Zama design challenge

Purush

May 8, 2024

1 Chronology of understanding

To begin, I'll outline my approach to understanding the problem and devising a solution. Recognizing the importance of comprehending homomorphic encryption and its underlying mathematics, I delved into these concepts to gain a deeper appreciation of the complexity involved. This exploration also provided insights into the current and potential future benchmarks regarding latency, throughput, and implementation costs.

With these objectives in mind, the initial three days were dedicated to studying a relevant blog post and academic paper. Although I initially started reading the paper, it quickly became apparent that I lacked the necessary mathematical foundation to grasp its contents. Consequently, I shifted focus to the blog post, using it as a starting point to familiarize myself with the essential mathematical principles. This involved consulting various papers and watching instructional videos on platforms like YouTube. However, I encountered discrepancies in the terminology and concepts presented, which led to more confusion.

The root of this confusion lies in the disparity between the notions discussed. While the theory of polynomial fields resonated with me due to prior experience with Galois fields in Reed-Solomon encoding, considerable effort and time were devoted to understanding the distinctions between them. In essence, the crux of the matter can be summarized as follows:

- A polynomial has two characteristics: its length 'n' and its coefficient modulo 'q', represented as R_q. The length 'n' is fixed, and a sample from the polynomial can be expressed as an array of 'n' numbers, each element ranging from $[0,\ q-1]$. Therefore, the array can be represented with unsigned numbers of bit width $\lceil \log_2(q) \rceil$. Although not necessary, choosing 'N' and 'Q' as powers of 2 can lead to more efficient hardware implementations.
- From a usage perspective, there are two fundamental operations: ADD and MULT. Additionally, there is a need for a 'reduction' operation, which in this context is a power of 2 modulus.

1.1 Python code

The code doesn't always function correctly. This could be due to an issue with the algorithm or parameter settings, or possibly due to overflow/underflow errors in the Python code (Figure 1, 2). Interestingly, the values are generally close and approximately correct, which suggests that the problem may be related to numerical precision errors.

```
decrypted_ct6 /= pt6
[+] Decrypted ct6=(ct1 * ct2): [ 93 118 150  60 139  69 126  60   3 137  81 182  43 228  79  50 132 131
  35 173  71 241 105 158  96 214 182  79 228 243 228  35 171  72 175 226
  40 193 225 168 206  65 215  35 216 156  99  41 143  66 137  26  20 170
 207   7   6 220 105 152 190   5  42   1 143  84  16 220   1 133 157  56
 171 234 121 251 127 127  13  30  59 170  29 128 123  46 194  70 107 119
 158 111   6 150  38 121  29  26  22  90 147  36  40 182  21 173 164 128
 205  96  98 174 189 183 187 125 141 186 110   3 129 228  85 245 238 173
  65  33], expected: [ 93 118 150  60 139  69 126  60   3 137  81 182  43 228  79  50 132 131
  35 173  71 241 105 158  96 214 182  79 228 243 228  35 171  72 175 226
  40 193 225 168 206  65 215  35 216 156  99  41 143  66 137  26  20 169
 206   6   5 219 104 151 189   4  41   0 142  83  15 219   0 132 156  55
 170 233 120 250 126 126  12  29  58 169  28 127 122  45 193  69 106 118
 157 110   5 149  37 120  28  25  21  89 146  35  39 181  20 172 163 127
 204  96  98 173 189 183 187 125 141 186 110   3 129 228  85 245 238 173
  65  33]
```

Figure 1: Decryption error: example1

After some investigation, the rounding errors might come from the fact that *rlwe_he_scheme_updated.polymul()* implementation calculates the final value of multiplication thorough a final modulus and *np.int64()* function. The model also encounters issues when used with parameter set C, due to the fact that the ciphertext modulo $Q = 64$ exceeds the capacity of the signed implementation of *np.int64()* in the Python model. Similar issues

```
decrypted_ct6 /= pt6
[+] Decrypted ct6=(ct1 * ct2): [ 55 240 234 186 122  53  67  29 138 217 184 237 199  58 158 167   2  80
 193 155  58  90  58 146 112 152  83  81  87 169 111   6 245 231 238  67
 241 179  33  51  33 120  84 148 227  92  32 150 163 126 207 163 137 144
 249 129  97 172  39 241 229 121 157  40 215 193 120 244 212  37 119 219
 134 207 210   0 223  68  17  69  60 189  30 202 138 209  55  16 177 223
 101  82  18 179 162  64  51 200  27 170 115 207 117  87 204 233 213 142
 185 237  94 217 165  61 190  77  61  31 212 113  97 172 237 184 244 218
  52 107], expected: [ 54 239 233 185 121  52  66  28 137 216 183 236 198  57 157 166   1  79
 192 154  57  89  57 145 111 151  82  80  86 168 110   5 244 230 237  66
 240 178  32  50  32 119  83 147 226  91  31 149 162 125 206 162 136 143
 248 128  96 171  38 240 228 120 156  39 214 192 119 243 211  36 118 218
 133 206 209 255 222  67  16  69  60 189  30 202 138 209  55  16 177 223
 101  82  18 179 162  64  51 200  27 170 115 207 117  87 204 233 213 142
 185 237  94 217 165  61 190  77  61  31 212 113  97 172 237 185 245 219
  53 108]
```

Figure 2: Decryption error: example2

may arise when running the decryption algorithm with *relinearization_ver2* for Parameter set A. However, *relinearization version1* operates without any issues for the same parameter set. Finally, running parameter set B in python is out of question, as (q=2^{512}).

2 Design challenge: Part 1

2.1 Encryption

This part is to focus on encryption algorithm and propose an architecture for it. The encryption equation is given as for every message(m) one has to calculate:

$$c_t = \left([(pk0 \cdot u + e_1 + \Delta \cdot m)]_{R_q}, [(pk1 \cdot u + e_2)]_{R_q} \right) \in R_q \times R_q$$

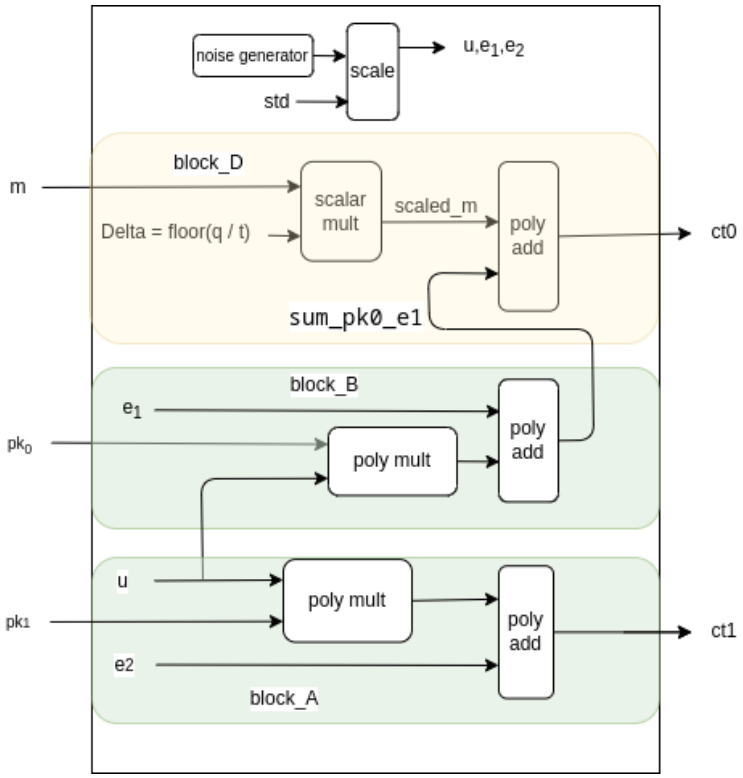

Figure 3: System Architecture

2.2 Analysis

From the given use case, we know public key tuple [pk1, pk0] is pre-computed at the host once at the beginning and is used all encryptions.

for every m,

- sample: r1,r2: normal distribution in R
  u : binary, uniform distribution sample from R
- *block_A* and *block_B* operation is independent of *Block_D*, provided u is saved for use in *block_B*
- Since q and t are constant $\Delta = \lfloor q/t \rfloor$ is also constant and can be expressed as a power of 2. Hence, scaling is just a left-shift operation on the individual fields of incoming message vector m.

In our particular case, i.e., for parameter sets A and B, (m) is left-shifted by 24 and 448, respectively. It is worth noting that even after scaling, the field elements of (*scaled_m*) are never greater than q. Therefore, no overflows occur after scaling of field values, and they can still be represented in $(\log_2(q))$ bits. In summary, reducing the latency of encryption is constrained to calculation of one *poly_add* function.

2.3 Thought process

Consider parameter set A. First we calculate, $C_1 = pk0 \cdot u + e_1$ mod q and $C_0 = pk1 \cdot u + e_2$ mod q. $pk0 \cdot u$ is a 128 coeffcient, polynomial multiplication. Since, it is relatively a small number we can do it serially as done in part 2 of the challenge.

Next let's calculate $C_1' + \Delta.m$. Since, lower 24 bits of $\Delta.m$ is zero. $(e_1 + \Delta.m)$ mod q, can be done in one 8-bit adder. Hence the design would be: with two serial multipliers, followed by a 32 bit adder, one of which is further follwed by a 8 bit adder.

Cost of 1 polynomial serial multiplier= Polymult = 2*[(2579 LUTs + 2172 FFs)]

Hence the total hardware cost: 2*[2*Polymult + 32 bit adder]

and the latency is : 128 clock cycles.

The serial architecture is as shown in figure 4. But this solution, is naive and baised towards using the serial multiplier in part 2 of the challenge. So note this is NOT the final solution, but might pass as A solution.

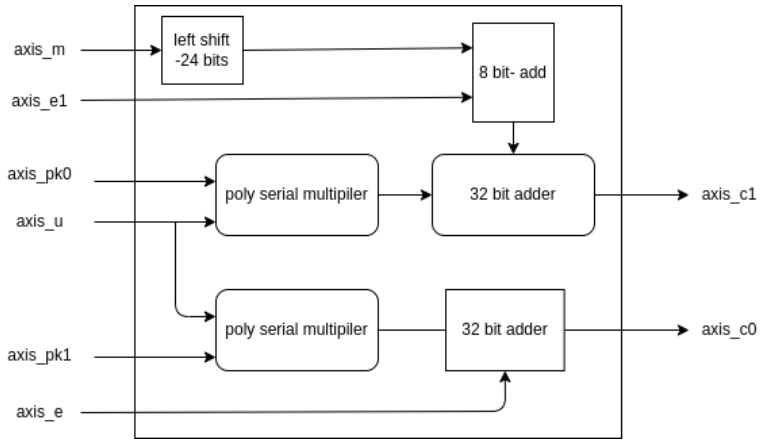

Figure 4: System Architecture: Naive Encryption parameter set A

Let's consider design for parameter set B. The polynomial size is : 16384. One obvious elimination is the above serial multiplication. In this case, we can approach the problem using number theoritic transform, which is number theory version of FFT. We know that the number of multiplications reduces from $O(N^2)$ to O(Nlog2(N)) 'scalar' multiplications and accomplishes it in log2(N) = 14 stages.

The algorithm for polynomial multiplication would be in this case be:

$$C_1' = INTT(\ NTT(Pk0)\ o\ NTT(u)), \tag{1}$$

Where o represents the is an element-wise vector multiplication in R in section 4.1. From equation 1, we can notice that $Pk0$, is pre known and constant, hence it can be pre-calculated. So the actual hardware implementation cost is calculating INTT() and NTT(u).

With no full understanding of NTT in number theory, (other than the reading reference (1)), I guess NTT(u) is binary and still behaves as uniform random sample ?. If that's the case, we can skip calculating NTT(u) and directly multiply u with NTT(Pk0).

3 Solution

But, the final solution may look like the architecture in figure 5 (A) for parameter set A. For parameter set B, use 128 instances of this to realise architecture figure 5 B. I am skipping the details, for a reason that I cannot make sound reasoning without full understanding of NTT. But, the intuitively, the solution lies in dividing the 16384 long pk0 NTT transform into 128 chunks each of size 128 multiply 128 bits of the 'u'.

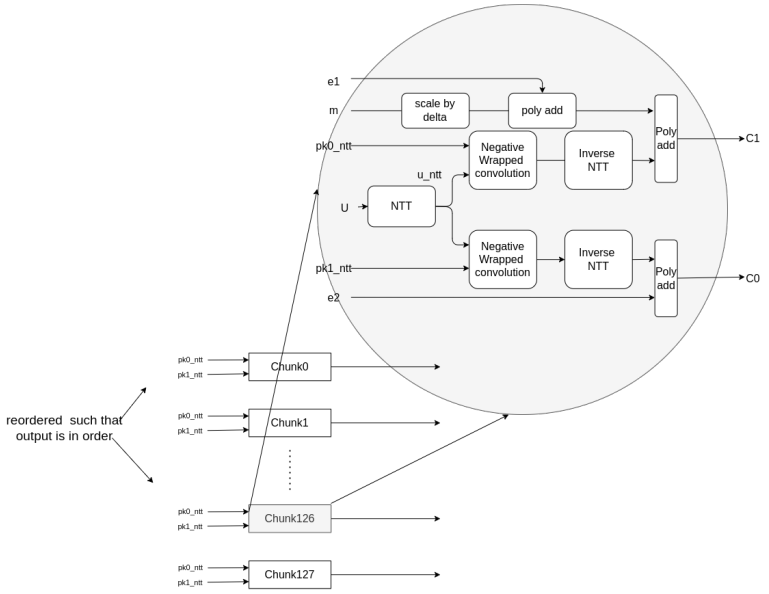

Figure 5: System Architecture: Propose Encryption parameter set A

4 Design challenge: Part 2

This part zooms into implementation of $z = pk0 \cdot u$ part of the encrption algorithm. It gives us some idea of the complexity of polynomial multiplication even though one of the parameters u is just one-bit wide.

So the hardware needs to do polynomial multiplication of $u \in R_2$ with $p \in R_q$ to produce a result $z \in R_q$. The fields in/out through an AXI stream interface.

4.1 Algorithm: Negative wrapped convolution

(NOTE: I wrote this before getting actual name for the method, in reference (1). I choose to let it stay to for the sake of completeness.) Polynomial multiplication can be seen as circular convolution coeffcients of the two polynomials. Say N=4, two polynomials A(x) and B(x) are to be multiplied, the final result $C(x)$ is given as below.

$$f(x) * g(x) = \sum_{k=0}^{N-1} f(k) \cdot g((n-k) \mod N) \tag{2}$$

$$A(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 \tag{3}$$

$$B(x) = b_0 + b_1 x + b_2 x^2 + b_3 x^3 \tag{4}$$

$$C(x) = P(x) \times Q(x) = c_0 + c_1 x + c_2 x^2 + c_3 x^3 + c_4 x^4 + c_5 x^5 + c_6 x^6 \tag{5}$$

$$C(x) = c_0 + c_1 x + c_2 x^2 + c_3 x^3 - c_4 - c_5 x - c_6 x^2 \quad subsituting \quad X^4 = -1 \tag{6}$$

Where,

$$c_0 = a_0 \cdot b_0 \tag{7}$$

$$c_1 = a_0 \cdot b_1 + a_1 \cdot b_0 \tag{8}$$

$$c_2 = a_0 \cdot b_2 + a_1 \cdot b_1 + a_2 \cdot b_0 \tag{9}$$

$$c_3 = a_0 \cdot b_3 + a_1 \cdot b_2 + a_2 \cdot b_1 + a_3 \cdot b_0 \tag{10}$$

$$c_4 = a_1 \cdot b_3 + a_2 \cdot b_2 + a_3 \cdot b_1 \tag{11}$$

$$c_5 = a_2 \cdot b_3 + a_3 \cdot b_2 \tag{12}$$

$$c_6 = a_3 \cdot b_3 \tag{13}$$

$$\tag{14}$$

$$C(x) = (c_0 - c_4) + (c_1 - c_5)x + (c_2 - c_6)x^2 + c_3x^3 \tag{15}$$
$$C(x) = s_0 + s_1x + s_2x^2 + s_3x^3 \tag{16}$$
$$s_0 = (a_0 \cdot b_0 - a_1 \cdot b_3 - a_2 \cdot b_2 - a_3 \cdot b_1)\ modQ \tag{17}$$
$$s_1 = (a_0 \cdot b_1 + a_1 \cdot b_0 - a_2 \cdot b_3 - a_3 \cdot b_2)\ modQ \tag{18}$$
$$s_2 = (a_0 \cdot b_2 + a_1 \cdot b_1 + a_2 \cdot b_0 - a_3 \cdot b_3)\ modQ \tag{19}$$
$$s_3 = (a_0 \cdot b_3 + a_1 \cdot b_2 + a_2 \cdot b_1 + a_3 \cdot b_0)\ modQ \tag{20}$$
$$\tag{21}$$

To realise the implementation in hardware, we set N partial product calulators (pprod). Each pprod, starts one clock cycle after the previous one and continues to calculate partial product for N clock cycles. Pictorially, it can be shown as in figure 6.

pProd0 is active from clk0 to clk3, pProd1 is active from clk1 to clk4, and so on. The final results which are C(0),C(1),C(2),C(3), are read out start clk4,clk5, clk6 and clk7 respectively.

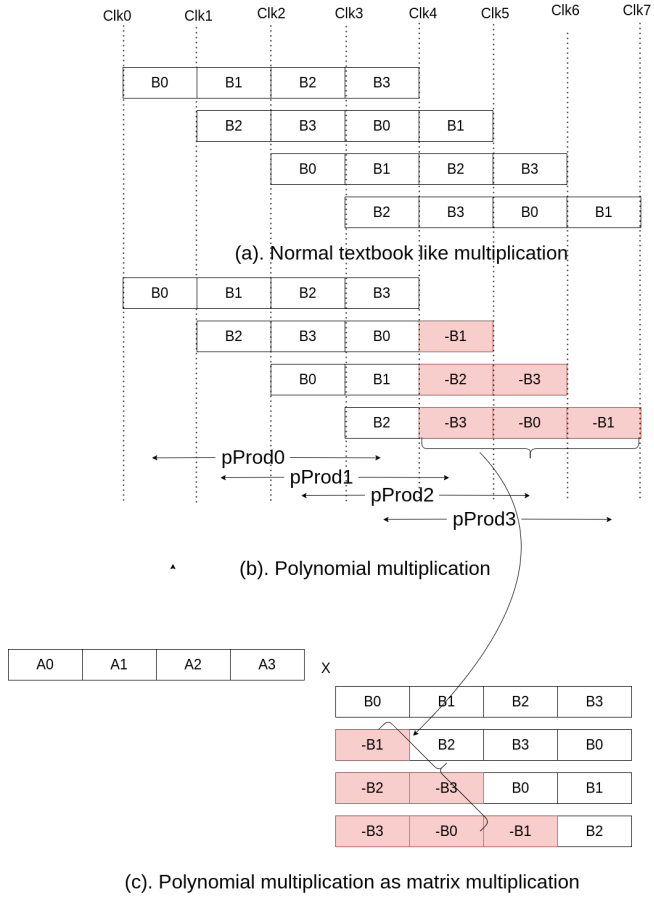

Figure 6: Multiplier dataflow

Figure 6.c show calculate C(x) through matrix multiplication. The structure of the matrix is similar to circular convolution matrix except for the sign changes that occurs due to polynomial division.

In summary, it can be seen serial polynomial multiplication happens in N clock cycles. In order to achieve non-stop streaming operation, two such multipliers are instantiated as shown in **??** each operating on alternate packets.

4.2 Simulation

The simulation environment is abysmally simplistic, in order meet the task completion deadline. Ideally, cocotb would have been used for faster checking. But that would have meant mandatory use of cocotb for even simple code sanity check.

The code in *tb_multiplier.sv* instantiates *multiplier_top* by default. This instantiates two multipliers and streams three packets back-to-back. To test only the single multipler one can just change the instance name to *multiplier* and comment extra packets being sent.

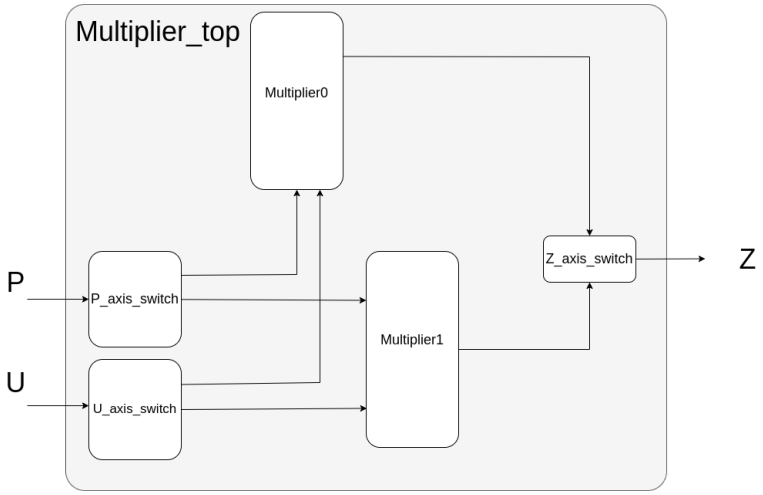

Figure 7: Multiplier top instance

4.3 Synthesis

To meet this goal few extra blocks were quickly added intentful code writing. This includes an AXI stream lfsr generating random samples and the configurable packet size N, as source for P and U ports. For Z output interface writes into a simple AXI memory. But synthesising this lead to total optimization of the built logic. The synthesis tool ultimately optimises every away. This was a wasted use of time, writing system verilog code with interfaces passed on to tasks exposed some quirks of the tool that were previously unknown to me.

A simple synthesis instantiating the single multiplier to run at 172 MHz, with 2579 LUTS and 2172 FFs. This is the case, when meeting timing for the entire design including the pins is considered.

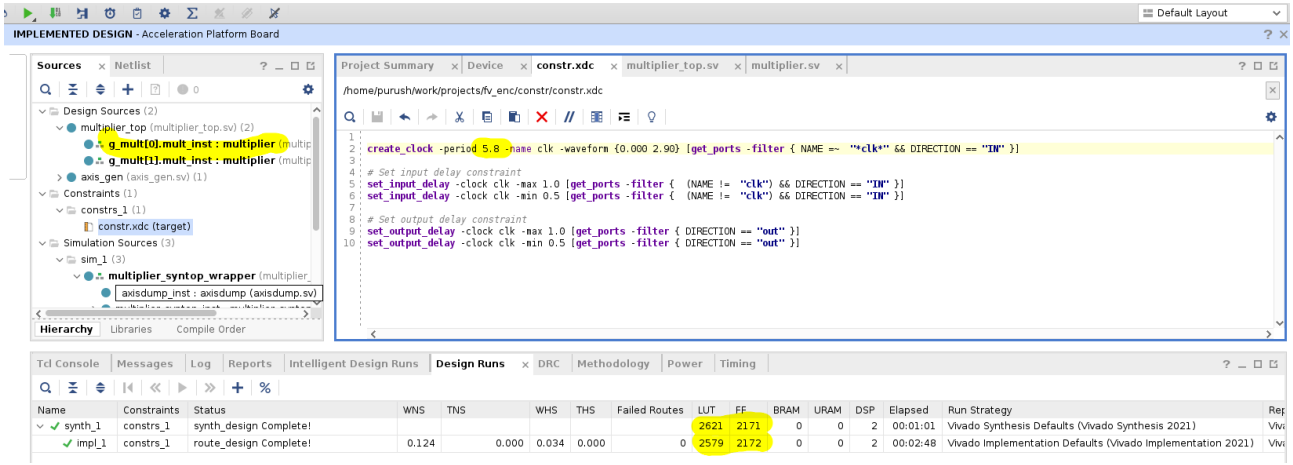

Figure 8: Synthesis results

5 What if...more time was given?

As the saying goes, with if's and buts one can build castles in the air. But being pragmatic, these are the top priorities in the to do list

- Understand the use of DSP48E2 block's multiplixing modes.
- Write a more complete test bench for the design and check random constrained cases and also bit width growth. Perhaps with cocotb.
- Spend time in reading and understanding in detail basic algorithms, Barret's algorithm for reduction. NTT for multiplication.
- Revisit code for cleanness, and do some refactoring.

6 References

Number theory https://eprint.iacr.org/2024/585.pdf

FAB (https://bu-icsg.github.io/publications/2023/fhe_accelerator_fpga_hpca2023.pdf)

rdware Primitives (https://arxiv.org/pdf/1903.03735)

Blog post (https://bit-ml.github.io/blog/post/homomorphic-encryption-toy-implementation-in-python)

orphic Encryption (https://ascslab.org/papers/he-library.pdf)

RISE (https://arxiv.org/pdf/2302.07104)

MAD (https://bu-icsg.github.io/publications/2023/Agrawal_MICRO_2023.pdf)