Fall 2024, 6.205 Final Project

CIA: Ciphered Inference Accelerator

Team 27 - Shreya Chaudhary (*shreyach*) and Ruth Lu (*ruthluvu*)

Contents

Project Overview - Motivation

Someone has a cool machine learning model!

You want to send them data... but you don't trust them :(

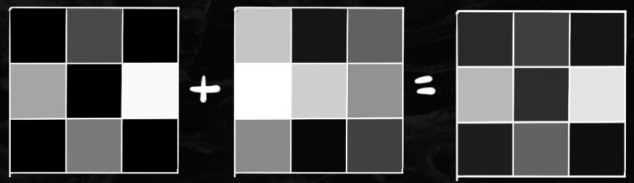
CIA (Ciphered Inference Accelerator) to the rescue!

Project overview

- Build a system that can correctly and quickly **encrypt** using general learning with errors (GLWE), GLWE **decrypt**, and run **inference** on a ciphertext using fully homomorphic encryption (FHE)

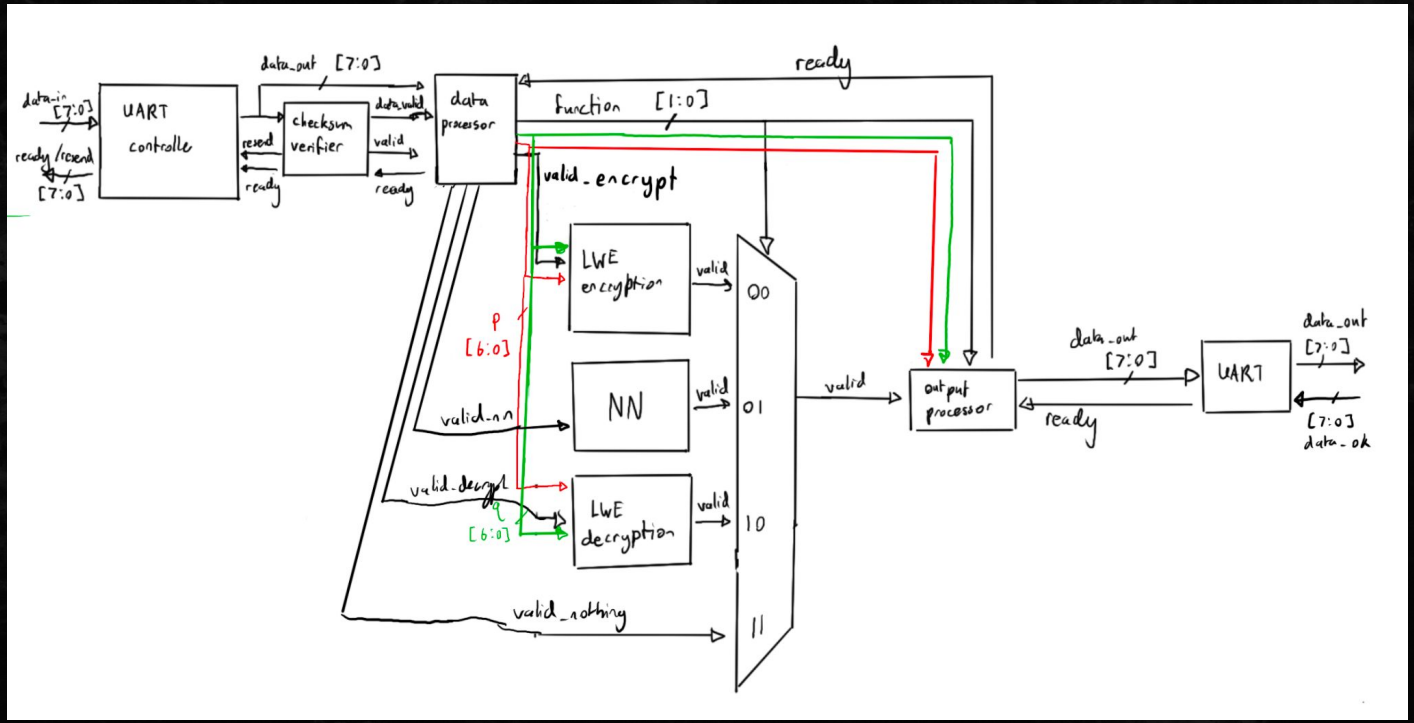
What is GLWE / FHE?

GLWE: Encryption method that adds random noise to the ciphertext (solving noisy linear equations)

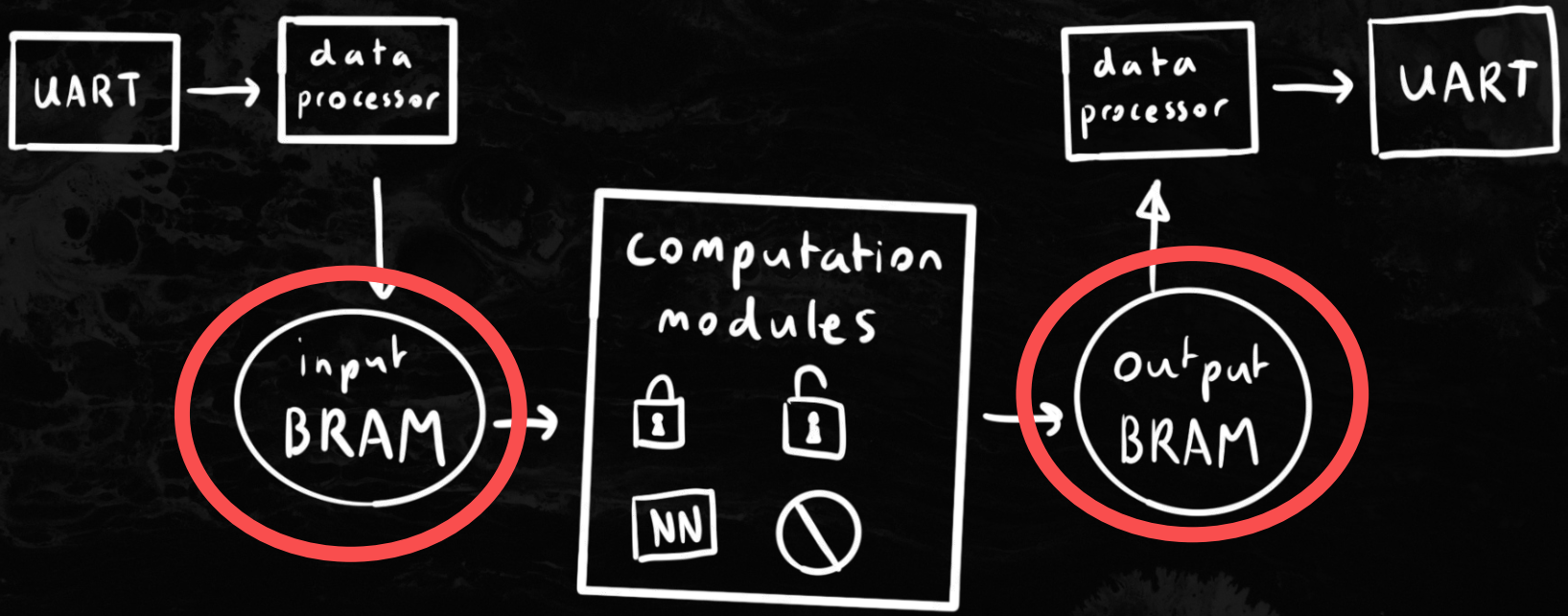


FHE: Allows us to perform addition and multiplication on encrypted data

Block Diagram

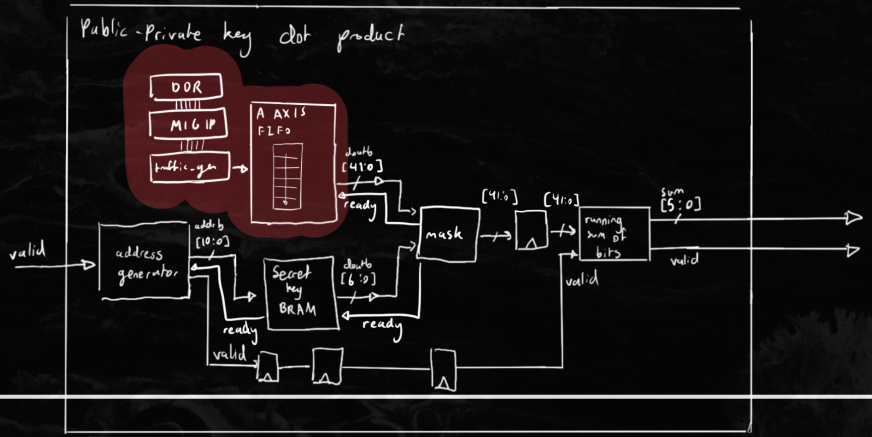


Original Overall Data Flow



Details - How we'll move data

- Almost all data / outputs stored in BRAMs
  - Modules will send when BRAMs are ready to other modules
- **Exception for ciphertext and A (public key)**
  - **Will use DRAM with FIFO buffer (similar to lab06)**



GLWE Encryption/Decryption - Theory

- **Idea: Noisy Matrix Multiplication**
  - ct = (A, m*q/p + A*s + e)
  - pt = ROUND((m - A*s)/(q/p))
- FPGA Advantage
  - q, p powers of 2 → can round with a comparison, all divisions are bit shifts
  - Vector operations parallelizable
- Hyperparameters: N (size of message), p, q (powers of 2, security parameter), k (security parameter)
  - **N = 784, k = 1000, p = 16, q = 64**
  - Size of A: 1000*784 numbers, each 6 bits = **4.7 Mbits**
  - Size of ct: size(A) + 784 * 6 bits = **4.7 Mbits**

GLWE Enc/Dec - Implementation

Matrix Multiplication

- A will be stored in DRAM with BRAM buffers around it
- BRAM output buffer will store 112 “rows” of A at once to allow parallelization
- A is less than 1 MB, DRAM can read/write 100 MB/s on average - Read time for A will be around 0.1s
- Outputting ciphertext will also need to be done with a BRAM buffer (+0.1s for writing)

Adding Error/Noise

- Sampling from a few chained LFSRs

$$CT = PT + A*s + e$$

FHE - Theory

- **Idea: Add and Multiply Directly on Ciphertexts (and while the noise is low, we can still decrypt)**
- **Addition**: Adding the ciphertexts directly, add delta*c for constant addition
- **Multiplication**: Multiplication works by scaling the ciphertext
  - *Note: this only works for small-magnitude constants/ciphertexts! To guarantee correctness, we need to add bootstrapping to make sure we don't have too much noise!*
- FPGA Advantage
  - Vector operations, can be parallelized

Secure Inference NN

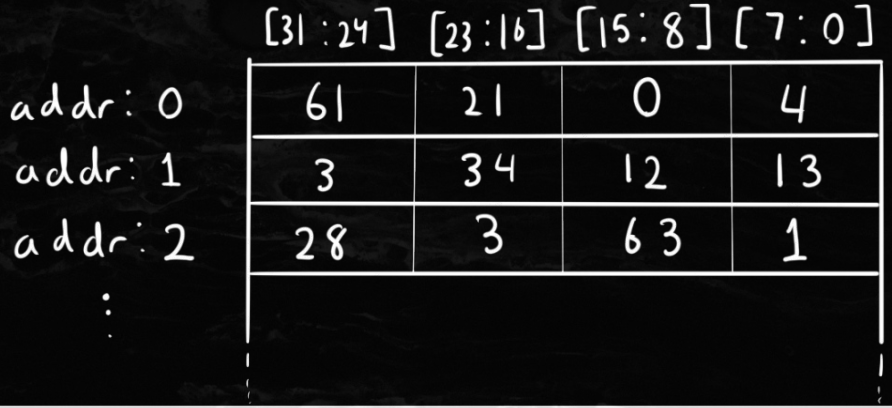
- Similar change to enc/dec to handle large ciphertext
  - Nodes for first layer (ciphertext) will need to be stored on DRAM
  - Nodes for all other layers can be stored in BRAM
- Simple fully connected neural net with no activation functions
- Weights are added with matrix multiplication, biases are added with vector addition
- Opportunities for parallelization at both!

Bootstrapping

- **Idea: Reduce the noise after some computations by encrypting the decryption information**
- Problem: Generally the bottleneck of FHE
- Implemented **gate bootstrapping** which effectively scales down the values on specific operations to reduce the noise
- Also implemented **modulus switching, sample extraction, and blind rotation** (vector operations, matrix multiplication, and moves) for a more general version of bootstrapping

Achieving Parallelization

- Since matrix operations rely on nearby data, we will store/load many values into the same address in BRAM
- When we pull one address out of BRAM, we can do calculations on many values in parallel



Memory Calculations

BRAM Usage

- Input buffer for A (public key): 336 kbits
- Output buffer for A (public key): 336 kbits
- Input buffer for CT: 336 kbits
- Output buffer for CT: 336 kbits
- Secret/switching key: 784 kbits
- Neural net weights and biases: 129 kbits
- **Total: ~2257 kbits / 2700 available kbits used of BRAM**

Plans - Benchmarking Performance

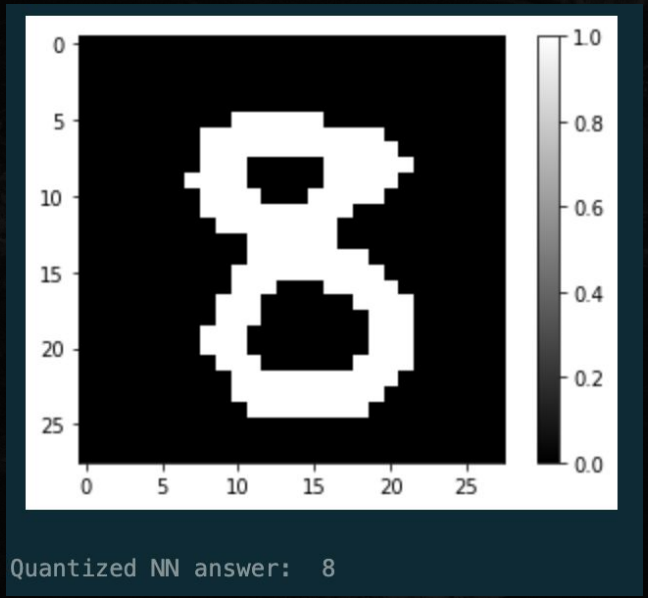
- Input data processor will start counting clock cycles as soon as it has loaded everything into memory
- Compare against Shreya's CPU time taken for that computation using Python's own timing library
  - Preliminary timing: ~2m30s for encryption and decryption

CPU Code is Ready!!

GLWE Encoding/Decoding +
Simple Linear Regression

```
inp: [ 5 14  4  5]
eq: 2x + 1
pt computation: [11 13  9 11]
ct computation: [11. 13.  9. 11.]
```

8-bit Quantized NN



Stretch Goals

Communication between FPGAs - Demonstration Purposes

- Encrypt on FPGA 1, send to FPGA 2
- Secure inference on FPGA 2, send result to FPGA 1
- Decrypt on FPGA 1
- Simulates what a real system might behave like

Further Optimization - Can we beat Numpy?

- Numpy is highly optimized CPU code
- Can we take advantage of more parallelization to do better than a Numpy-based FHE NN?

Final boss →

Project Timeline

	Shreya	**Ruth**
11/3	Implement CPU FHE	Quantize NN
11/10	Top Level & UART Communications	Public-Secret Matrix Multiplication
11/17	Encryption / Decryption LWE	FHE Matrix Multiplication
11/24	Bootstrapping	FPGA NN implementation

Combined:

- Determine correctness of inference (11/27)
- Benchmark timing and further optimize (12/1)
- Messaging between FPGAs? (12/8)

Checkoff

- **Commitment**
  - Input and output data with UART and store data on BRAM/DRAM
  - **Decrypt and encrypt** data with GLWE
  - Perform a simple **linear regression**
- **Goal**
  - Add **bootstrapping** modules
  - Conduct inference with a quantized **neural network** [replaces the linear regression module]
  - Make the code **2x faster than unoptimized CPU code**
- **Stretch Goal**
  - Use **two separate FPGAs** (one for encrypting/decrypting and the other for the neural network computation)
  - Make the code **2x faster than NumPy**

Thanks for listening!
Questions?