Zama design challenge

Purush

May 7, 2024

1 Chronology of understanding

I would first start with how I went about understanding the problem in order to try and solve it. I also felt understanding homomorphic encrption and its internal math is needed to develop deeper appreciation of complexity involved. It also sets the expectations on what is the latency, throughput and area cost of implementation currently possible and what could be the future goals.

In view of the above intentions, the first 2 days went into reading the blogpost and the paper. Although I started with paper it was evident that I didn't have the math backing to understand the language. So I started with blog post and got the basic math needed reading through various papers and youtube videos. But then got confused with difference in notions used by them.

The reason being, the polynomial fields theory part was familiar due my previous working with Galois fields for reed solomon encoding. But much of the effort and time went into understanding the difference. The gist in a nutshell is that:

- A polynomial has two charecteristics: its length 'n' and its coeffcient modulo 'q' and represented as R_q. Where the length 'n' is understood. Thus a sample from the polynominal can be represented as an array of 'n' numbers with each element being in [0,..q-1]. Hence, the array can be represented with unsigned numbers $\lceil log_2(q) \rceil$. Although not necessary, choosing 'N' and 'Q' as a power of 2 leads to efficient hardware.
-
- From an usage perpective there are two fundamental operations: ADD and MULT. Although to realise this there needs also a need for 'reduction' operation.

1.1 Python code

The code doesn't seem to always work. Either there is an algorithmic/parameters setting problem or the python code is overflow/underflowing sometimes. Interestingly, the values are close enough and approximately correct. Thus pointing 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 digging 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 fails to work with parameter_set C due to fact that the cypher text modulo Q=64 is greater that *np.int64* signed implementation of the python model. The might be a similar issue with running decryption algorithm with relinearization_ver2 even for Parameter set A. But, relinearization version1 work without issue for the same set.

Finally, running parameter set B in python is out of question, as Q=512.

```
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

1.2 Encryption

For every message(m) one has to calculate:

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

2 Design challenge part1

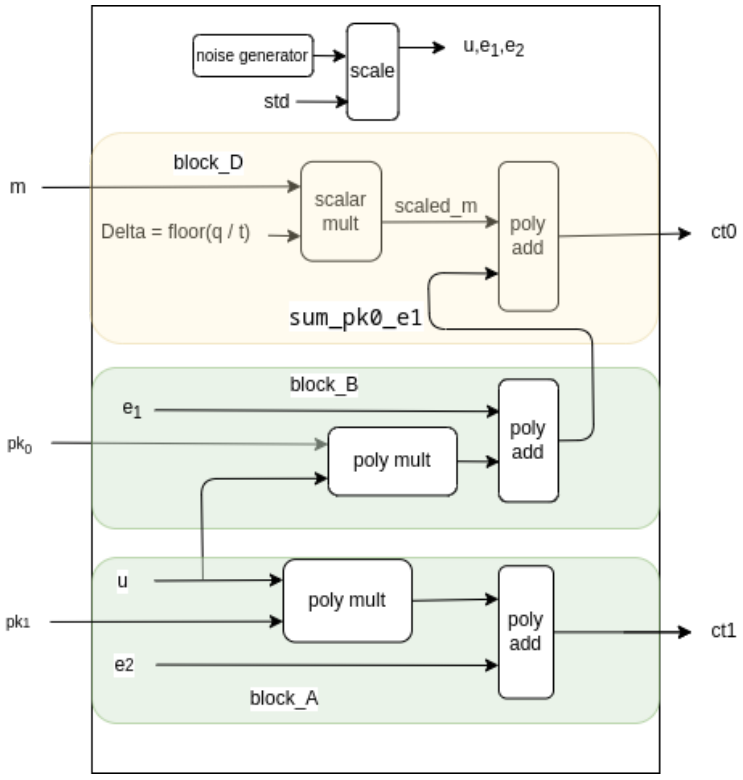

Figure 3: System Architecture

2.1 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 shifts of 24 and 448 respectively. It is good to note that even after scaling, field elements of '*scaled_m*' is never greater than 'q'. So no overflows occur after scaling of field values and can still be represented in 'log2(q)' bits. In our case, it is 32 and 512 bit for parameter sets A and B respectively.

In summary, reducing the latency of encryption is constrained to calculation of one ***poly_add*** function.

3 Design challenge part2

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 Algorithm

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.

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

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

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

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

$$C(x) = c_0 + c_1 x + c_2 x^2 + c_3 x^3 - c_4 - c_5 x - c_6 x^2 \quad subsituting \quad X^4 = -1 \tag{5}$$

Where,

$$c_0 = a_0 \cdot b_0 \tag{6}$$

$$c_1 = a_0 \cdot b_1 + a_1 \cdot b_0 \tag{7}$$

$$c_2 = a_0 \cdot b_2 + a_1 \cdot b_1 + a_2 \cdot b_0 \tag{8}$$

$$c_3 = a_0 \cdot b_3 + a_1 \cdot b_2 + a_2 \cdot b_1 + a_3 \cdot b_0 \tag{9}$$

$$c_4 = a_1 \cdot b_3 + a_2 \cdot b_2 + a_3 \cdot b_1 \tag{10}$$

$$c_5 = a_2 \cdot b_3 + a_3 \cdot b_2 \tag{11}$$

$$c_6 = a_3 \cdot b_3 \tag{12}$$

$$\tag{13}$$

$$C(x) = (c_0 - c_4) + (c_1 - c_5)x + (c_2 - c_6)x^2 + c_3 x^3 \tag{14}$$

$$C(x) = s_0 + s_1 x + s_2 x^2 + s_3 x^3 \tag{15}$$

$$s_0 = (a_0 \cdot b_0 - a_1 \cdot b_3 - a_2 \cdot b_2 - a_3 \cdot b_1) \; modQ \tag{16}$$

$$s_1 = (a_0 \cdot b_1 + a_1 \cdot b_0 - a_2 \cdot b_3 - a_3 \cdot b_2) \; modQ \tag{17}$$

$$s_2 = (a_0 \cdot b_2 + a_1 \cdot b_1 + a_2 \cdot b_0 - a_3 \cdot b_3) \; modQ \tag{18}$$

$$s_3 = (a_0 \cdot b_3 + a_1 \cdot b_2 + a_2 \cdot b_1 + a_3 \cdot b_0) \; modQ \tag{19}$$

$$\tag{20}$$

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 4.

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 4.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.

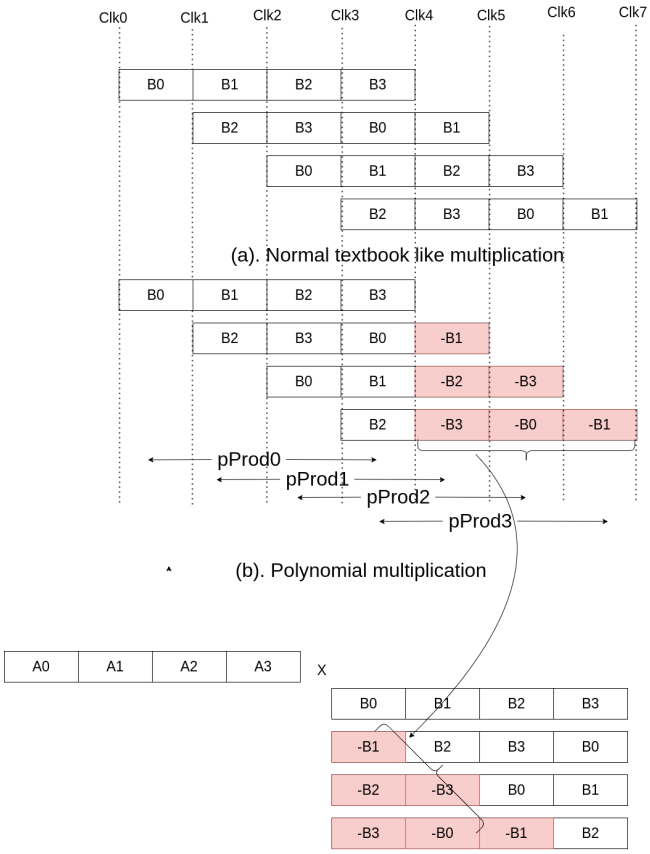

Figure 4: Multiplier dataflow

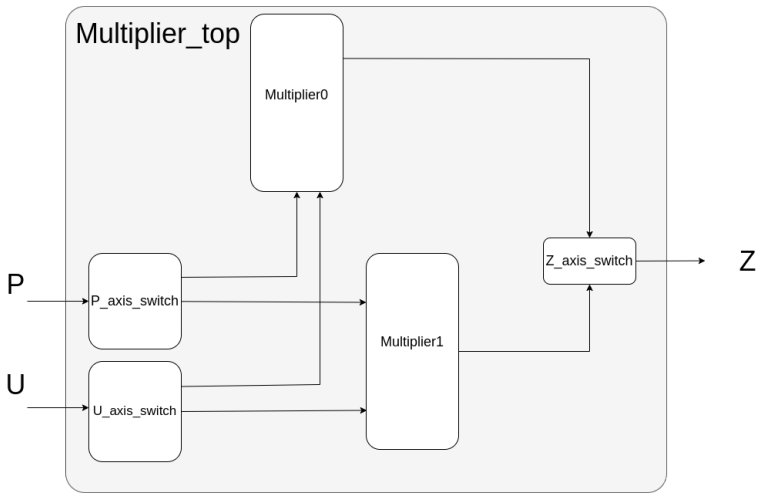

Figure 5: Multiplier top instance

4.1 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 3 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.

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

– Spend time in reading and understanding in detail basic algorithms,