6.2050 Final Project Block Diagram
Team: Ruth Lu (ruthluvu)*, Shreya Chaudhary (shreyach)*
Working Title: Ciphertext Inference Accelerator (CIA): FPGA Accelerator for Fully Homomorphic Encryption for Inference Models - Block Diagram

1 System Overview

Our system, the Ciphertext Inference Accelerator (CIA), will use threshold fully homomorphic encryption (TFHE) to perform inference tasks on data encoded with learning with error (LWE, specifically general LWE or GLWE). Our goal is to accelerate secure inference tasks by using the FPGA. For our system, we will be building a system that can perform encryption, decryption, and a simple MNIST classifier on an encrypted image.

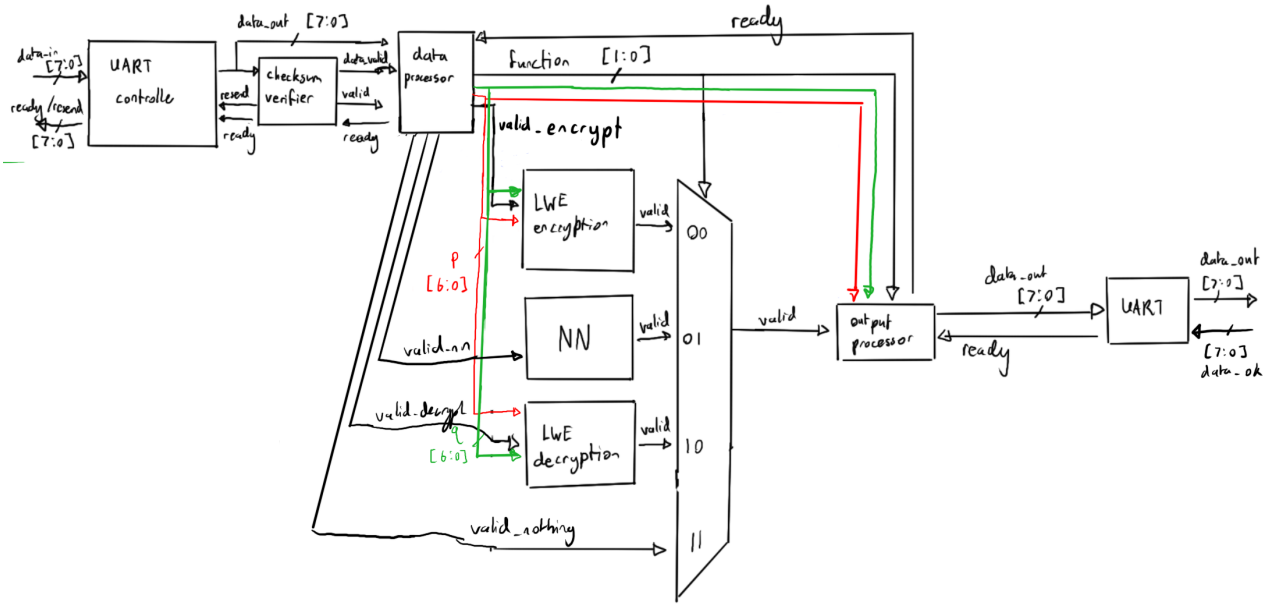
The core modules for FHE inference will include:

- A UART controller for communication with outside parties
- GLWE encryption and decryption
  - Random number generator to add small gaussian errors (ϵ) to ciphertext
  - Public-secret key multiplication module
- Inference / NN
  - FHE Matrix multiplication module
  - Bootstrapping module

Our system will do the following: an external device can send it a piece of unencrypted data and a public/secret key, and our system will encode the data and store the ciphertext. The external device can also send it encrypted input, and the FPGA will run it through an inference model, then send the encrypted output back to the device. The external device can also send it an encrypted input, and the FPGA will decrypt it. Finally, the FPGA can just send back what the external device sent it.

We chose to make these 4 separate data flows in our system to model potential real world use cases. It would be unlikely for a user to ask a device to encrypt, run inference, and decrypt a piece of plaintext data since that would defeat the point of ciphered inference. A system will likely only perform one of the three. The option for the FPGA to send the same data back is for debugging purposes.

2 Block Diagram



2.1 High Level Details

The UART module will receive data from the computer. It can also send back its own message to let the computer know to resend (in case checksum shows data was corrupted) or to tell the computer that the system is busy and cannot take more data.

The data flow is as follows: when the UART module receives data, the first two bits will encode the function (encryption, inference, decryption, or nothing). Using this information, the data processor will save all following information into the necessary BRAMs (plaintext BRAM, ciphertext BRAM, public key BRAM, or secret key BRAM) and registers. Once the data processor has saved everything into registers and has received a valid signal from the checksum, it sends out a valid signal to let one module know to start computation, and output processor will inform the data processor that it is no longer ready to accept inputs.

A valid signal is sent to one of the 3 core computation modules, then the mux selects which valid output the output processor pays attention to. Once output processor sees a valid signal, it will send the proper data (determined by the function implemented) to the UART. Once it is done sending (and the computer verifies it with checksum), it will let data

processor know it is ready, which will remove the backpressure on the system so the UART receiver will start being able to receive data and pass it to the data processor again.

2.2 IPs needed

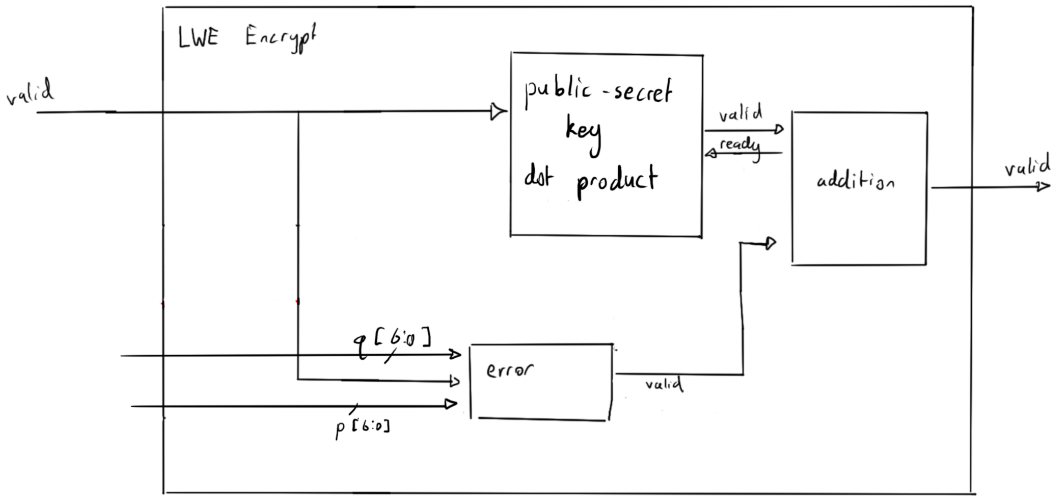
- BRAM interface - We will store the client's data on the BRAM as we expect the size of it to be less than 1000 bits. We will also store the weights of the NN, the public key, the switching key, and the secret key.

2.3 Memory

- Storing plaintext and ciphertext. MNIST is 28 by 28 pixels, and each pixel can be described as 0 or 1. It will require 784 bits plaintext, and after encryption (if we choose q = 32), it will be about 25 kb.
- The public key, secret key, and switching key (used for bootstrapping) will each be 1kb. We limit k = 1000, which is secure as it is larger than the number of bits being encrypted, preventing some attacks, but not overly large (which will make the system slower and use more resources). These will each be stored in BRAM.
- Our planned network architecture will have an input of 784 nodes, two hidden layers with 10 nodes, and an output layer of 10 nodes. This means there will be 784*10+(10*10)*2 = 8040 weights and 8040 biases. We will be using 8-bit quantization, which means the entire neural net will take about 129 kb. (It can be stored entirely on BRAM.)

3 GLWE Module

3.1 Encryption

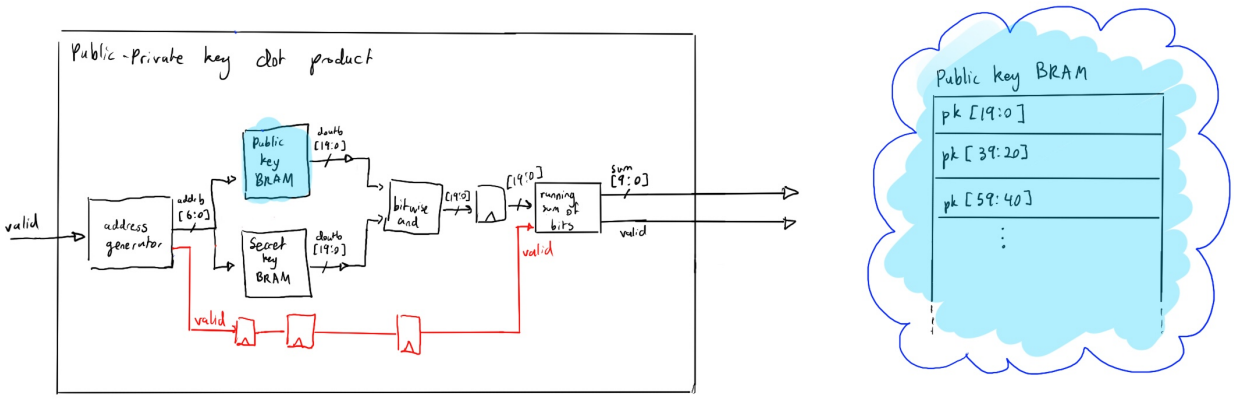


GLWE encryption is defined as $a \cdot s + \epsilon + b\lfloor \frac{q}{p} \rfloor$ where q, p are a power of 2. This will use two main modules: public-private key dot product, vector addition, and the epsilon generator.

The relevant variables are as following:

- a: The public key mod q of size 1001. This will be stored in the BRAM.
- s: The secret input by the user of size 1000. This will be stored in the BRAM.
- b: The plaintext message. This will also be stored in the BRAM.
- q, p: The parameters of the problems. This will be stored in a register and passed to the error module.

3.2 Public-secret key dot product



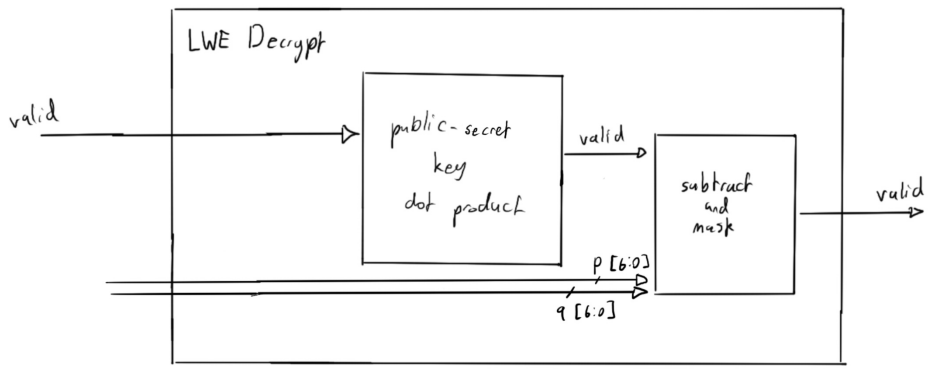
This module will perform an operation critical to both encryption and decryption with parallelization: taking the dot product of the public and secret keys, which are both binary vectors. The resulting value is added to the noised plaintext to encrypt it.

Parallelization is achieved by storing 20 bits of each key in each address of BRAM, which allows computation on all these bits to be done in the 3 clock cycles needed to load an address and perform the bitwise and. More bits could potentially be parallelized, although we would need to see if this meets timing.

3.2.1 ϵ Generator

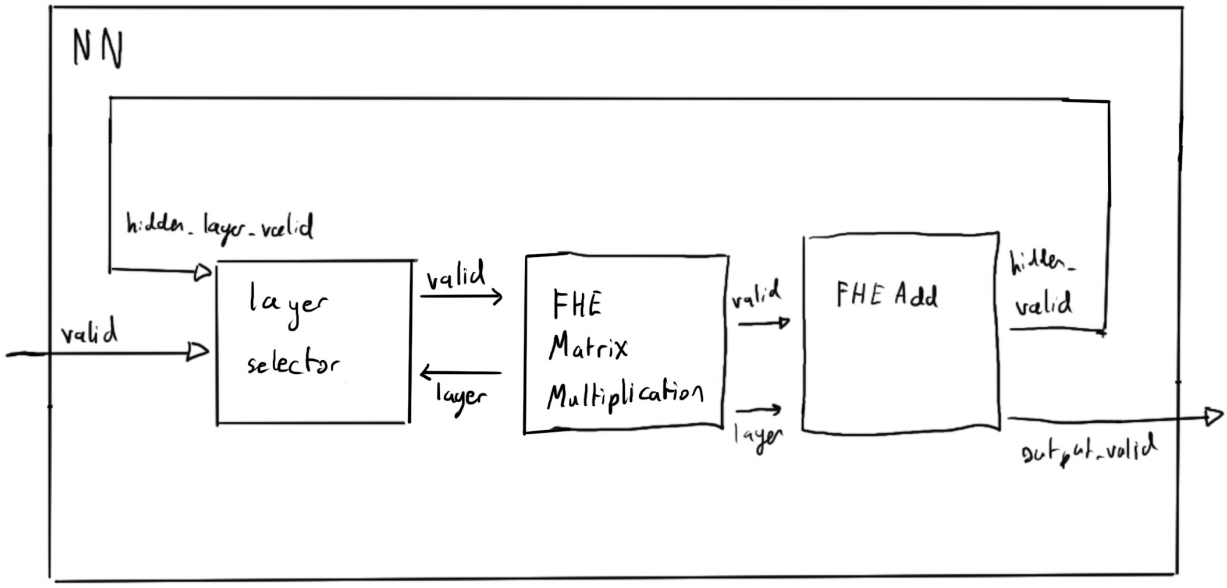
The ϵ generator will need to randomly generate a number between a given parameter [-B, B] for generally small $B << q/p$. This will initially be implemented in a similar manner as in lab (random based effectively on a timer) and later can be converted into something more secure.

3.3 Decryption



LWE decryption checks if $(b - a \cdot c)/(q/p)$. This will once again involve a public-private key dot product and vector subtraction and will use similar modules as the encryption module.

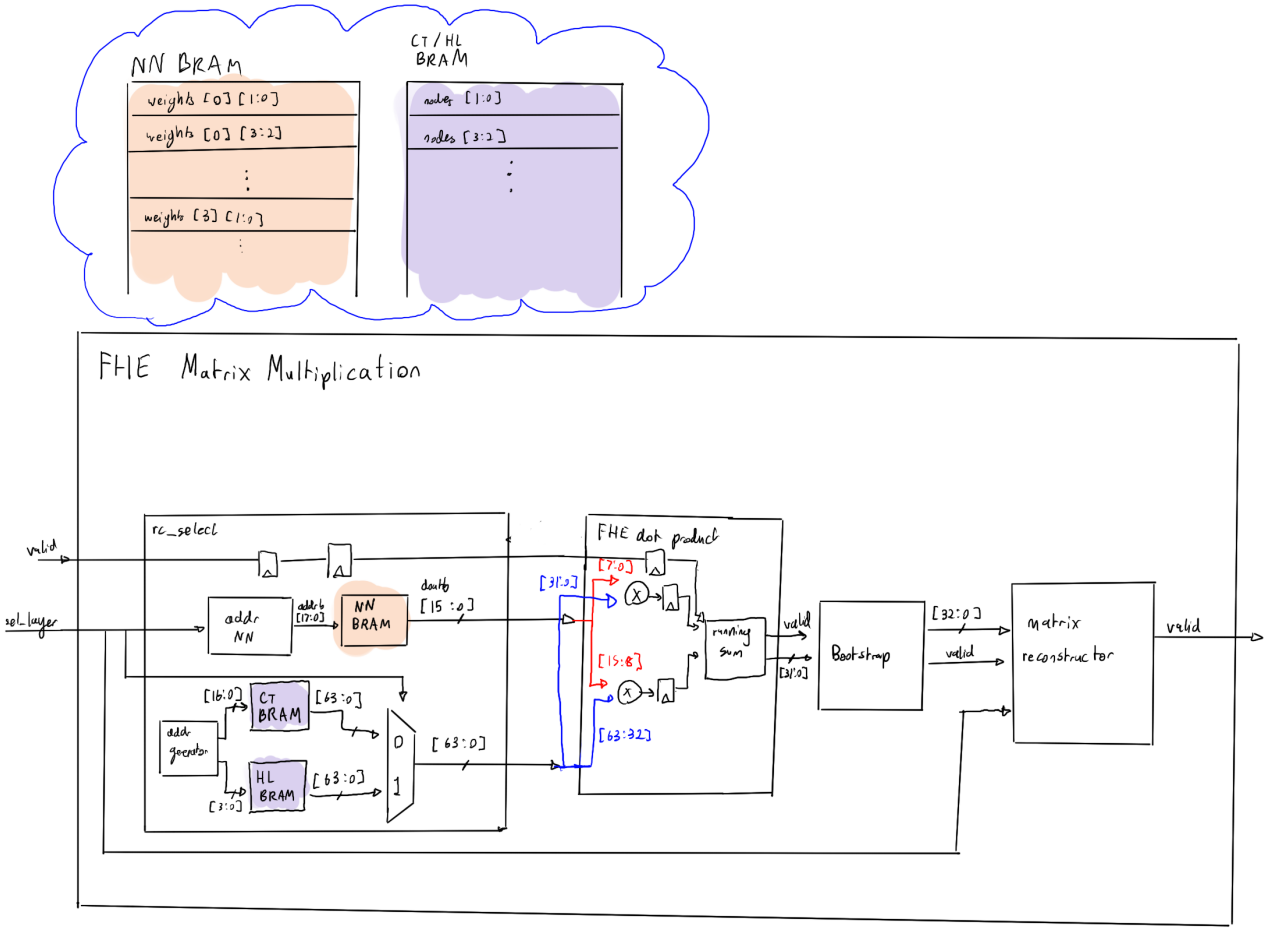
4 NN Module



This module will work with a pretrained neural network. We will then quantize all of the weights using 8-bit linear quantization, and these quantized weights will be loaded into BRAM during build. We decided to quantize these weights to make the neural network smaller without losing significant accuracy. This will allow us to run the computations on the FPGA without taking a significant amount of space to store the weights.

The main purpose of the NN module is to control the flow of data through the neural net. Layer selector will keep track of which layer the computation is currently in and pass that information to the matrix multiplication and addition modules.

4.1 FHE Matrix Multiplication



The FHE matrix multiplication module is similar to the matrix multiplication module and will take in the following inputs:

- reset - active high reset will make the module reset (not pictured)
- valid - active high valid signal indicates the module should start computation
- *sel_layer* - which layer of weights to look at (0 or 1) and selects which node BRAM to look at (0 for ciphertext BRAM, 1 for NN hidden layer BRAM)

The *rc_select* module will read one address (2 values are stored in each address) from each of the corresponding BRAMs (the NN BRAM and the node BRAM chosen by *sel_layer*). The dot product module will multiply then add them together before sending to the bootstrapping module to reduce the error. Doing 2 calculations at once adds parallelization, but makes it a bit more complex. The dot product will then be sent to the matrix reconstructor, which will store them into either the hidden layer BRAM or the NN output BRAM. When all operations are done, the module will send valid to the next module to let it know that the correct BRAM has the necessary information.

4.2 Bootstrapping

Because the error will corrupt the encrypted message after too many operations, bootstrapping is necessary to successfully decrypt the data. Effectively, bootstrapping will re-encrypts the offset necessary to decrypt, and by doing this, it can reduce the total noise of the circuit.

Bootstrapping consists of three main parts:

1. Modulus switching
2. Blind Rotation
3. Sample extraction

Generally, the bootstrapping scheme is the most complicated and time- and resource-intensive part of the FHE scheme. Each of the above modules builds upon existing matrix multiplication and vector operation modules, so we can efficiently add parallelization.

4.2.1 Modulus Switching

Modulus switching then converts the ciphertext from mod q to mod ω where ω is a power of 2 such that $p < \omega < q$. This is effectively conducted with an element-wise scaling of the original GLWE-encrypted ciphertext by round($\frac{\omega}{q}a$). This submodule will consequently utilize existing modules to perform the vector scaling.

4.2.2 Blind Rotation

It next uses blind rotation to shift all of the coefficients of the input polynomial by an encrypted amount. Effectively assuming M is a matrix of the coefficients, we are multiplying by a $X^{-\pi}$ where π is an encrypted value. This will effectively encrypt the value we need to decrypt / denoise the result.

The blind rotation algorithm will work by first decomposing the decryption value of π into powers of 2 (iterating over each bit of the value). If the ciphertext value at position i is 1, it will multiply the original message M by a rotation matrix X^{-i}. This module will then require iterative matrix multiplication. Given the previously defined modules, we can parallelize this operation.

4.2.3 Sample Extraction

Finally, sample extraction extracts the results from the blind rotation to get a new key that we can use for future operation which will yield results with a smaller noise. Effectively, sample extract can extract out the coefficients of an interim output of a scheme (for example, the output of blind rotation) to formulate a new ciphertext scheme (for example, GLWE) and will not introduce any additional noise to the ciphertext. Copying the coefficients of the public key to elsewhere should simply involve moving values from one BRAM location to another.

5 Conclusion

This design presents a correct and parallelizable implementation of FHE-secure inference. Our design lies the framework to create a correct implementation of TFHE that can securely run a neural network output. Our design tries to achieve our second goal by utilizing the FPGA's parallelization to reduce clock cycles both for the large number of computations needed for encryption/decryption and bootstrapping, providing for a faster accelerator.

There remain potential issues with the design. It is possible that the plaintext image (784 bits) needs to be stored in memory instead of registers. This would only significantly change our error module, since it would need to take the image out of memory instead of registers.

Some problems remain unresolved. We are still determining the best way to implement bootstrapping, as it is a complex computation that can take advantage of parallelization.

6 Appendix

6.1 Details about FHE Operations

6.1.1 Constant Scalar Multiplication

In TFHE, multiplying by a constant can be done by simply multiplying the ciphertext. This will be done often in matrix multiplication when multiplying by a weight.

6.1.2 Constant Addition

Addition by a constant requires encryption by a trivial LWE ciphertext (setting A and error to 0, just multiplying the addition by a constant).