DGX, convolution accounts on average for 70.0% of the total execution time, while rescaling represents 29.9% and encoding less than 0.1%. In contrast, rescaling dominates the operation-level DGX baseline, accounting for 72.5% of the execution time, while convolution

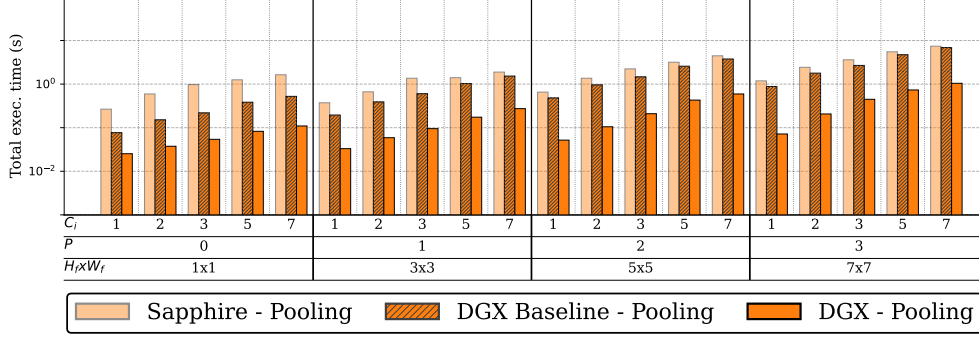


Fig. 3 Execution time of the pooling layer for different input channels (C_i), padding (P), and pooling window sizes ($H_f \times W_f$).

and encoding account for 25.8% and 1.7%, respectively. This behavior is consistent with the operation-level execution model of the baseline, where rescaling is applied independently to each output ciphertext. Since rescaling involves computationally expensive polynomial transformations, such as NTT and INTT operations, this execution model limits the parallelism that can be exploited across ciphertexts. The proposed implementation instead performs these transformations across multiple ciphertexts concurrently, exploiting the additional layer-level parallelism and substantially reducing the relative contribution of rescaling to the overall execution time. On **Sapphire**, encoding and rescaling times were measured on a single CPU core, since each core independently performs these operations on its assigned workload. Based on these measurements, convolution accounts for 79.6% of the total execution time, while rescaling and encoding represent 17.3% and 3.1%, respectively. The higher encoding contribution on **Sapphire** reflects the weight-encoding strategy of the original multicore implementation described above.

5.2.2 Pooling

Figure 3 reports the execution time of the pooling layer for the different configurations considered. In the HE CNNs evaluated in this work, pooling is implemented as a sum over the elements of each pooling window, without performing the division typically required to compute an average. Consequently, the operation only involves coefficient-wise additions between ciphertexts and does not require encoding, relinearization, or rescaling.

The proposed layer-level GPU implementation consistently achieves the lowest execution time across all evaluated configurations. Compared with the 96-core **Sapphire** implementation, it achieves speedups ranging from $6.87\times$ to $18.12\times$, with an average of $11.45\times$. Compared with the operation-level baseline running on the same DGX platform, the proposed implementation achieves speedups ranging from $3.06\times$ to $12.25\times$, with an average of $6.43\times$. Since pooling only requires additions over each pooling window, its simple computational structure is particularly well suited to the layer-level implementation, which processes coefficients from multiple ciphertexts concurrently

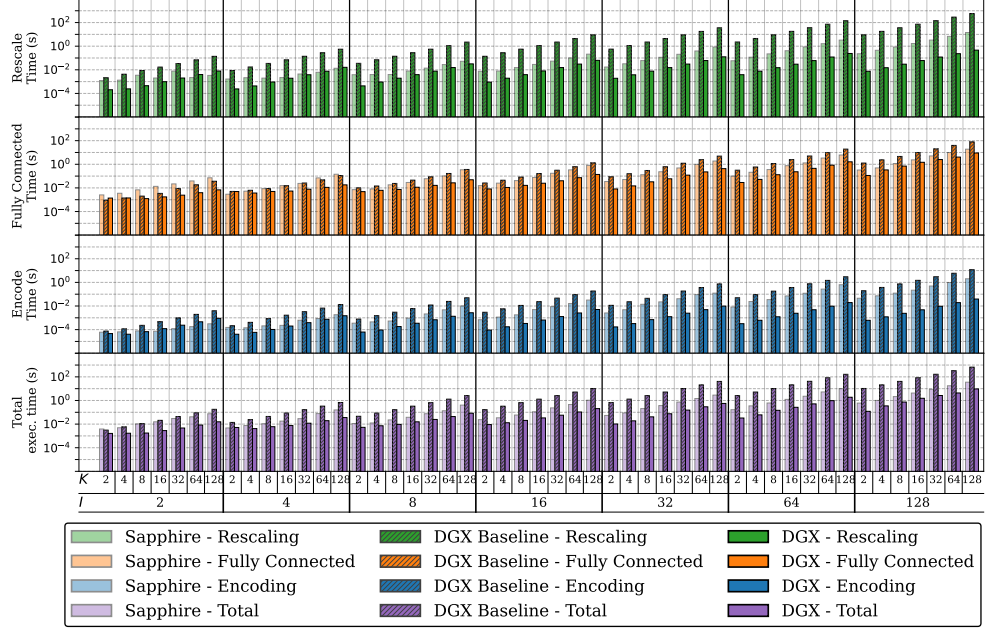


Fig. 4 Execution time of the fully connected layer for different input (I) and output (K) dimensions.

rather than operating on them individually. In contrast, as the pooling window and workload increase, the execution time of the operation-level DGX baseline approaches that of the 96-core **Sapphire** implementation, with the difference decreasing to only $1.07\times$ for the largest evaluated configuration.

5.2.3 Fully Connected

Figure 4 reports the execution time of the fully connected layer for input and output dimensions ranging from 2 to 128. Following the general formulation of Section 4.3, where $EI \in M \times I$, the synthetic benchmark assumes $M = I$, resulting in a square encrypted input matrix of dimensions $I \times I$, while K determines the number of output columns. In this case, the execution time is decomposed into encoding, fully connected computation, and rescaling.

The layer-level GPU implementation achieves the lowest execution time for almost all evaluated configurations, with an average speedup of $3.86\times$ over the 96-core **Sapphire** implementation and a maximum of $6.33\times$. As the layer dimensions increase, the proposed implementation generally benefits from a larger amount of parallel work.

The improvement is substantially larger over the operation-level GPU baseline, with speedups from $1.90\times$ to $87.09\times$ and an average of $41.65\times$. Since both implementations run on DGX, this difference highlights the impact of the proposed layer-level parallelization. While the baseline computes the layer through individual ciphertext operations, the proposed kernels process multiple output ciphertexts and their polynomial coefficients concurrently.

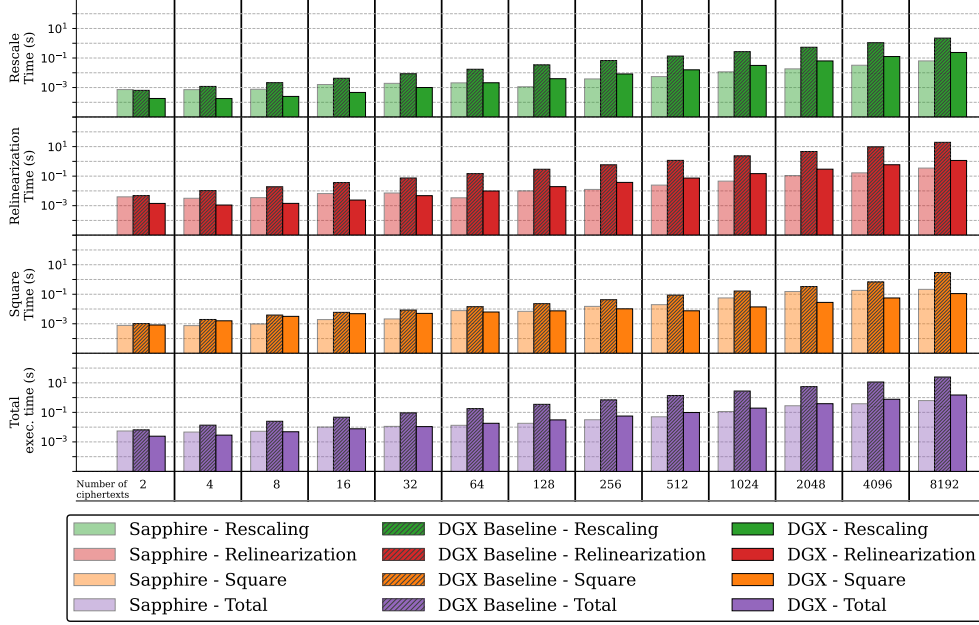


Fig. 5 Execution time of the square activation for an increasing number of ciphertexts. The execution time is decomposed into square, relinearization, and rescaling stages.

The execution-time breakdown further illustrates this difference. In the proposed implementation, fully connected computation accounts on average for 78.0% of the execution time, followed by rescaling with 20.0% and encoding with 2.0%. In the operation-level baseline, rescaling dominates with 83.0%, compared with 15.1% for fully connected computation and 1.9% for encoding, as rescaling is performed independently for each ciphertext. On **Sapphire**, fully connected computation, rescaling, and encoding account for 69.3%, 27.0%, and 3.8%, respectively.

5.2.4 Square Activation

Figure 5 reports the execution time of the square activation for an increasing number of ciphertexts. The proposed layer-level GPU implementation achieves the lowest execution time for the smallest configurations and remains substantially faster than the operation-level DGX baseline across the entire evaluated range. Compared with the 96-core **Sapphire** implementation, the Sapphire-to-DGX ratio ranges from $0.42\times$ to $2.25\times$, with an average of $0.92\times$. Thus, DGX outperforms **Sapphire** for the smallest workloads, whereas **Sapphire** becomes more competitive as the number of ciphertexts increases. For 8192 ciphertexts, the ratio decreases to $0.42\times$, indicating that **Sapphire** is faster for the largest evaluated configuration.

A markedly different behavior is observed for the operation-level DGX baseline. The proposed layer-level implementation is between $2.66\times$ and $16.41\times$ faster than the baseline, with an average speedup of $10.35\times$. Moreover, the performance gap generally

increases with the number of ciphertexts. This difference follows from the parallelization strategy of the proposed implementation. The operation-level baseline applies the individual ciphertext operations provided by **Troy** independently, whereas the layer-level CUDA kernels process polynomial data from multiple ciphertexts concurrently. This additional layer-level parallelism becomes increasingly relevant as the number of ciphertexts grows.

The execution-time breakdown shows that relinearization is the dominant stage in all three implementations. It accounts on average for 59.0% of the total execution time in the proposed layer-level DGX implementation, compared with 80.9% in the operation-level DGX baseline and 52.7% on **Sapphire**. The square operation represents 29.1%, 9.6%, and 35.2%, respectively, while rescaling accounts for 11.9%, 9.4%, and 12.1%. The particularly high relinearization contribution in the operation-level DGX baseline explains a substantial part of its increasing execution time. In absolute terms, the proposed implementation accelerates relinearization over the operation-level DGX baseline by $3.38\times$ – $16.85\times$, with an average speedup of $14.08\times$, while rescaling is accelerated by $3.54\times$ – $9.38\times$, with an average of $8.06\times$.

The comparison with **Sapphire** also reveals that the behavior of the individual stages changes with the workload. In particular, the proposed layer-level DGX implementation provides a clear advantage in relinearization and rescaling for the smallest configurations, but this advantage decreases as the number of ciphertexts grows. Relinearization comprises multiple computational stages and relies on costly NTT and inverse NTT transformations [10]. Its performance is therefore strongly determined by the efficiency of these underlying transformations. Consequently, although the proposed layer-level implementation substantially improves square activation over the operation-level DGX baseline, the 96-core **Sapphire** implementation remains competitive for large workloads. This behavior differs from convolution, pooling, and fully connected layers, where the proposed layer-level parallelization provides a more consistent advantage over the CPU implementation.

5.3 CryptoNets Architecture

We first evaluate the proposed layer-level GPU implementation on the **CryptoNets** [16] architecture for MNIST digit classification. The architecture employed in our experiments is summarized in Table 1. The resulting network achieves approximately 98% accuracy, preserving the predictive performance of the unencrypted network. The evaluation considers the execution time of each encrypted layer and the complete forward pass on DGX and **Thor**, compared with the multicore CPU implementation on **Sapphire**, allowing us to assess whether the trends observed in the synthetic benchmarks are maintained in a complete CNN inference.

Figure 6 reports the execution time of the individual layers and the complete forward pass on the three platforms. The convolutional layers benefit significantly from GPU acceleration, achieving speedups of $4.85\times$ and $3.36\times$ on DGX, and $2.74\times$ and $1.45\times$ on **Thor**, for the first and second convolutional layers, respectively. The pooling layers exhibit the largest improvements, reaching speedups of $9.85\times$ and $10.62\times$ on DGX, and $12.46\times$ and $12.63\times$ on **Thor**. The fully connected layers also benefit from

Table 1 CryptoNets model architecture for MNIST digit classification.

#	Layer	Kernel size	Padding	Stride	Output shape
L1	Conv2D	$(5 \times 1 \times 5 \times 5)$	(1, 1)	(2, 2)	$5 \times 13 \times 13$
L2	Square-Activation	—	—	—	$5 \times 13 \times 13$
L3	Pool	$(1 \times 3 \times 3)$	(1, 1)	(1, 1)	$5 \times 13 \times 13$
L4	Conv2D	$(50 \times 5 \times 5 \times 5)$	(0, 0)	(2, 2)	$50 \times 5 \times 5$
L5	Pool	$(1 \times 3 \times 3)$	(1, 1)	(1, 1)	$50 \times 5 \times 5$
L6	FC1	—	—	—	100
L7	Square-Activation	—	—	—	100
L8	FC2	—	—	—	10

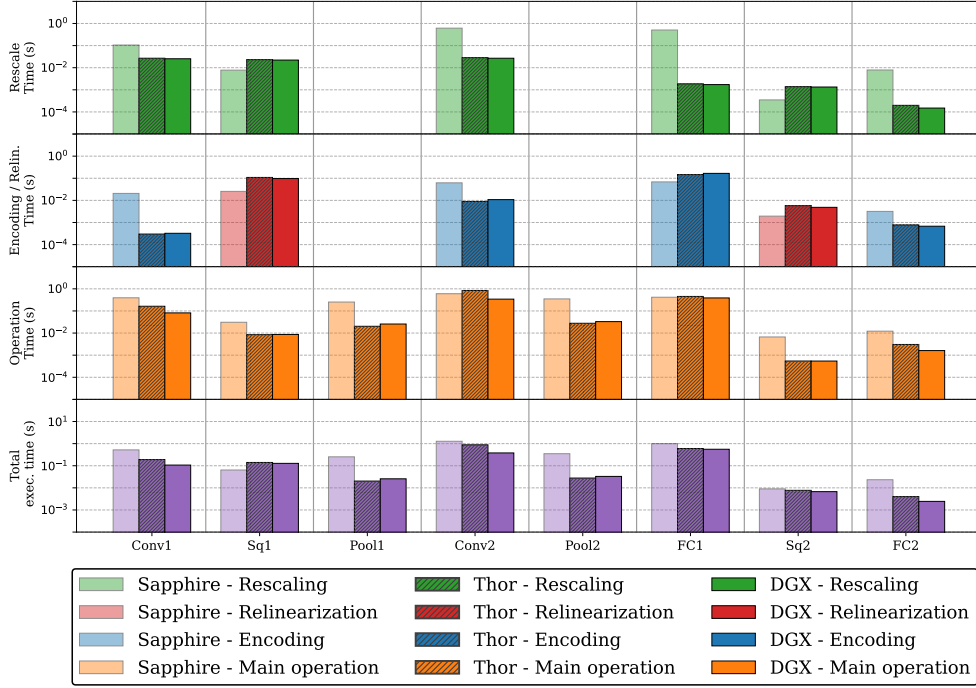


Fig. 6 Per-layer and complete forward-pass execution times of the CryptoNets architecture on DGX, Thor, and Sapphire for $N/2$ batched images.

GPU execution, with speedups ranging from $1.78\times$ to $9.51\times$ on DGX and from $1.65\times$ to $5.78\times$ on Thor.

The execution-time breakdown shows that the main computation dominates the convolutional layers on both GPUs, accounting for 76.0%–90.1% on DGX and 85.7%–95.7% on Thor, while encoding remains a minor contribution. On Sapphire, rescaling has a greater impact, particularly in the second convolution, where it represents 48.0% of the execution time. For the fully connected layers, encoding becomes more noticeable

on the GPUs, accounting for 27.8%–30.0% on DGX and 19.2%–24.4% on Thor, whereas rescaling has a larger contribution on Sapphire.

The first square activation favors Sapphire, which is $2.00\times$ faster than DGX and $2.20\times$ faster than Thor, consistent with the synthetic results. In contrast, the smaller second square activation is faster on the GPUs, with speedups of $1.33\times$ on DGX and $1.16\times$ on Thor over Sapphire. Specifically, relinearization dominates both square activations on the GPUs, accounting for 72.1%–77.5% of their execution time, whereas on Sapphire the square computation itself represents a considerably larger fraction.

Despite the lower performance of the GPUs for the first square activation, the acceleration achieved across the remaining layers dominates the complete inference time. In fact, the complete forward pass requires 3.752 s on Sapphire, 1.894 s on Thor, and 1.270 s on DGX. Consequently, the complete inference is $2.95\times$ faster on DGX and $1.98\times$ faster on Thor than on Sapphire, while DGX is $1.49\times$ faster than Thor.

5.4 Mini-LoLa Architecture

We next evaluate the proposed layer-level GPU implementation on Mini-LoLa, an architecture derived from the M5 network proposed in UniHENN [19], which is a reduced version of the LoLa model [17], primarily using fewer output channels. The architecture employed in our experiments is summarized in Table 2. The resulting network achieves approximately 75% accuracy, maintaining the predictive performance of the unencrypted network. The execution time of each encrypted layer and the complete forward pass are evaluated on Sapphire, DGX, and Thor, providing a more computationally demanding scenario than CryptoNets.

Table 2 Mini-LoLa model architecture for CIFAR-10 object classification.

#	Layer	Kernel size	Padding	Stride	Output shape
L1	Conv2D	$(16 \times 3 \times 3 \times 3)$	(1, 1)	(1, 1)	$16 \times 32 \times 32$
L2	Pool	$(16 \times 2 \times 2)$	(0, 0)	(2, 2)	$16 \times 16 \times 16$
L3	Conv2D	$(64 \times 16 \times 4 \times 4)$	(0, 0)	(1, 1)	$64 \times 13 \times 13$
L4	Square-Activation	—	—	—	$64 \times 13 \times 13$
L5	Pool	$(64 \times 2 \times 2)$	(0, 0)	(2, 2)	$64 \times 6 \times 6$
L6	Conv2D	$(128 \times 64 \times 3 \times 3)$	(0, 0)	(1, 1)	$128 \times 4 \times 4$
L7	Square-Activation	—	—	—	$128 \times 4 \times 4$
L8	Pool	$(128 \times 4 \times 4)$	(0, 0)	(4, 4)	$128 \times 1 \times 1$
L9	FC1	—	—	—	10

As shown in Figure 7, the convolutional and pooling layers consistently benefit from GPU acceleration. The three convolutional layers achieve speedups of $5.64\times$, $2.16\times$, and $1.80\times$ on DGX, and $3.05\times$, $1.71\times$, and $1.39\times$ on Thor, respectively. Pooling provides the largest gains, with speedups ranging from $7.97\times$ to $18.38\times$ on DGX and from $9.25\times$ to $20.04\times$ on Thor. The fully connected layer is also accelerated by $6.75\times$ and $6.52\times$ on DGX and Thor, respectively.

The execution-time breakdown shows that the main computation dominates the convolutional layers on both GPUs, accounting for 77.1%–97.9% on DGX and 85.8%–98.2% on Thor, while encoding remains a minor contribution. On Sapphire, the first

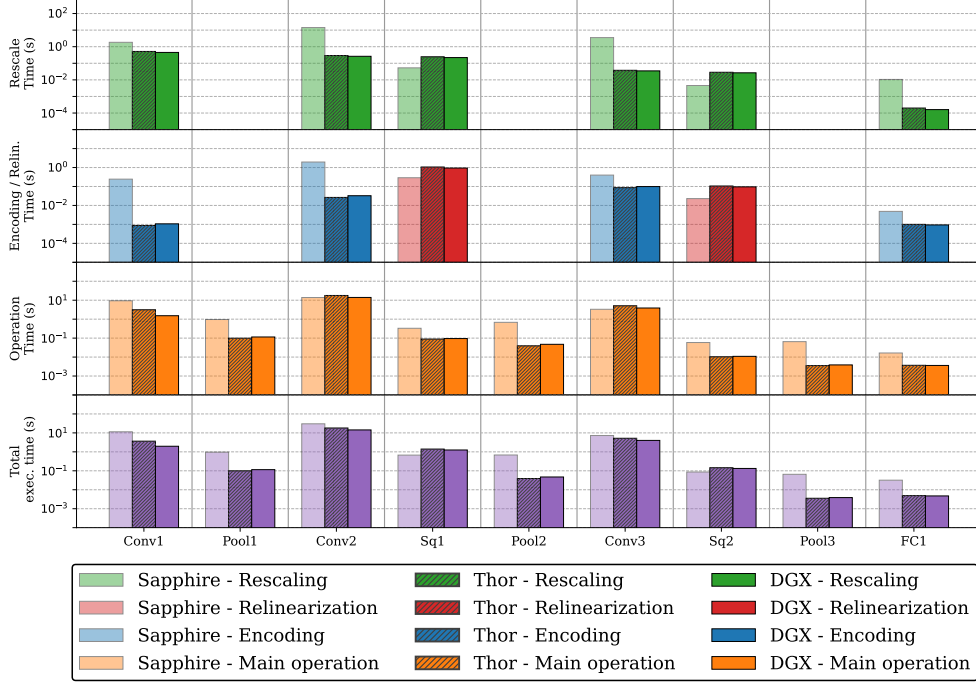


Fig. 7 Per-layer and complete forward-pass execution times of the Mini-LoLa architecture on DGX, Thor, and Sapphire for $N/2$ batched images.

convolution exhibits a similar behavior, whereas rescaling becomes comparable to the main computation in the second and third convolutions, accounting for approximately half of their execution time.

As observed in the previous experiments, the square activations remain the main limitation of the GPU implementation. **Sapphire** is $1.94\times$ and $2.17\times$ faster than **DGX** and **Thor** for the first square activation, and $1.90\times$ and $2.07\times$ faster for the second. Specifically, relinearization dominates both square activations on the GPUs, accounting for 71.8–76.1% of their execution time, whereas the square computation itself represents a considerably larger fraction on **Sapphire**.

Despite the overhead of the square activations, the acceleration achieved by the remaining layers compensates for this cost. In fact, the complete forward pass requires 51.80 s on **Sapphire**, 28.67 s on **Thor**, and 21.87 s on **DGX**. Consequently, the complete inference is $2.37\times$ faster on **DGX** and $1.81\times$ faster on **Thor** than on **Sapphire**, while **DGX** is $1.31\times$ faster than **Thor**.

5.5 Error Analysis

Table 3 analyzes the numerical error introduced by encrypted inference with respect to the corresponding plaintext execution on the GPU platforms. The comparison is performed in terms of the variation in classification accuracy, reported as Accuracy

($\Delta\%$), mean absolute error (MAE), root mean squared error (RMSE), and maximum absolute error (MaxAE) for each class.

Both DGX and Thor closely reproduce the plaintext outputs. For CryptoNets, the classification accuracy is preserved for all classes on both platforms. Numerical differences remain small, with MAE and RMSE values on the order of 10^{-3} and maximum absolute errors on the order of 10^{-2} . For Mini-LoLa, deviations are even smaller, with MAE and RMSE values on the order of 10^{-4} and MaxAE on the order of 10^{-3} . These differences have a negligible effect on the final classification results: DGX preserves the accuracy of every class, while Thor only exhibits a -0.10 percentage-point variation for class 8.

Overall, the results show that the GPU implementation introduces only small numerical deviations with respect to plaintext inference, while essentially preserving the classification behavior of both networks.

Table 3 Numerical differences between encrypted and plaintext inference for CryptoNets and Mini-LoLa in the DGX and Thor platforms. Accuracy differences are expressed in percentage points.

Network	Platform	Metric	Class									
			0	1	2	3	4	5	6	7	8	9
CryptoNets	DGX	Accuracy ($\Delta\%$)	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
		MAE (10^{-3})	3.29	2.50	3.34	3.90	3.36	3.58	3.39	3.06	3.23	3.82
		RMSE (10^{-3})	4.85	3.61	4.87	5.71	4.96	5.29	4.91	4.50	4.70	5.66
		MaxAE (10^{-2})	4.64	2.44	3.66	4.55	3.68	4.74	3.56	3.81	3.38	3.51
	Thor	Accuracy ($\Delta\%$)	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
		MAE (10^{-3})	3.48	2.62	3.37	3.87	3.22	3.64	3.37	3.02	3.31	3.83
		RMSE (10^{-3})	5.10	3.83	4.82	5.58	4.66	5.36	4.95	4.38	4.89	5.64
		MaxAE (10^{-2})	4.09	2.30	3.99	4.14	3.52	4.24	3.52	2.97	4.45	4.23
Mini-LoLa	DGX	Accuracy ($\Delta\%$)	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
		MAE (10^{-4})	2.98	3.04	3.01	3.05	2.65	3.13	3.07	3.02	2.87	2.99
		RMSE (10^{-4})	4.48	4.52	4.61	4.49	3.97	4.66	4.66	4.46	4.23	4.45
		MaxAE (10^{-3})	4.16	3.95	4.29	3.54	3.16	2.78	4.23	3.26	4.52	3.90
	Thor	Accuracy ($\Delta\%$)	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-0.10	0.00
		MAE (10^{-4})	2.92	2.97	3.10	2.88	2.92	3.11	3.14	3.01	2.99	2.98
		RMSE (10^{-4})	4.44	4.42	4.67	4.33	4.56	4.66	4.74	4.51	4.46	4.53
		MaxAE (10^{-3})	4.00	4.05	3.73	3.77	3.81	4.17	5.53	4.20	3.60	5.96

6 Conclusions

We have presented a layer-level GPU execution strategy for encrypted CNN inference, extending our previous multicore CPU approach beyond server-class CPU systems to modern GPU platforms, including compact and edge devices, within the TenSEAL-Troy framework. The proposed implementation moves the execution of convolution, pooling, square activation, and fully connected layers beyond sequences of individual HE operations by introducing dedicated CUDA kernels that operate directly on the RNS-CKKS representation. This design exposes layer-level parallelism across encrypted tensor elements while preserving the high-level programming model provided by TenSEAL.

The proposed layer-level GPU execution strategy achieves substantial performance gains across the evaluated tensor operations. On DGX Spark, the proposed implementation achieves speedups of up to $7.55\times$, $18.12\times$, and $6.33\times$ over the 96-core server **Sapphire** implementation for convolution, pooling, and fully connected layers, respectively. Compared with the operation-level GPU baseline on the same device, speedups reach up to $258.30\times$, $12.25\times$, $87.09\times$, and $16.41\times$ for convolution, pooling, fully connected, and square activation, respectively. The comparison against the operation-level GPU baseline on the same device further shows that these gains are not solely associated with executing the operations on the GPU, but also with changing the execution granularity and exploiting parallelism across multiple encrypted tensor elements. These improvements extend to end-to-end inference, with speedups of $2.95\times$ for **CryptoNets** and $2.37\times$ for **Mini-LoLa** on DGX Spark over the multicore CPU implementation on **Sapphire**. Square activation exhibits a different behavior, particularly due to the cost of relinearization, for which increasing the workload does not provide the same GPU advantage. This result highlights the need to adapt the parallelization strategy to the computational characteristics of each HE operation.

Future work will focus on exploring alternative GPU mappings for each tensor operation, since the most appropriate execution strategy depends on its computational structure and per-output workload. While the current layer-level kernels assign independent output coefficients to individual CUDA threads, alternative mappings could distribute the computation of an output cooperatively among the threads of a warp or block. Our NTT implementation already adopts such a block-level mapping, where one RNS polynomial is assigned to a block and its threads cooperatively execute the butterfly operations while keeping the polynomial in shared memory throughout the transformation. Extending this strategy to tensor operations is non-trivial, since the data required to compute an output depends on the layer configuration and may exceed the shared-memory capacity of a block. Consequently, we will investigate thread- and block-level mappings, including cooperative and tiled strategies, to determine the most suitable execution granularity for each tensor operation.

Declarations

Funding Authors from Universitat Jaume I were supported by the project PID2023-146569NB-C22, funded by MICIU/AEI/10.13039/501100011033 and ERDF/UE. Manuel F. Dolz was supported by project CNS2025-165098, funded by MICIU/AEI/10.13039/501100011033, and by the Plan Gen-T grant CIDEXG/2022/13 from Generalitat Valenciana, under which Núria Moreno-Chamorro is supported as a predoctoral researcher. This work was partially supported by Portuguese national funds through Fundação para a Ciência e a Tecnologia I.P. (FCT) under projects UID/50021/2025 and UID/PRR/50021/2025.

Conflict of interest/Competing interests The authors declare that they have no competing interests.

Ethics approval and consent to participate Not applicable.

Consent for publication Not applicable.

Data availability The MNIST and CIFAR-10 datasets used in this work are publicly available at <http://yann.lecun.com/exdb/mnist/> and <https://www.cs.toronto.edu/~kriz/cifar.html>, respectively.

Materials availability Not applicable.

Code availability The source code developed in this work is available at <https://github.com/hpca-uji/TenSEAL-troy> and <https://github.com/hpca-uji/Troy>.

Author contribution Núria Moreno-Chamorro contributed to the conceptualization and methodology, implemented the proposed GPU execution strategy and CUDA kernels, and conducted the experimental evaluation and validation. Leonel Sousa and Aleksandar Ilić contributed to the architectural analysis of the target GPU platforms. Maribel Castillo contributed to the methodology and supervised the overall research. Manuel F. Dolz contributed to the methodology, supervised the overall research, and contributed to funding acquisition. All authors contributed to writing the manuscript and approved the final version.

References

- [1] Gentry, C.: A fully homomorphic encryption scheme. (2009). <https://api.semanticscholar.org/CorpusID:53903759>
- [2] Xu, R., Baracaldo, N., Joshi, J.: Privacy-Preserving Machine Learning: Methods, Challenges and Directions. <https://arxiv.org/abs/2108.04417> (2021)
- [3] Lee, J.-W., Kang, H., Lee, Y., Choi, W., Eom, J., Deryabin, M., Lee, E., Lee, J., Yoo, D., Kim, Y.-S., No, J.-S.: Privacy-Preserving Machine Learning with Fully Homomorphic Encryption for Deep Neural Network (2021). <https://arxiv.org/abs/2106.07229>
- [4] Podschwadt, R., Takabi, D., Hu, P.: SoK: Privacy-preserving Deep Learning with Homomorphic Encryption (2022). <https://arxiv.org/abs/2112.12855>
- [5] Cheon, J.H., Kim, A., Kim, M., Song, Y.: Homomorphic Encryption for Arithmetic of Approximate Numbers. Cryptology ePrint Archive, Paper 2016/421 (2016). <https://eprint.iacr.org/2016/421>
- [6] Lyubashevsky, V., Peikert, C., Regev, O.: On Ideal Lattices and Learning with Errors Over Rings. Cryptology ePrint Archive, Paper 2012/230 (2012). <https://eprint.iacr.org/2012/230>
- [7] Cheon, J.H., Han, K., Kim, A., Kim, M., Song, Y.: A Full RNS Variant of Approximate Homomorphic Encryption: 25th International Conference, Calgary, AB, Canada, August 15–17, 2018, Revised Selected Papers. Springer (2019). https://doi.org/10.1007/978-3-030-10970-7_16
- [8] Marcolla, C., Sucasas, V., Manzano, M., Bassoli, R., Fitzek, F.H.P., Aaraj, N.: Survey on Fully Homomorphic Encryption, Theory, and Applications. Cryptology

- ePrint Archive, Paper 2022/1602 (2022). <https://doi.org/10.1109/JPROC.2022.3205665> . <https://eprint.iacr.org/2022/1602>
- [9] Albrecht, M., Chase, M., Chen, H., Ding, J., Goldwasser, S., Gorbunov, S., Halevi, S., Hoffstein, J., Laine, K., Lauter, K., Lokam, S., Micciancio, D., Moody, D., Morrison, T., Sahai, A., Vaikuntanathan, V.: Homomorphic Encryption Standard. Cryptology ePrint Archive, Paper 2019/939 (2019). <https://eprint.iacr.org/2019/939>
 - [10] Gong, Y., Chang, X., Mišić, J., Mišić, V.B., Wang, J., Zhu, H.: Practical Solutions in Fully Homomorphic Encryption – A Survey Analyzing Existing Acceleration Methods (2023). <https://arxiv.org/abs/2303.10877>
 - [11] Podschwadt, R., Takabi, D., Hu, P., Rafiei, M.H., Cai, Z.: A survey of deep learning architectures for privacy-preserving machine learning with fully homomorphic encryption. *IEEE Access* **10**, 117477–117500 (2022) <https://doi.org/10.1109/ACCESS.2022.3219049>
 - [12] Yuan, J., Liu, W., Shi, J., Li, Q.: Approximate homomorphic encryption based privacy-preserving machine learning: a survey. *Artificial Intelligence Review* **58** (2025)
 - [13] Moreno-Chamorro, N., Castillo, M., Aliaga, J.I., Dolz, M.F.: Parallel and memory-efficient homomorphically encrypted cnns using vector and tensor representations. *Journal of Computational and Applied Mathematics* **488**, 117797 (2026) <https://doi.org/10.1016/j.cam.2026.117797>
 - [14] lightbulb128: Troy: GPU implementation of BFV, CKKS and BGV homomorphic encryption schemes (2024). <https://github.com/lightbulb128/troy>
 - [15] Benaissa, A., Retiat, B., Cebere, B., Belfedhal, A.E.: TenSEAL: A Library for Encrypted Tensor Operations Using HE (2021). <https://arxiv.org/abs/2104.03152>
 - [16] Gilad-Bachrach, R., Dowlin, N., Laine, K., Lauter, K., Naehrig, M., Wernsing, J.: CryptoNets: Applying neural networks to encrypted data with high throughput and accuracy. In: *Intl. Conference on Machine Learning*, pp. 201–210 (2016). PMLR
 - [17] Brutzkus, A., Elisha, O., Gilad-Bachrach, R.: Low Latency Privacy Preserving Inference (2019). <https://arxiv.org/abs/1812.10659>
 - [18] Kim, D., Park, J., Kim, J., Kim, S., Ahn, J.H.: HyPHEN: A hybrid packing method and its optimizations for homomorphic encryption-based neural networks. *IEEE Access* **12**, 3024–3038 (2024) <https://doi.org/10.1109/ACCESS.2023.3348170>

- [19] Choi, H., Kim, J., Kim, S., Park, S., Park, J., Choi, W., Kim, H.: UniHENN: Designing faster and more versatile HE-based CNNs without im2col. *IEEE Access* **12**, 109323–109341 (2024)
- [20] Boemer, F., Lao, Y., Cammarota, R., Wierzynski, C.: nGraph-HE: A Graph Compiler for Deep Learning on Homomorphically Encrypted Data (2019). <https://arxiv.org/abs/1810.10121>
- [21] Dathathri, R., Saarikivi, O., Chen, H., Laine, K., Lauter, K., Maleki, S., Musuvathi, M., Mytkowicz, T.: Chet: an optimizing compiler for fully-homomorphic neural-network inferencing, pp. 142–156 (2019). <https://doi.org/10.1145/3314221.3314628>
- [22] Dathathri, R., Kostova, B., Saarikivi, O., Dai, W., Laine, K., Musuvathi, M.: Eva: an encrypted vector arithmetic language and compiler for efficient homomorphic computation. *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (2019)
- [23] Cryptography Research Group, Microsoft: Microsoft SEAL for Python. <https://github.com/microsoft/SEAL-Python>. Python bindings for the Microsoft SEAL homomorphic encryption library using pybind11. MIT License. (2021)
- [24] PALISADE Lattice Cryptography Project: PALISADE Lattice Cryptography Library. <https://palisade-crypto.org/> (2020)
- [25] Badawi, A.A., Alexandru, A., Bates, J., Bergamaschi, F., Cousins, D.B., Erabelli, S., Genise, N., Halevi, S., Hunt, H., Kim, A., Lee, Y., Liu, Z., Micciancio, D., Pascoe, C., Polyakov, Y., Quah, I., R.V., S., Rohloff, K., Saylor, J., Saponitsky, D., Triplett, M., Vaikuntanathan, V., Zucca, V.: OpenFHE: Open-Source Fully Homomorphic Encryption Library. *Cryptology ePrint Archive*, Paper 2022/915 (2022). <https://eprint.iacr.org/2022/915>
- [26] EPFL-LDS, Tune Insight SA: Lattigo v6. Online: <https://github.com/tuneinsight/lattigo> (2024)
- [27] Dai, W., Sunar, B.: cuHE: A Homomorphic Encryption Accelerator Library. *Cryptology ePrint Archive*, Paper 2015/818 (2015). <https://eprint.iacr.org/2015/818>
- [28] Zhai, Y., Ibrahim, M., Qiu, Y., Boemer, F., Chen, Z., Titov, A., Lyashevsky, A.: Accelerating Encrypted Computing on Intel GPUs (2021). <https://arxiv.org/abs/2109.14704>
- [29] Riazi, M.S., Laine, K., Pelton, B., Dai, W.: HEAX: An Architecture for Computing on Encrypted Data. *Cryptology ePrint Archive*, Paper 2019/1066 (2019). <https://eprint.iacr.org/2019/1066>

- [30] Yang, H., Shen, S., Dai, W., Zhou, L., Liu, Z., Zhao, Y.: Phantom: A CUDA-Accelerated Word-Wise Homomorphic Encryption Library. Cryptology ePrint Archive, Paper 2023/049 (2023). <https://doi.org/10.1109/TDSC.2024.3363900> . <https://eprint.iacr.org/2023/049>
- [31] Choi, W., Kim, J., Ahn, J.H.: Cheddar: A Swift Fully Homomorphic Encryption Library Designed for GPU Architectures. ACM (2025). <https://doi.org/10.1145/3760250.3762223> . <http://dx.doi.org/10.1145/3760250.3762223>
- [32] Özcan, A., Savaş, E.: HEonGPU: a GPU-based Fully Homomorphic Encryption Library 1.0. Cryptology ePrint Archive, Paper 2024/1543 (2024). <https://eprint.iacr.org/2024/1543>
- [33] Dubetsky, V., Cano, M.-D.: Feasibility study of cuda-accelerated homomorphic encryption and benchmarking on consumer-grade and embedded gpus. Big Data and Cognitive Computing **10**(3) (2026)
- [34] Fan, S., Wang, Z., Xu, W., Hou, R., Meng, D., Zhang, M.: TensorFHE: Achieving Practical Computation on Encrypted Data Using GPGPU (2022). <https://arxiv.org/abs/2212.14191>
- [35] Liu, X., Liu, Z., Li, Q., Xu, K., Xu, M.: Pencil: Private and Extensible Collaborative Learning without the Non-Colluding Assumption. (2024)
- [36] D’Agata, L., Agulló-Domingo, C., Vera-López, O., Shivdikar, K., Yudha, A.W.B., Yaman, F., Kaeli, D., Abellán, J.L., Colbert, I., Cano, J.: GPU Acceleration of Sparse Fully Homomorphic Encrypted DNNs (2026). <https://arxiv.org/abs/2604.11659>
- [37] Seo, S., Min, C.: Zephyr: GPU-Efficient Homomorphic Encryption for Privacy-Preserving Transformer Inference. Cryptology ePrint Archive, Paper 2026/932 (2026). <https://eprint.iacr.org/2026/932>
- [38] Cooley, J.W., Tukey, J.W.: An algorithm for the machine calculation of complex fourier series. Mathematics of Computation **19**, 297–301 (1965)
- [39] Marti-Puig, P., Reig Bolano, R.: Radix-4 fft algorithms with ordered input and output data. In: 2009 16th International Conference on Digital Signal Processing, pp. 1–6 (2009). <https://doi.org/10.1109/ICDSP.2009.5201151>
- [40] Bailey, D.H.: Ffts in external or hierarchical memory. In: Supercomputing ’89: Proceedings of the 1989 ACM/IEEE Conference on Supercomputing, pp. 234–242 (1989). <https://doi.org/10.1145/76263.76288>
- [41] Satriawan, A., Mareta, R., Lee, H.: A Complete Beginner Guide to the Number Theoretic Transform (NTT). Cryptology ePrint Archive, Paper 2024/585 (2024). <https://doi.org/10.1109/ACCESS.2023.3294446> . <https://eprint.iacr.org/>

- [42] Boemer, F., Kim, S., Seifu, G., Souza, F.D.M., Gopal, V.: Intel HEXL: Accelerating Homomorphic Encryption with Intel AVX512-IFMA52 (2021). <https://arxiv.org/abs/2103.16400>
- [43] Barrett, P.: Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor, 311–323 (1987)
- [44] Microsoft SEAL (release 4.1). <https://github.com/Microsoft/SEAL>. Microsoft Research, Redmond, WA. (2023)