Ciphered Inference Accelerator Final Report

1st Shreya Chaudhary
EECS
MIT
Cambridge, MA, US
shreyach@mit.edu

2nd Ruth Lu
EECS
MIT
Cambridge, MA, US
ruthluvu@mit.edu

***Abstract*—We present a design for an FPGA-based accelerator for encryption and decryption using learning with errors (LWE), as well as inference on MNIST digits using fully homomorphic encryption (FHE) and bootstrapping for error reduction. FHE is a cryptographic algorithm that allows operations to be performed directly on ciphertext encrypted with LWE. It has important implications for secure machine learning, allowing a machine learning algorithm to perform inference on data without the data being revealed to the algorithm. However, because it is slow, FHE is not currently used in practice. Consequently, we propose using an FPGA to accelerate using FHE on machine learning models. We specifically focus on implementing Torus FHE (TFHE) which is generally better for machine learning models, and we improve latency by exploiting parallelism in the matrix operation steps and optimizing data transfer. We achieve 26 times speedup over unoptimized CPU Python code.**

***Index Terms*—Digital systems, Field programmable gate arrays, Cryptography, Hardware security, Accelerator architectures, Matrices**

I. Introduction

Fully homomorphic encryption (FHE) is a post-quantum-secure algorithm allowing computation on encrypted data such that an adversarial algorithm can compute an output without learning any information about the input or output. FHE is an especially powerful algorithm for ensuring data privacy for inference models and can play an increasingly important role as machine learning models are used more in practice. Users can send encrypted data to an untrusted party's model for training or predictions, ensuring privacy for sensitive information like medical data. Despite these benefits, FHE is not used in practice due to its slowness: it involves many matrix computations for encryption, decryption, and operations on encrypted data. Specialized hardware, like that of an FPGA, could be used to accelerate these specific and highly parallelizable operations.

Consequently, we built an accelerator for running machine learning inference models with fully homomorphic encryption (FHE). Our accelerator uses UART to send and receive data including the public key, secret key, plaintext, and ciphertext between the computer and FPGA. We currently support encryption using LWE, decryption using LWE, and noiseless inference using a neural network for a simple MNIST prediction.

Fig. 1. The final system will perform 3 different operations, each corresponding to part of the secure inference process.

II. Data Transfer and Storage

To send matrices to and from the computer, we use the UART protocol. Data was sent at a BAUD rate of 100,000, with 1 ms of sleep between each send. The timeout between sending the data was necessary to allow the UART receive to "catch up," avoiding bugs with sending 0s or nonsense values. This data is then put into a compress module the A BRAM has a width of 32 bits while UART can only send 8 bits at a time.

Matrices are stored in 4 BRAMS: one to store the public key (or the ciphertext input for inference) of width 32 bits and depth 25250, one to store the secret key (or switching key for inference) of width 2 bits and depth 250, one to store the plaintext of width 2 bits and depth 50 bits, and one to store the message (or output of inference) of width 32 bits and depth 2500 bits. This means that all matrix operations are parallelized by a factor of 2. Due to the limits of the BRAM, given our parameters, this is the largest possible parallelization unless BRAM sharding is used.

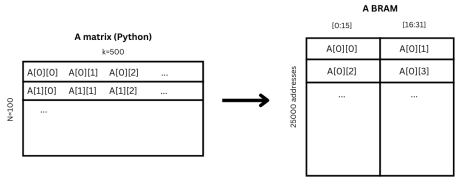

Fig. 2. The format the A matrix was stored in BRAM.

There is also a BRAM that stores neural net weights of width 3 and depth 1000, and another that stores biases of width 3 and depth 10. These are read from a .mem file using a pretrained neural net.

Once data has been sent in, the user flips a switch when they wish to begin computation. The computation to perform is selected by using 2 other switches, which will decide if the

system sends the data back out, encrypts the data, decrypts the data, or performs inference on the data.

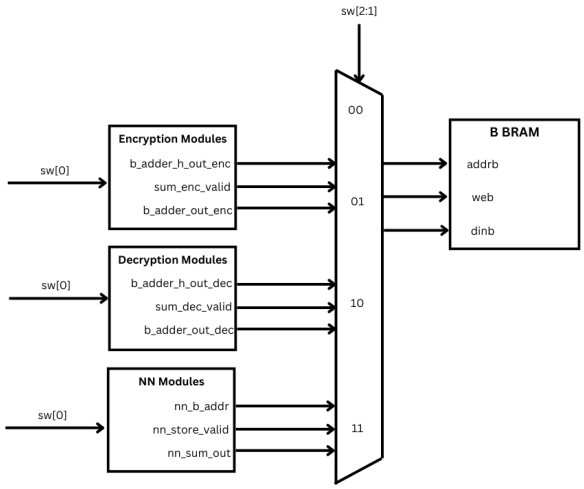

Fig. 3. Calculations begin when switch 0 is turned on. Switches 1 and 2 decide which output is stored into the BRAM.

We then use UART to send the results back to the computer, reading from each of the four BRAMs (not including the neural network weight BRAMs) and decomposing the results into a series of bytes.

III. LWE Encryption and Decryption

To suit the needs of the MNIST inference, the encryption module needs to encode 100 bits of data (N = 100), which represent 100 pixels of a scaled-down MNIST input image.

For decryption, the output of the neural net will be 10 integers, so the decryption module only needs to handle N = 10.

A. LWE Background

LWE is a type of encryption that takes advantage of the difficulty of solving noisy linear equations.

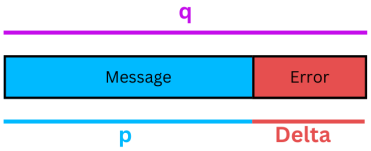

Fig. 4. Ciphertext broken down by sections based on the parameter values of p, q, and delta.

We set our parameters to k = 500, q = 2^{16}, and $p = 2^{10}$, which means we have a Δ of 2^6 bits. Setting p to 2^{10} ensures there is enough room to perform all necessary neural net computations. Setting q to 2^{16} gives us enough room for errors to build up before we bootstrap, although it limits the amount of parallelization we can achieve. Finally, setting k to 500, which is significantly greater than our N of 100, ensures the computations are secure.

B. Python Implementation of LWE

To implement LWE [3], we implemented the aforementioned encryption and decryption modules and added functions for a simple addition and multiplication of ciphertext. We did this without using NumPy and instead manually multiplying the vectors by iterating through the Python lists.

Python performance was benchmarked by having the program encode or decode 100 random plaintexts or ciphertexts. Encryption with the above parameters takes 0.022 seconds on average. Decryption takes 0.0064 seconds on average.

C. FPGA Polynomial Multiplication

To perform the public-private encoding of the plaintext, the system iterates over 2 terms of each polynomial in the secret key at a time and multiplies it with 2 terms of the corresponding polynomial of the public key. This is repeated for every polynomial, and the terms (along with the noised ciphertext output of the error addition module) are added to the b BRAM, which stores the encrypted message.

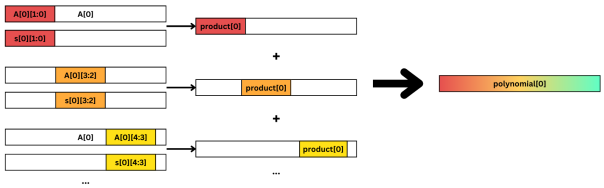

Fig. 5. Example of performing polynomial multiplication on the first row of the public and secret keys. There are a total of N=100 rows.

D. Error Addition

For adding error to the ciphertext, we randomly sample 2 bits from an LSFR module. In the same module, we also left shift the plaintext by Δ bits and add it to the error. Together, these are sent to be added to the ciphertext along with the output of the polynomial multiplication.

The output of error addition and polynomial multiplication is added with the b memory contents and put back into the b memory.

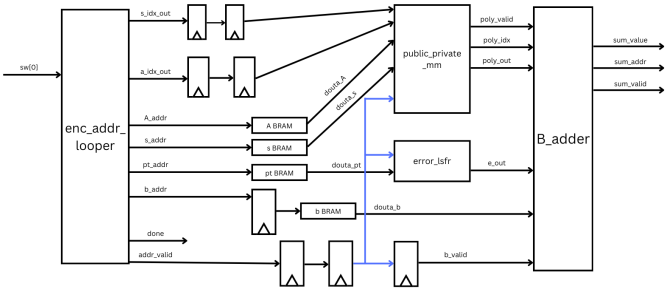

Fig. 6. Block diagram of the encryption process.

E. Verification and Benchmarking

Since the error addition is random, we ensured our encrypted output was correct by decrypting it and verifying that it was the same as the inputted plaintext. For decryption, we check that the output was identical to the original plaintext that we encrypted and sent in.

Because the FPGA uses counters to decide which addresses need to be read out of the matrix, it is possible to directly calculate the number of iterations the counter uses for a full encode cycle. There are a few cycles of pipelining afterwards, but these can be disregarded. As each address is read, the system performs computation on them. It needs to iterate over a total of 100 (N) * 250 (security parameter divided by 2) addresses. This means the system will read addresses for a total of 25k cycles for encryption, which translates to $8.3 * 10^{-4}$ of calculation.

For decryption, the system will read from 2.5k addresses since N is 10 instead of 100, which will take about $8.3 * 10^{-5}$ seconds.

IV. Secure Inference

To demonstrate a simplified example use case of the CIA, we designed it to perform inference on MNIST, a popular handwritten digit recognition task [1].

Since we are passing in a ciphered input, we used FHE [3] operations to ensure that the output remains correct.

A. FHE Background

With most encryption schemes, it is impossible to perform most functions on the ciphertext without destroying its integrity. But when using LWE ciphertexts, FHE operations can be used to perform certain arithmetic operations without knowing the content of the ciphertext.

Matrix multiplication involves repeated multiplication and addition. To multiply a ciphertext with a constant number, we can multiply every value in A and the message by that number. To add two ciphertexts, we can add every value in A and the message.

The addition of a ciphertext and an unencrypted number (which will be used to add biases to the neural net) can be done by turning the number into a trivial ciphertext (a ciphertext where A, s, and the error are all set to 0, and the message is left shifted by Δ bits), then adding it to the ciphertext.

B. Designing the Quantized NN

Because our system can only handle integer operations, we must quantize the floating point weights of the neural net into integers. In order to prevent the error from growing too rapidly, we use 3-bit zero-point quantization [4].

To create the input, we resized the 28 by 28 images of MNIST to 10 by 10 using PyTorch, then made all pixels with values above 127 equal to 1 and the rest equal to 0.

With an input layer of size 100 and an output layer of size 10, and no hidden layers, we were able to achieve 75% accuracy on MNIST validation data with 3-bit quantization. Adding more layers led to worse performance in the quantized neural net, which may be due to the increasing loss of information with each quantized layer. While 75% is significantly worse than SOTA (which often uses CNNs, which we have decided not to implement because it would require significantly more operations), it is sufficient to show that FPGAs can correctly accelerate NN inference with FHEs.

C. FHE NN in Python

To perform the FHE inference step in Python, we encrypt the 100 pixels (each represented by 1 bit). Then, we multiply the weights matrix by the ciphertext matrix, and get the new public key by multiplying the weights matrix by the public key matrix. By decrypting, we were able to confirm that the output of running inference on ciphertext is the same as running inference on plaintext.

In vanilla Python, this takes 0.22 seconds.

D. NN Implementation on FPGA

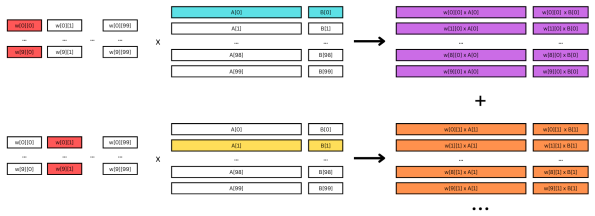

Fig. 7. The NN module iterates over the corresponding rows and columns of the weights and A matrix, then adds the result in the B BRAM.

To implement matrix multiplication, we iteratively multiply a weight column by a ciphertext row and add it to the output matrix. The output matrix includes both the public key output and the ciphertext. Iterating this way allows us to bootstrap.

Finally, to add the biases, we left shift the biases by Δ and add them to all values in the output matrix.

Each weight has a magnitude of 3 bits (using two's-complement to handle negatives). Assuming there are less than 50 pixels in the input equal to 1 while the rest are 0 (which seems reasonable, as most MNIST images are mostly background), the output can be stored in 10 bits, which is why p is 2^{10}.

Theoretically, the above procedure can be repeated for as many layers as there are in the neural net. Since our neural net only has an input and output layer, we only do it once.

To add the biases, during the first loop of the matrix multiplication, a trivial ciphertext encoding the bias was added to the output layer.

For a cycle of matrix multiplication, we iterate over 100 values of N, for each of which there are 251 memory addresses in the ciphertext, for each of which need to multiplied by 10 weights. (We add biases during the multiplication process, so that would not add extra calculation time.) This requires about $8.4 * 10^{-3}$ seconds.

While this neural net works if the error is small, if the error is too large, it will become greater than 2^{Δ}, which means it will corrupt the output data. To combat this, we need to

implement bootstrapping, a method of lowering noise in LWE ciphertext.

Although bootstrapping was not succesfully implemented in hardware, our system can still perform FHE inference on a ciphertext. Incoporating bootstrapping into hardware requires a more theoretical understanding of how the components fit together and since bootstrapping is memory-intensive, will require us to use DRAM for our parameters.

V. Bootstrapping

Bootstrapping is a method to re-encode the ciphertext with a new secret key, allowing the noise to be reduced. Some operations such as multiplication lead to very large amounts of noise which result in the solution being decrypted incorrectly. Consequently, most FHE schemes require bootstrapping to reduce the noise to allow for the ciphertext to be decrypted correctly. At this time, we have implemented components of bootstrapping in Python and in Verilog simulation, but it is not yet incorporated into the full system.

We focused on implementing two of the main components of bootstrapping for TFHE: modulus switching and blind rotation [3]. Together, these components allow us to effectively reencrypt the data to reduce noise.

A. Modulus Switching

Modulus switching 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 $\frac{\omega}{q}$.

We accomplish this by taking in the value at a certain address location, scaling each packed value, then returning the newly packed value. This gives us parallelization by modulus switching two parts of the ciphertext in parallel. Since $\omega < q$, we can use the same BRAM to store the output. Both $omega$ and q are powers of 2, so in Verilog, we can subtract these parameters to get $\log_2(\frac{q}{\omega}) = \log_2(q) - \log_2(\omega)$, then right shift a component. This will effectively multiply the element by $\frac{\omega}{q}$ as required by the algorithm while keeping the values used as integers and saving clock cycles from unnecessary multiplication and division. This entire process takes one cycle or 10 ns for two values of the ciphertext to complete. So for a ciphertext of size 200, this will take 10^{-6} seconds. If we include the time to read and write the BRAM, this is 5 cycles (two cycles to read, one cycle for the computation, and two cycles to write) so $5 * 10^{-6}$ seconds.

B. 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 works 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 rotate the message.

Since our data is compact with two values per row, our Verilog implementation of blind rotation is slightly different between even rotation amounts and odd rotation amounts. Our Verilog module takes in a rotation amount along with the previous address and previous data. If the rotation amount is even, it will output a new address rotated by the rotation amount divided by two to the left with a conditional ensuring the address wraps. This takes the module one cycle per address to rotate. If, however, the rotation amount is odd, for the first input, we store it and don't immediately return a valid address. For subsequent addresses, however, we concatenate the LSB of the previous value with the MSB of the new value and return the previous address rotated with this new concatenated data. To successfully rotate a full input, the module need one extra call at the end. In addition, the odd number of rotations assumes that the addresses will be fed into the module in order.

VI. Overall System Evaluation

A. Throughput, Latency, and Timing Requirements

As this project was designing an accelerator, one of our most important design goals was throughput. We compared our throughput with baseline Python without using any optimized libraries. As previously discussed, for the Python implementation, we estimated the runtime using the time module in Python based on multiple runs. The FPGA runtime was estimated based on the cycle count.

	Baseline Python	CIA
Encryption	0.022	$8.3 * 10^{-4}$
Decryption	0.0064	$8.3 * 10^{-5}$
Inference	0.22	$8.4 * 10^{-3}$

TABLE I
Throughput in seconds of CIA vs Un-optimized Python Implementation

One of the main bottlenecks of our system is the time it takes to pass the data in and out via UART. Currently, it takes over 2 minutes to send the data from the computer to the FPGA and less than a minute to send the data from the FPGA to the computer. We did not account for the data transfer time when comparing throughput. In addition, due to the long data transfer time, latency was not a concern, since transferring caused a bottleneck.

We were able to successfully satisfy timing with a positive WNS=1.531. We did not run into issues with a negative slack and consequently did not need to add additional pipelining.

B. Memory

We use four BRAMs for the A matrix (public key), secret key, plaintext, and b.

The FHE algorithm itself is very memory-intensive, requiring large matrices to maintain security. Consequently, we did not have much to optimize over for memory, so we used the memory necessary to store the matrices.

	Width	Depth	Size
A	32	25250	0.101 MB
PT	2	100	200 bits
SK	2	250	500 bits
B	32	2510	0.01 MB
NN (weights)	3	1000	3000 bits
NN (bias)	3	10	30 bits
Total			0.1115 MB

TABLE II
MEMORY REQUIREMENTS OF CIA

C. Checklist

We were able to successfully accomplish our commitment, as we can use UART to send and receive data and both encrypt and decrypt LWE ciphertexts. In addition, we made progress towards our goal with the neural network computation working on the FPGA and components of bootstrapping working in simulation. The last step to finish our goal would be better understanding the bootstrapping algorithm to incorporate it into our neural network computation to allow it to use noisy (and therefore secure) ciphertext while getting the correct result.

D. Implementation Insights

Surprisingly, one of the most challenging aspects of implementing the accelerator was using UART to transmit data from the computer to the FPGA. We ran into a variety of errors, including having a faulty wire which led to one of our machines getting the correct answer and the other getting the incorrect answer. Furthermore, after getting the wire issue resolved, we later noticed our uart receive stopped reading data at random times around a certain a value which would change with different BAUD rates and timeouts between the sends. We eventually used a BAUD rate of 100,000 with 1 ms of sleep between each transmission. Although we were not able to diagnose the exact cause, we suspect the UART receive module was not robust enough to handle receiving over 100,000 values continuously. Solving this took many weeks, so in hindsight, we are glad we started working on it early on.

While the inability to receive matrices initially was a bottleneck for testing the actual computation modules on hardware, we figured out we could test by preloading the BRAMs with .mem files, which allowed significant progress to be made in making the computation modules functional before the data pipeline was.

One implementation lesson learned was that using AXIS was unnecessary for this accelerator and actually hindered our implementation attempts. Because of the organized nature of matrix operations and the fact that all the data was already stored on the board before computation began, it ended up making more sense to just pass valid signals through instead of telling any module to wait.

One theoretical mistake we made was initially using general LWE (GLWE) instead of LWE. GLWE treats the entire input image as one uniform ciphertext that needed to be operated on as a whole, which prevented us from treating each pixel as a node in a neural net. Because we did not write the CPU code for the neural net before writing the encryption and decryption hardware modules, we implemented GLWE on the hardware. Fortunately, the switch from GLWE to LWE was fairly simple, although it still would've been better to catch the error in software before spending time on an incorrect algorithm in hardware.

One persistent issue we encountered is that we misunderstood the memory requirements of the system. For instance, we initially thought A was smaller than it actually was, and we chose a value of q that was too small for what we wanted to do. This problem was compounded by the fact that we could not get DRAM to work. In the end, we relied on significantly compressing the input and using a lower quantization on the neural net in order to ensure all the data we wanted to process would fit in BRAM. An unintended effect of using 3-bit quantization was that adding a hidden layer actually made the performance worse, so we removed that in our final design.

Finally, a possible timing optimization that we could've made early on was BRAM sharding. This could have led to a two or four times improvement on processing speed, but it was too complex to implement after the data pipeline and computation modules were already written.

CODE REPOSITORY

VII. INDIVIDUAL CONTRIBUTIONS

A. Code

Ruth trained and quantized the neural net in Python, as well as wrote the FPGA modules for encryption, decryption, and inference.

Shreya created the Python implementations of encryption and decryption, as well as wrote the FPGA modules for the pipeline for passing data in and out of the system and bootstrapping.

We both collaborated for debugging various issues that arose.

B. Report

We both collaborated on sections VI and II. Ruth wrote sections III and IV and created the figures for the documents. Shreya wrote sections I and V.

ACKNOWLEDGMENT

We thank Professor Joe Steinmeyer and the 6.205 teaching team for their continued assistance and advice on implementing this project.

REFERENCES

[1] Deng, L. The mnist database of handwritten digit images for machine learning research. IEEE Signal Processing Magazine, 29(6), pp.141–142, 2012.

[2] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. TFHE: Fast Fully Homomorphic Encryptionover the Torus. In Journal of Cryptology, volume 33, pp. 34–91, 2020.

[3] I. Chillotti. TFHE Deep Dive. Zama.ai Blog, May 2022.

[4] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, D. Kalenichenko. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. Dec 2017.

[5] A. Feldmann, N. Samardzic, A. Krastev, S. Devadas, R. Dreslinski, C. Peikert, and D. Sanchez. 2021. F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption. In MICRO '21: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18ś22, 2021, Virtual Event, Greece. ACM, New York, NY, USA, 15 pages.

[6] Y. Zhu, X. Wang, L. Ju and S. Guo, "FxHENN: FPGA-based acceleration framework for homomorphic encrypted CNN inference," 2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA), Montreal, QC, Canada, 2023, pp. 896-907, doi: 10.1109/HPCA56546.2023.10071133.