Aurva: Reconfigurable Fabric based Kernel for Secure CNN Inference

Ketan Anand Roy Daksh Pandey

September 3, 2025

Abstract

Deep neural networks are increasingly deployed in cloud environments where data privacy is a critical concern. Fully Homomorphic Encryption (FHE) provides strong security guarantees by enabling computation directly over encrypted data, but its computational overhead remains the main bottleneck for deployment at scale. In this work, we focus on accelerating convolutional neural network (CNN) inference over CKKS-encrypted data. We propose **Aurva**, a reconfigurable hardware kernel that implements a convolutional unit tailored to the CKKS scheme. The kernel exploits the SIMD property of CKKS to pack multiple plaintext values into the $N/2$ independent slots of a ciphertext, and then applies a Number Theoretic Transform (NTT) to bring the ciphertexts into a representation suitable for efficient homomorphic convolution. By mapping the polynomial arithmetic and multiply–accumulate patterns of CNNs onto a pipelined FPGA design, our approach bridges the gap between the mathematical structure of CKKS and the architectural requirements of hardware acceleration. The result is a secure, scalable building block for encrypted CNN inference on cloud platforms.

1 Introduction

The increasing demand for privacy-preserving machine learning in domains such as healthcare, finance, and defense requires computation frameworks that operate directly on encrypted data. Fully Homomorphic Encryption (FHE) addresses this need by allowing arbitrary computations on ciphertexts without decryption. However, despite its strong security guarantees, FHE introduces large computational costs, particularly for deep learning workloads.

Among FHE schemes, CKKS (Cheon–Kim–Kim–Song) has become a practical choice for encrypted machine learning due to its support for approximate arithmetic on vectors. By enabling SIMD-style packing of multiple plaintext values into a single ciphertext, CKKS aligns naturally with the vectorized computations of neural networks. Yet, the convolutional layers—which dominate the cost of CNN inference—require repeated polynomial multiplications, rotations, and modular reductions, which are computationally expensive in software-only implementations.

To overcome these challenges, we present **Aurva**, a reconfigurable fabric based kernel designed for CNN inference on CKKS-encrypted inputs. Our work focuses on the convolutional unit, the most critical component for accelerating encrypted inference. The kernel interprets CKKS ciphertexts as polynomials, applies an NTT transformation for efficient arithmetic, and executes homomorphic multiply–accumulate operations in a pipelined hardware flow. In doing so, we exploit both the algebraic structure of CKKS and the parallelism of FPGA architectures to achieve scalable encrypted convolution.

Parameters Used in This Work

For our implementation, we adopt the following parameters:

- **N = 256**: Degree of the polynomial ring, corresponding to the number of slots available for SIMD packing in a single ciphertext (**N/2**).
- **Q = 132,120,577**: Ciphertext modulus used for all modular arithmetic operations, ensuring correctness of the NTT and homomorphic computations.
- **K = 27**: Bit-width of the modulus Q, indicating that all coefficients are represented using 27 bits.
- **barret_const = 136,348,167**: Precomputed constant used in Barrett reduction to efficiently perform modular multiplications without explicit division.
- **Image Size = 16 × 8**: Each ciphertext encodes a 16x8 image in its SIMD slots.
- **Kernel Size = 2 × 2**: Convolution kernel applied homomorphically over the encrypted image slots.

These parameters are chosen to simplify the POC demonstration of accelerating convolution layers of CNN over CKKS encrypted data.

2 Background

2.1 Homomorphic Encryption and CKKS

Fully Homomorphic Encryption (FHE) enables computations on encrypted data without exposing the underlying plaintext. The CKKS scheme is particularly suited to machine learning workloads, as it allows approximate arithmetic over real and complex numbers. In CKKS, a vector of values is encoded into a polynomial of degree N, which is then encrypted into a ciphertext (c_0, c_1).

A key feature of CKKS is its SIMD capability: up to $N/2$ plaintext slots can be embedded into a single ciphertext. This makes it possible to apply the same homomorphic operation across many values in parallel, aligning well with the structure of CNN computations.

2.2 NTT and Polynomial Arithmetic

Operations involving ciphertexts (hereafter referred to as **CT**) and plaintexts (hereafter referred to as **PT**)—such as CT–PT addition, CT–PT multiplication, and CT–CT multiplication—essentially reduce to one or more polynomial multiplications.

The standard algorithm for polynomial multiplication has a time complexity of $O(N^2)$, which becomes inefficient for large-degree polynomials ($N = 256, 512, 1024$, etc.). Hence, we transform the polynomials into the **NTT domain** by applying the Number Theoretic Transform (NTT).

It is useful to think of the NTT as the *ring-theoretic analogue* of the Fourier Transform, which is used to convert signals into the frequency domain. To formalize:

Let q be a prime such that $q \equiv 1 \pmod{2N}$. Then there exists a primitive N-th root of unity $\omega \in \mathbb{Z}_q$ such that $\omega^N \equiv 1 \pmod{q}$ and $\omega^k \not\equiv 1 \pmod{q}$ for $0 < k < N$.

Given a polynomial

$$a(x) = \sum_{j=0}^{N-1} a_j x^j \quad \text{with coefficients } a_j \in \mathbb{Z}_q,$$

the Number Theoretic Transform (NTT) of $a(x)$ is defined as

$$A_k = \sum_{j=0}^{N-1} a_j \, \omega^{jk} \pmod{q}, \quad k = 0, 1, \ldots, N-1.$$

The inverse NTT is given by

$$a_j = N^{-1} \sum_{k=0}^{N-1} A_k \, \omega^{-jk} \pmod{q}, \quad j = 0, 1, \ldots, N-1,$$

where N^{-1} is the modular multiplicative inverse of N modulo q.

2.3 CKKS Slot Rotations

Many algorithms (e.g., convolutions, dot products, pooling) require moving data between slots. CKKS supports this through *slot rotations*, which cyclically shift the contents of the ciphertext. If a ciphertext ct encodes the vector

$$(x_0, x_1, x_2, \ldots, x_{N/2-1}),$$

then a rotation by k slots yields another ciphertext ct' encoding

$$(x_k, x_{k+1}, \ldots, x_{N/2-1}, x_0, x_1, \ldots, x_{k-1}),$$

i.e., a cyclic shift of the vector entries. By applying the automorphism:

$$X \to X^g \pmod{X^N + 1}$$

where g is an odd integer modulo $2N$, we can permute the slots of CKKS ciphertext in a structured way.

In NTT domain, this automorphism manifests as a **permutation of frequency components.** Since the INTT maps the polynomial coefficients into evaluations at roots of unity, applying an automorphism $X \to X^g$ corresponds to reindexing those evaluation points.

Input: Ciphertext polynomial $c(X) = \sum_{i=0}^{N-1} c_i X^i$, automorphism index g with $\gcd(g, 2N) = 1$
Output: Permuted ciphertext polynomial $c'(X)$
for $i \leftarrow 0$ **to** $N - 1$ **do**
  $j \leftarrow (i \cdot g) \bmod N$;
  if $(i \cdot g) \bmod 2N < N$ **then**
    $s \leftarrow +1$;
  else
    $s \leftarrow -1$;
  end
  $c'_j \leftarrow (s \cdot c_i) \bmod q$;
end
return $c'(X)$;

Algorithm 1: Automorphism on CKKS Ciphertext

Applying an automorphism also transforms the underlying secret key. Specifically, if a ciphertext $c = (c_0, c_1)$ is originally valid with respect to the secret key s, then after applying the automorphism $\tau_g : X \mapsto X^g$, the resulting ciphertext becomes valid under the transformed key $\tau_g(s)$. Consequently, decryption now requires the transformed secret key rather than the original s.

To maintain a consistent key basis for further homomorphic operations, an additional step—**key switching**—is required. Key switching leverages precomputed evaluation keys to convert the ciphertext back to a form decryptable under the original secret key s, thereby preserving correctness and enabling seamless continuation of computations.

2.4 Encrypted Convolutions

Convolutions in CNNs are inherently multiply–accumulate operations performed over sliding input windows. Under CKKS encryption, these operations are realized through **ciphertext–plaintext multiplications** (for weights) and **ciphertext rotations** (for aligning slots). By encoding multiple activations across ciphertext slots, the **SIMD** property of CKKS is directly exploited to perform multiple convolution steps in parallel.

However, implementing convolutions in the encrypted domain is non-trivial. Each ciphertext–plaintext multiplication corresponds to a **polynomial multiplication** in the underlying ring $\mathbb{Z}_q[X]/(X^N + 1)$. Furthermore, ciphertext rotations are carried out through **automorphisms** of the polynomial ring, requiring additional **key-switching** operations. As a result, even a single convolutional layer translates into a large number of **polynomial multiplications**, **additions**, and **modular reductions**—operations that dominate the runtime in software implementations.

To mitigate these costs, **Aurva** accelerates encrypted convolutions by mapping them to a **pipelined NTT-based architecture**. The kernel first transforms ciphertext and weight polynomials into the **NTT domain**, where polynomial multiplication reduce to **pointwise multiplications**. It then applies **modular accumulation** and efficiently orchestrates **slot rotations**. This hardware design not only reduces the asymptotic complexity of **polynomial arithmetic** but also exploits **FPGA-level parallelism** to overlap computation and memory access, thereby delivering scalable throughput for CKKS-compatible CNNs.

3 Aurva Architecture Elements

Aurva is an accelerator designed to accelerate the inference of CNN over CKKS encrypted data by utilizing the parallelism of FPGA boards to accelerate convolutions performed on the encrypted image.

3.1 AXI Integration

Aurva leverages the **AXI4** protocol family to sustain high-throughput communication between the host processor and the kernel:

- **AXI4-Lite** is used for lightweight configuration and control such as signalling execution start and execution finish.
- **AXI4-Memory Mapped (AXI4-MM)** enables direct access to external DRAM for staging encrypted images, which are then loaded into on-chip BRAMs for fast access.

3.2 Pipelined NTT Engines

Two deeply pipelined **NTT engines** running in parallel handle the conversion of ciphertext polynomials into the NTT domain, where polynomial multiplication reduces to coefficient-wise multiplication.
Each engine is composed of two **butterfly units** that implement the Cooley–Tukey (CT) NTT/FFT algorithm. The butterfly units are fully pipelined with no stalls and are capable of outputting four transformed coefficients every clock cycle, ensuring continuous data flow through the convolution pipeline.

To ensure conflict free BRAM memory mapping of polynomials, two pairs of BRAMS (Four total) are used to store each polynomial. A ping pong buffering is implemented between the two pairs to ensure no read/write conflicts occur between the NTT stages. Initially, all even index coefficients are mapped to bram index 0, and all odd index coefficients are mapped to bram index 1. The other pair of BRAM is left empty at initialization.

The roots are unity used for performing NTT are precomputed and stored in a Dual Port BRAM.

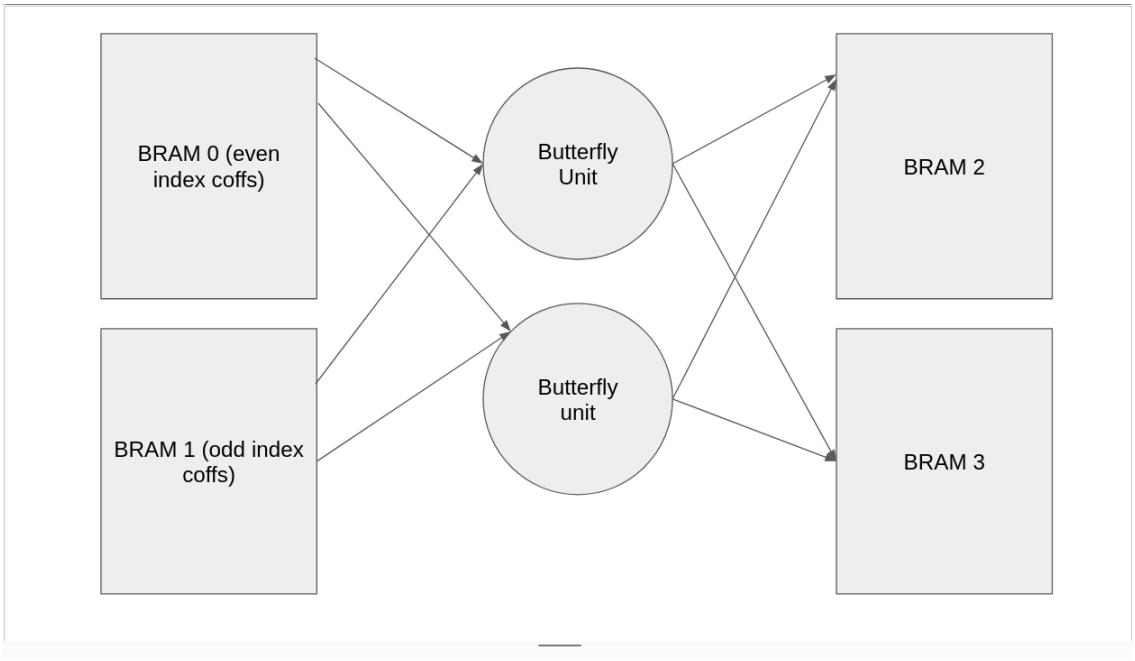

Figure 1: Ping Pong Buffer Memory Access inside NTT Engine

3.3 Modular Arithmetic units

In the **CKKS** scheme, both plaintexts and ciphertexts are represented as polynomials in a cyclotomic ring:

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

where q is the ciphertext modulus and N is the ring dimension. Thus, a plaintext is an element of R_t for some smaller modulus t, while a ciphertext resides in $R_q \times R_q$. Hence, all operations performed in this space must be performed modulus q. Modulus operations are often expensive in the hardware domain. Particularly, mod multiplications

$$a = (b * c) \mod c$$

are especially resource intensive and slow. To optimize modular multiplications, we use the barret reduction algorithm

Input: Integer x, modulus q, precomputed constant $\mu = \lfloor 2^k/q \rfloor$ with $k \geq 2 \cdot \lceil \log_2 q \rceil$
Output: $r = x \bmod q$
$q_1 \leftarrow \lfloor x/2^{k-1} \rfloor$;
$q_2 \leftarrow q_1 \cdot \mu$;
$q_3 \leftarrow \lfloor q_2/2^{k+1} \rfloor$;
$r \leftarrow x - q_3 \cdot q$;
if $r \geq q$ **then**
  $r \leftarrow r - q$;
end
if $r < 0$ **then**
  $r \leftarrow r + q$;
end
return r;

Algorithm 2: Barrett Reduction

Barrett reduction avoids explicit division by q, replacing it with multiplications and shifts.

Additionally, modular additions and modular subtraction units are implemented. All modular arithmetic units are fully pipelined to support continuous throughput.

3.4 SIMD Property and Galois Automorphism

To rearrange slots in CKKS encrypted CT, we use **Galois automorphisms**, which permute the ciphertext in a structured way. Conceptually, applying an automorphism corresponds to a *cyclic rotation of the encoded vector*. This operation is critical in applications such as convolutions, dot products, and pooling, where data needs to be shifted between slots.

Input: Ciphertext polynomial $c(X) = \sum_{i=0}^{N-1} c_i X^i$, automorphism index g
Output: Permuted ciphertext polynomial $c'(X)$
for $i \leftarrow 0$ **to** $N-1$ **do**
  $j \leftarrow (i \cdot g) \bmod N$;
  if $(i \cdot g) \bmod 2N < N$ **then**
    $s \leftarrow +1$;
  else
    $s \leftarrow -1$;
  end
  $c'_j \leftarrow (s \cdot c_i) \bmod q$;
end
return $c'(X)$;

Algorithm 3: Automorphism on CKKS Ciphertext

Hardware Architecture. In our implementation, we focus on efficiently realizing this permutation step in hardware. Since polynomial coefficients are stored in BRAM, we fetch **4 coefficients at a time** (two dual-port BRAMs). Each fetched group is then routed to its new permuted position according to the automorphism index. The Galois index calculation can be done dynamically, but for fixed problem sizes we precompute the required rotations, which saves logic and latency. For instance, for a 16×8 image, only -1, -16, and -17 rotations are needed, so the index mapping is stored and directly applied in hardware.

This architecture balances flexibility (on-the-fly index computation) with efficiency (precomputed indices for fixed workloads), enabling high-throughput slot rotations as part of the overall CKKS pipeline.

3.5 Key Switching

Key switching is a crucial step in CKKS whenever an automorphism or a ciphertext multiplication changes the underlying secret key basis. After such operations, a ciphertext $c = (c_0, c_1)$ may be decryptable under a transformed key s' instead of the original key s. The goal of key switching is to produce a new ciphertext $c' = (c'_0, c'_1)$ such that all further operations remain valid under the original secret key s:

$$\langle c', s \rangle \equiv \langle c, s' \rangle \pmod{q}.$$

This transformation uses **evaluation keys** that encode the relationship between s' and s. In hardware, we implement key switching in two main steps:

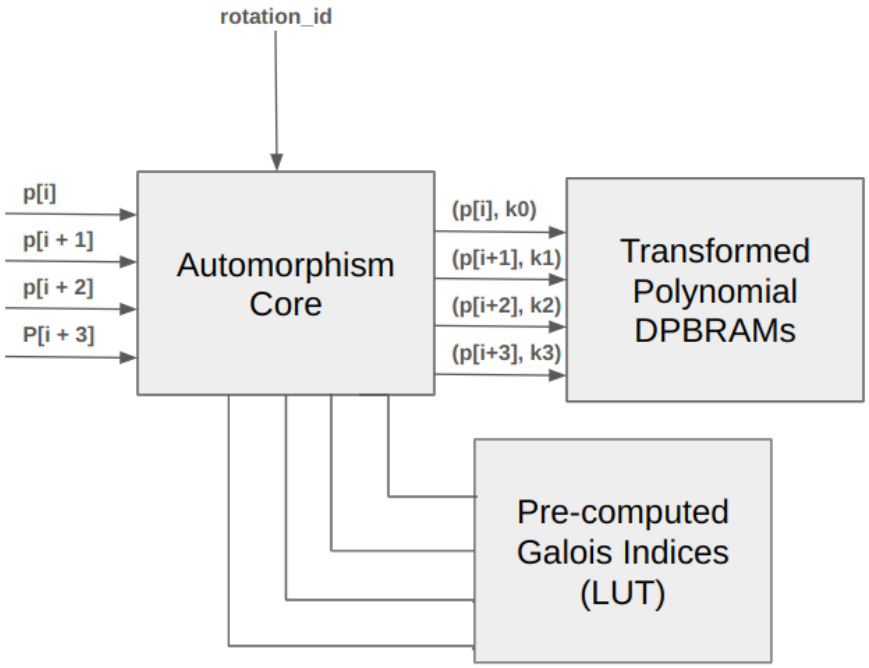

Figure 2: Automorphism coefficient mapping in NTT domain

1. **Pointwise multiplication:** Multiply c_1 with the evaluation key shares $(\text{evk}_0, \text{evk}_1)$ to produce intermediate values p_0 and p_1.
2. **Recombination:** Add p_0 to c_0 to obtain c'_0, and set $c'_1 = p_1$.

Input: Ciphertext $c = (c_0, c_1)$, evaluation key $(\text{evk}_0, \text{evk}_1)$
Output: Ciphertext $c' = (c'_0, c'_1)$
$p_0 \leftarrow c_1 \cdot \text{evk}_0 \pmod q$;
$p_1 \leftarrow c_1 \cdot \text{evk}_1 \pmod q$;
$c'_0 \leftarrow c_0 + p_0 \pmod q$;
$c'_1 \leftarrow p_1$;
return (c'_0, c'_1);

Algorithm 4: Key Switching

Hardware Implementation In our convolution unit, rotated ciphertexts need to be accumulated across multiple slots. Since each automorphism changes the effective secret key, direct addition of two rotated ciphertexts would be invalid without key switching. Our architecture performs key switching before any accumulation or multiplication with kernel polynomials. For fixed image sizes, associated evaluation keys, minimizing runtime overhead and ensuring continuous throughput in the pipeline.
Evaluation keys are provided by the client; in our setup, they consist of encrypted and rotated versions of the secret key. This allows the hardware kernel to operate on rotated ciphertexts while preserving correct decryption under the original key.

3.6 Convolution Unit

The convolution unit operates on a ciphertext $c = (c_0, c_1)$. Under our parameterization, a single ciphertext can encode a 16×8 image across its SIMD slots. The convolution process is performed homomorphically as follows: the ciphertext is first rotated to align the desired kernel with the corresponding image block. Next, a ciphertext–plaintext multiplication is performed to apply the kernel to all aligned slots simultaneously. Finally, the results of multiple rotations are accumulated via ciphertext–ciphertext additions to produce the final encrypted convolution output. .
Key switching is performed prior to ciphertext–plaintext multiplication with the kernels. Performing key switching after other homomorphic operations can adversely affect noise growth and compromise correctness; therefore, it is preferable to apply key switching before multiplication to maintain proper noise management.

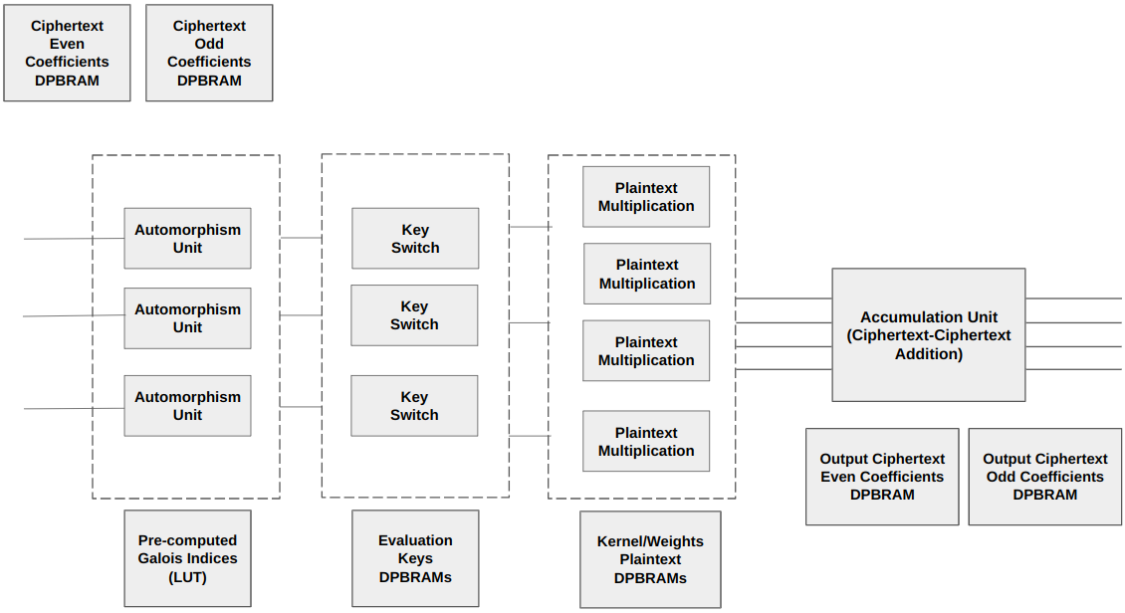

Figure 3: Convolution Unit

4 Implementation

- **Hardware Platform Overview** The project was implemented on the AMD Kria™ KV260 Vision AI Starter Kit. This platform is a two-part system composed of a Kria K26 System-on-Module (SOM) and a carrier card. The carrier card is mostly utilized for vision centric applications, which is out of scope for the Proof-of-Concept product at the time of submission.

- **Main Processing Unit:** KV260 has the Zynq Ultrascale+ MPSoC which combines two processing domains:

  – **Processing System (PS)**: a quad-core Arm® Cortex®-A53 Application Processing Unit (APU) and a dual-core Arm Cortex-R5F Real-Time Processing Unit (RPU). The PS runs the operating system and manages high-level application logic. We used a MicroSD card to boot the Ubuntu 24.04 LTS for Xilinx KV260 Vision Starter.

  - **Programmable Logic (PL)**: FPGA fabric which contains logic cells, BRAMs and DSP slices. It was used to run the synthesized hardware kernel.

- **Vitis Kernel Flow**: Vitis RTL Kernel flow was used to synthesize and run the hardware on the FPGA Fabric. The design was synthesized using Vivado 2025. s

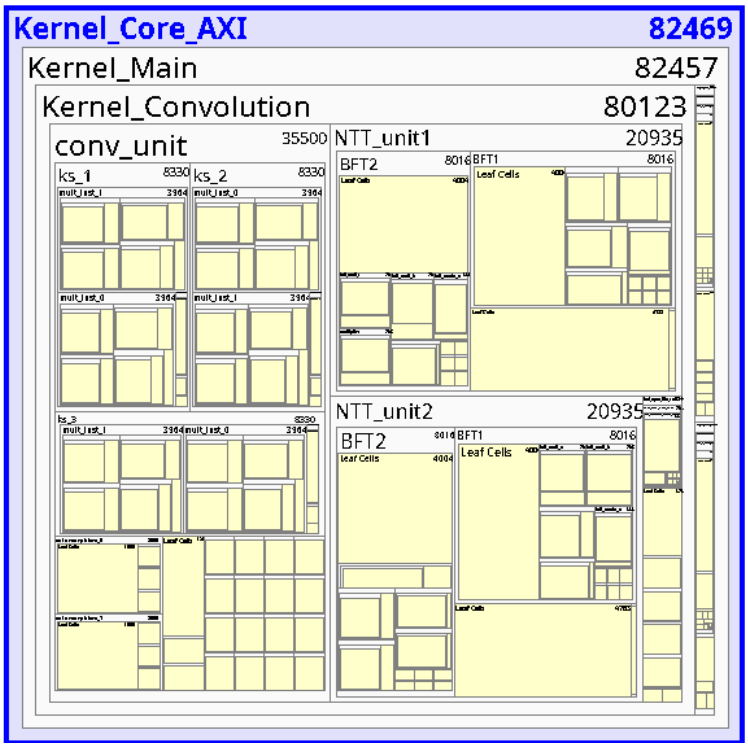

Figure 4: Overall architecture

5 Results

The **Aurva** kernel was implemented on the AMD Kria™ KV260 FPGA platform and evaluated for CKKS-encrypted CNN convolutions. Using our parameter set ($N = 256$, $Q = 132{,}120{,}577$, $K = 27$), the kernel achieves deeply pipelined convolution accelerator engine.

Resource	**Used**	**Available**	**Utilization (%)**
CLB LUTs	13,335	117,120	11.38
CLB Registers	33,927	234,240	14.48
Block RAM	80	144	55.55
DSP48E2	274	1248	21.95
Clock Freq. (MHz)	150		
Latency (cycles)	1340		

Table 1: Resource utilization and performance of `AurvaKernel` on `xck26-sfvc784-2LV-c`.

Operation	Clock Cycles
NTT / INTT	1000
Convolution Layer	340
Total	1340

Table 2: Clock cycle breakdown of `AurvaKernel` for NTT/INTT and convolution operations.

Compared to a software implementation of the same NTT/INTT operations running on an Intel i7 13^{th} Gen processor (with the NTT written in optimized C++), our FPGA-based kernel achieves a **25× speed-up** in throughput.

Considering the time to transfer data to and from the kernel over AXI channels, we obtain a speed up of **2**×. This demonstrates the effectiveness of mapping polynomial arithmetic to a fully pipelined FPGA architecture while exploiting precomputed roots of unity for accelerated NTT computations.

6 Past Literature

- **Florian Krieger et al. – Client-Side Acceleration:** Last year's **AMD Open Hardware** submission by a team from the **Institute of Information Security, Graz University of Technology, Austria** (**PhD Category**) focused on accelerating client-side operations of the CKKS FHE scheme, i.e., encoding/decoding and encryption/decryption. In contrast, our work focuses on accelerating cloud-side operations, with a particular emphasis on Convolution acceleration for CNN inference.
- **Cheon et al. – Channel-by-Channel Convolution:** The original authors of the CKKS scheme proposed channel-by-channel packing, encoding a single image over multiple ciphertexts. In our proof-of-concept, a single image of size 8×16 is encoded into a single ciphertext. Additionally, Cheon et al. did not explore hardware-level implementations of their proposed software architecture.
- **Xiao Hu et al. – Efficient Hardware Architectures for Convolution Units:** Their design leverages **reused rotations** to reduce key-switching overhead on the cloud side, at the cost of increased client-side computation. While our work reduces client-side computations by performing all required operations in the kernel deployed on the cloud.

7 Future Development

To make the system viable for larger-scale CNN applications, potential future developments include:

- **Enhanced security parameters:** Current parameters are smaller than typical LWE standards used in research and industry. Future work will involve increasing security parameters and optimizing the architecture for these enhanced settings.
- **RNS Variant:** Incorporating the CKKS-RNS variant would enable additional homomorphic operations, such as rescaling, multiplication, and bootstrapping. Our current design has been structured to accommodate this transition with minimal modifications.

– **Expansion to a full CNN:** Presently, the architecture is not capable of implementing Non Linear functions. Such functions are highly used as activation function before each fully connected layer. However, CKKS operates exclusively over polynomial rings and hence, polynomial approximations of these non-linear functions must be developed and implemented in hardware.
– **CT-CT Multiplication**: Implementing homomorphic multiplication between two ciphertexts will allow us to store the model parameters (weights,biases etc) in their encrypted form as well to enhance security of models against adversarial attacks.

References

[1] Florian Krieger, Florian Hirner, Ahmet Can Mert, and Sujoy Sinha Roy, "Aloha-HE: A Low-Area Hardware Accelerator for Client-Side Operations in Homomorphic Encryption," in *2024 Design, Automation Test in Europe Conference (DATE)*, 2024, pp. 1–6, doi: 10.23919/DATE58400.2024.10546608.

[2] Xiao Hu, Minghao Li, Jing Tian, and Zhongfeng Wang, "Efficient Homomorphic Convolution Designs on FPGA for Secure Inference," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 30, no. 11, pp. 1691–1703, Nov. 2022, doi: 10.1109/TVLSI.2022.3216592.

[3] Xiao Hu, Minghao Li, Jing Tian, and Zhongfeng Wang, "Efficient Homomorphic Convolution Designs on FPGA for Secure Inference," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 30, no. 11, pp. 1691–1703, Nov. 2022, doi: 10.1109/TVLSI.2022.3216592.