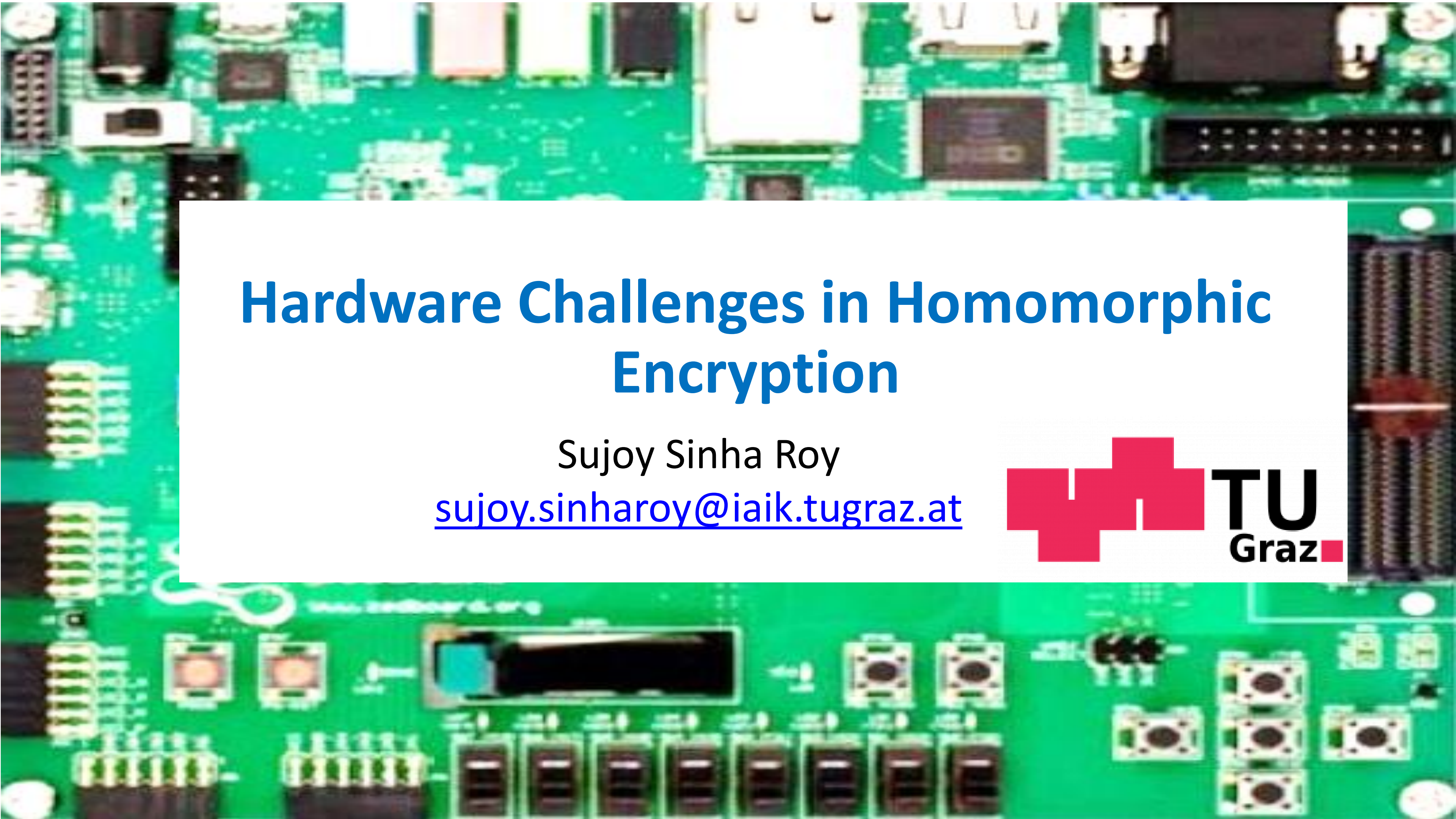
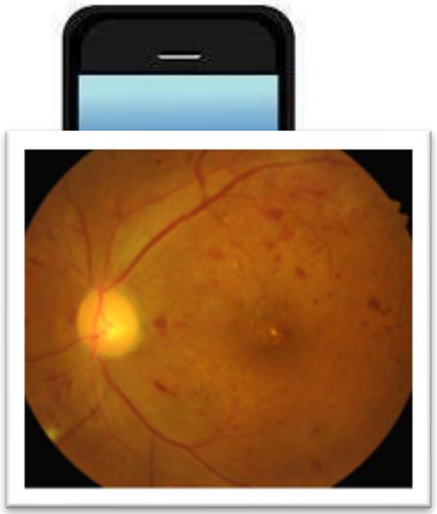


Hardware Challenges in Homomorphic Encryption

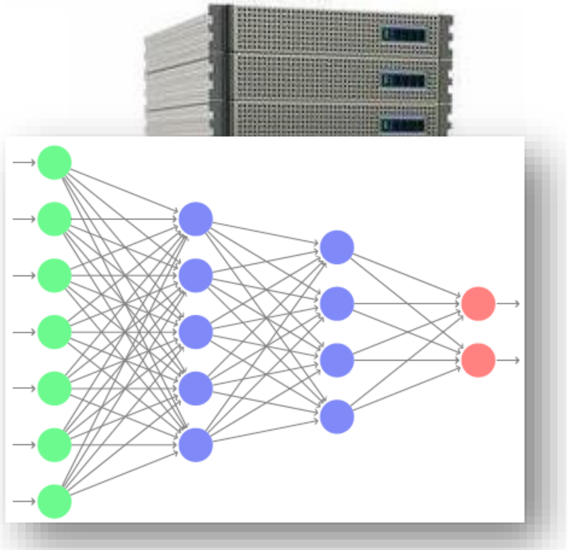
Sujoy Sinha Roy
sujoy.sinharoy@iaik.tugraz.at

Privacy-Preserving Outsourcing of Computation

data



foo()



Diabetic Retinopathy [Chao et al., 2019]

User wants to compute *foo*(*data*) in the cloud without loosing privacy.

Definition: Homomorphic Encryption Scheme

An encryption scheme Enc(· , ·) is homomorphic for an operation $\square$ on the message space iff

$$\mathrm{Enc}(m_1 \square m_2 , k_E) = \mathrm{Enc}(m_1 , k_E) \circ \mathrm{Enc}(m_2 , k_E)$$

with $\circ$ operation on the ciphertext.

- If $\square = +$ then Enc(·, ·) is additively homomorphic.
- If $\square = \times$ then Enc(·, ·) is multiplicatively homomorphic.

Example: Textbook RSA is multiplicatively homomorphic

- You have encryption of two messages m_1 and m_2 where

$$c_1 = m_1^e \bmod N$$
$$c_2 = m_2^e \bmod N$$

- By multiplying c_1 and c_2 you get

$$c_3 = c_1 \cdot c_2 = (m_1 \cdot m_2)^e \bmod N$$

- Hence, c_3 is encryption of $m_1 \cdot m_2$

Fully Homomorphic Encryption (FHE)

An encryption scheme Enc(· , ·) is homomorphic for an operation $\square$ on the message space iff

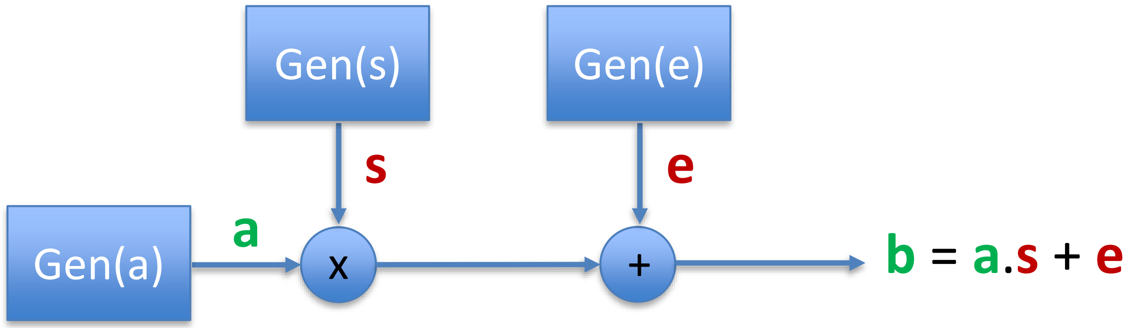
$$\mathrm{Enc}(m_1 \square m_2 , k_E) = \mathrm{Enc}(m_1 , k_E) \circ \mathrm{Enc}(m_2 , k_E)$$

An encryption scheme is called Fully Homomorphic Encryption (FHE) when it supports both + and × on ciphertexts.

- If $\square$ = + then Enc(·, ·) is additively homomorphic.
- If $\square$ = × then Enc(·, ·) is multiplicatively homomorphic.

Recap -- Ring LWE Public-Key Encryption (PKE)

- **Key Generation:**
  - **Output:** public key (pk), secret key (sk)



(Ring LWE sample)

Arithmetic operations are performed in a polynomial ring R_q

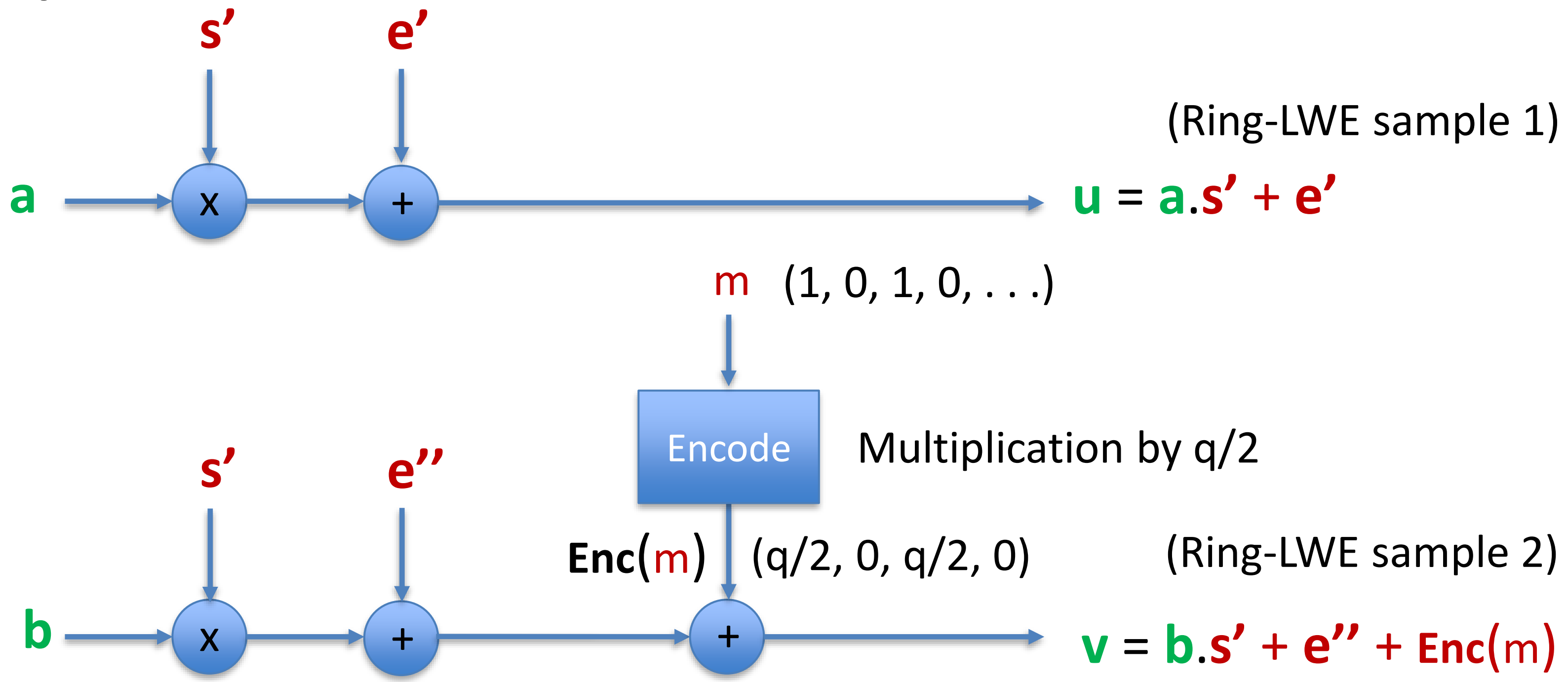
Public Key (pk): (a,b)

Secret Key (sk): (s)

V. Lyubashevsky, C. Peikert, and O. Regev. "On Ideal Lattices and Learning with Errors Over Rings". IACR ePrint 2012/230.

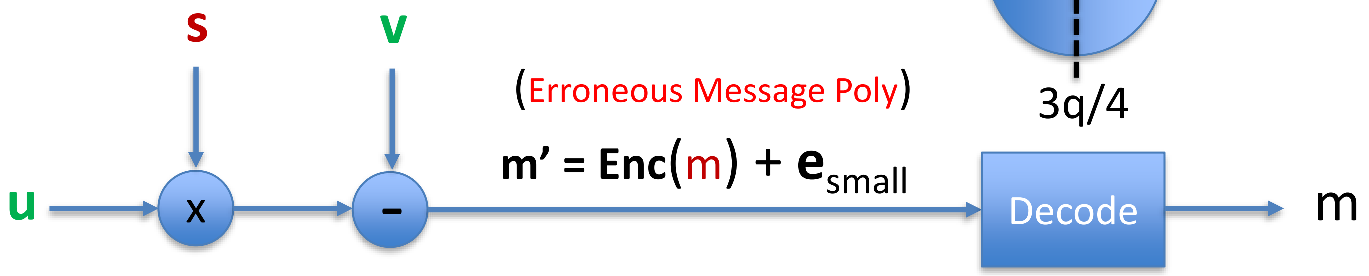
Recap -- Ring LWE Public-Key Encryption (PKE)

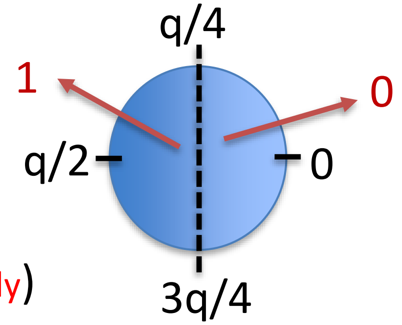
- **Encryption:**
  - **Input: pk** = (a,b), message m
  - **Output: ct** = (u,v)



Recap -- Ring LWE Public-Key Encryption (PKE)

- **Decryption:**
  - **Input:** ct = (u, v), sk = s
  - **Output:** m after decoding





$$v - u.s = m' = \mathrm{Enc}(m) + (e.s' + e'' + e'.s)$$
$$= \mathrm{Enc}(m) + e_{small}$$

Select most significant bit of each coefficient as the message bits

Ring-LWE PKE – Written with different symbols

Secret key: polynomial s

Public-key: polynomials (p_0, p_1)

Plaintext modulus: 2

Ciphertext modulus: q

Scale factor: $\Delta = q/2$

Encryption

$e_0, e_1, u \leftarrow$ error( );

$ct_0 = p_0 \cdot u + e_1 + \Delta \cdot m$

$ct_1 = p_1 \cdot u + e_2$

$\longrightarrow$

Decryption

$$\left\lceil \frac{ct_0 + ct_1 \cdot s}{\Delta} \right\rfloor \; mod \; t$$

Polynomials are in blue

Scalars are in red

Ring-LWE PKE shows Homomorphism

Encryption

$e_0, e_1, u \leftarrow$ error( );

$ct_0 = p_0 \cdot u + e_1 + \Delta \cdot m$

$ct_1 = p_1 \cdot u + e_2$

$$\longrightarrow$$

Decryption

$$\left\lfloor \frac{ct_0 + ct_1 \cdot s}{\Delta} \right\rceil \; \mathit{mod}\ t$$

Now consider two ciphertexts $Ct_A = \{ct_{A0}, ct_{A1}\}$ and $Ct_B = \{ct_{B0}, ct_{B1}\}$

$e_{A0}, e_{A1}, u_A \leftarrow$ error( );

$ct_{A0} = p_0 \cdot u_A + e_{A1} + \Delta \cdot m_A$

$ct_{A1} = p_1 \cdot u_A + e_{A2}$

$e_{B0}, e_{B1}, u_B \leftarrow$ error( );

$ct_{B0} = p_0 \cdot u_B + e_{B1} + \Delta \cdot m_B$

$ct_{B1} = p_1 \cdot u_B + e_{B2}$

Ring-LWE PKE: Additive Homomorphism

Encryption

$e_0, e_1, u \leftarrow$ error( );

$ct_0 = p_0 \cdot u + e_1 + \Delta \cdot m$

$ct_1 = p_1 \cdot u + e_2$

$\longrightarrow$

Decryption

$$\left\lceil \frac{ct_0 + ct_1 \cdot s}{\Delta} \right\rfloor \textit{ mod } t$$

Now consider two ciphertexts $Ct_A = \{ct_{A0}, ct_{A1}\}$ and $Ct_B = \{ct_{B0}, ct_{B1}\}$

$e_{A0}, e_{A1}, u_A \leftarrow$ error( );

$ct_{A0} = p_0 \cdot u_A + e_{A1} + \Delta \cdot m_A$

$ct_{A1} = p_1 \cdot u_A + e_{A2}$

$\oplus$

$e_{B0}, e_{B1}, u_B \leftarrow$ error( );

$ct_{B0} = p_0 \cdot u_B + e_{B1} + \Delta \cdot m_B$

$ct_{B1} = p_1 \cdot u_B + e_{B2}$

$ct_{C0} = p_0 \cdot (u_A + u_B) + (e_{A1} + e_{B1}) + \Delta \cdot (m_A + m_B)$

$ct_{C1} = p_1 \cdot (u_A + u_B) + (e_{A1} + e_{B1})$

Ring-LWE PKE: Multiplicative Homomorphism

Encryption

$e_0, e_1, u \leftarrow$ error( );

$ct_0 = p_0 \cdot u + e_1 + \Delta \cdot m$

$ct_1 = p_1 \cdot u + e_2$

$\longrightarrow$

Decryption

$$\left\lceil \frac{ct_0 + ct_1 \cdot s}{\Delta} \right\rfloor \textit{mod } t$$

Now consider two ciphertexts $Ct_A = \{ct_{A0}, ct_{A1}\}$ and $Ct_B = \{ct_{B0}, ct_{B1}\}$

$e_{A0}, e_{A1}, u_A \leftarrow$ error( );

$ct_{A0} = p_0 \cdot u_A + e_{A1} + \Delta \cdot m_A$

$ct_{A1} = p_1 \cdot u_A + e_{A2}$

$\otimes$

$e_{B0}, e_{B1}, u_B \leftarrow$ error( );

$ct_{B0} = p_0 \cdot u_B + e_{B1} + \Delta \cdot m_B$

$ct_{B1} = p_1 \cdot u_B + e_{B2}$

Polynomial multiplication

$ct_{A0} * ct_{B0} \rightarrow$ (**noisy crap**) $+ \Delta^2 \cdot (m_A \times m_B)$

Ring-LWE PKE: Multiplicative Homomorphism

Encryption

$e_0, e_1, u \leftarrow$ error( );

$ct_0 = p_0 \cdot u + e_1 + \Delta \cdot m$

$ct_1 = p_1 \cdot u + e_2$

$\longrightarrow$

Decryption

$\left\lceil \dfrac{ct_0 + ct_1 \cdot s}{\Delta} \right\rfloor$ *mod* t

Now consider two ciphertexts $Ct_A = \{ct_{A0}, ct_{A1}\}$ and $Ct_B = \{ct_{B0}, ct_{B1}\}$

$e_{A0}, e_{A1}, u_A \leftarrow$ error( );

$ct_{A0} = p_0 \cdot u_A + e_{A1} + \Delta \cdot m_A$

$ct_{A1} = p_1 \cdot u_A + e_{A2}$

$\otimes$

$e_{B0}, e_{B1}, u_B \leftarrow$ error( );

$ct_{B0} = p_0 \cdot u_B + e_{B1} + \Delta \cdot m_B$

$ct_{B1} = p_1 \cdot u_B + e_{B2}$

Polynomial multiplication

$ct_{A0} * ct_{B0} \rightarrow$ (**noisy crap**) $+ \Delta^2 \cdot (m_A \times m_B)$

After dividing the expression by Δ we get:

(**noisy crap**)/Δ $+ \Delta \cdot (m_A \times m_B)$

Ring-LWE PKE: Multiplicative Homomorphism

Encryption

$e_0, e_1, u \leftarrow$ error( );

$ct_0 = p_0 \cdot u + e_1 + \Delta \cdot m$

$ct_1 = p_1 \cdot u + e_2$

$\longrightarrow$

Decryption

$$\left\lceil \frac{ct_0 + ct_1 \cdot s}{\Delta} \right\rfloor \textit{ mod } t$$

Now consider two ciphertexts $Ct_A = \{ct_{A0}, ct_{A1}\}$ and $Ct_B = \{ct_{B0}, ct_{B1}\}$

$e_{A0}, e_{A1}, u_A \leftarrow$ error( );

$ct_{A0} = p_0 \cdot u_A + e_{A1} + \Delta \cdot m_A$

$ct_{A1} = p_1 \cdot u_A + e_{A2}$

$\longrightarrow \otimes \longleftarrow$

$e_{B0}, e_{B1}, u_B \leftarrow$ error( );

$ct_{B0} = p_0 \cdot u_B + e_{B1} + \Delta \cdot m_B$

$ct_{B1} = p_1 \cdot u_B + e_{B2}$

$\downarrow$

Polynomial multiplication

$ct_{A0} * ct_{B0} \rightarrow$ (**noisy crap**) $+ \Delta^2 \cdot (m_A \times m_B)$

After dividing the expression by Δ we get:

$\boxed{(\textbf{noisy crap})/\Delta + \Delta \cdot (m_A \times m_B)}$

This looks like an encryption of $(m_A \times m_B)$

Ring-LWE PKE: Multiplicative Homomorphism

Encryption

$e_0, e_1, u \leftarrow$ error( );
$ct_0 = p_0 \cdot u + e_1 + \Delta \cdot m$
$ct_1 = p_1 \cdot u + e_2$

$\longrightarrow$

Decryption

$$\left\lceil \frac{ct_0 + ct_1 \cdot s}{\Delta} \right\rfloor \mathit{mod}\ t$$

Now consider two ciphertexts $Ct_A = \{ct_{A0}, ct_{A1}\}$ and $Ct_B = \{ct_{B0}, ct_{B1}\}$

$e_{A0}, e_{A1}, u_A \leftarrow$ error( );
$ct_{A0} = p_0 \cdot u_A + e_{A1} + \Delta \cdot m_A$
$ct_{A1} = p_1 \cdot u_A + e_{A2}$

$\longrightarrow \otimes \longleftarrow$

$e_{B0}, e_{B1}, u_B \leftarrow$ error( );
$ct_{B0} = p_0 \cdot u_B + e_{B1} + \Delta \cdot m_B$
$ct_{B1} = p_1 \cdot u_B + e_{B2}$

$\downarrow$

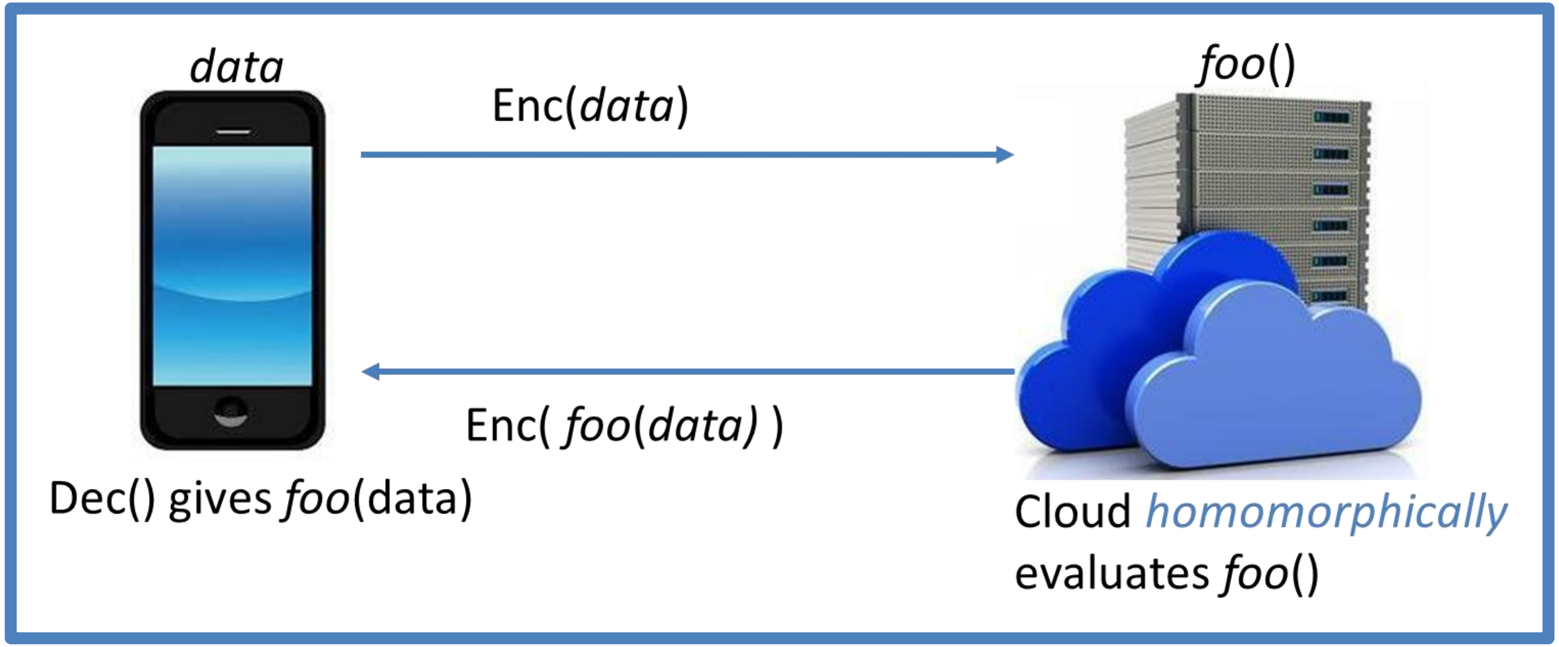
Polynomial multiplication
$ct_{A0} * ct_{B0} \rightarrow$ (**noisy crap**) $+ \Delta^2 \cdot (m_A \times m_B)$

After dividing the expression by Δ we get:
(**noisy crap**)$/\Delta + \Delta \cdot (m_A \times m_B)$

That is the basic idea only.

Actual Mult is a lot more complex!

The Biggest Problem in FHE

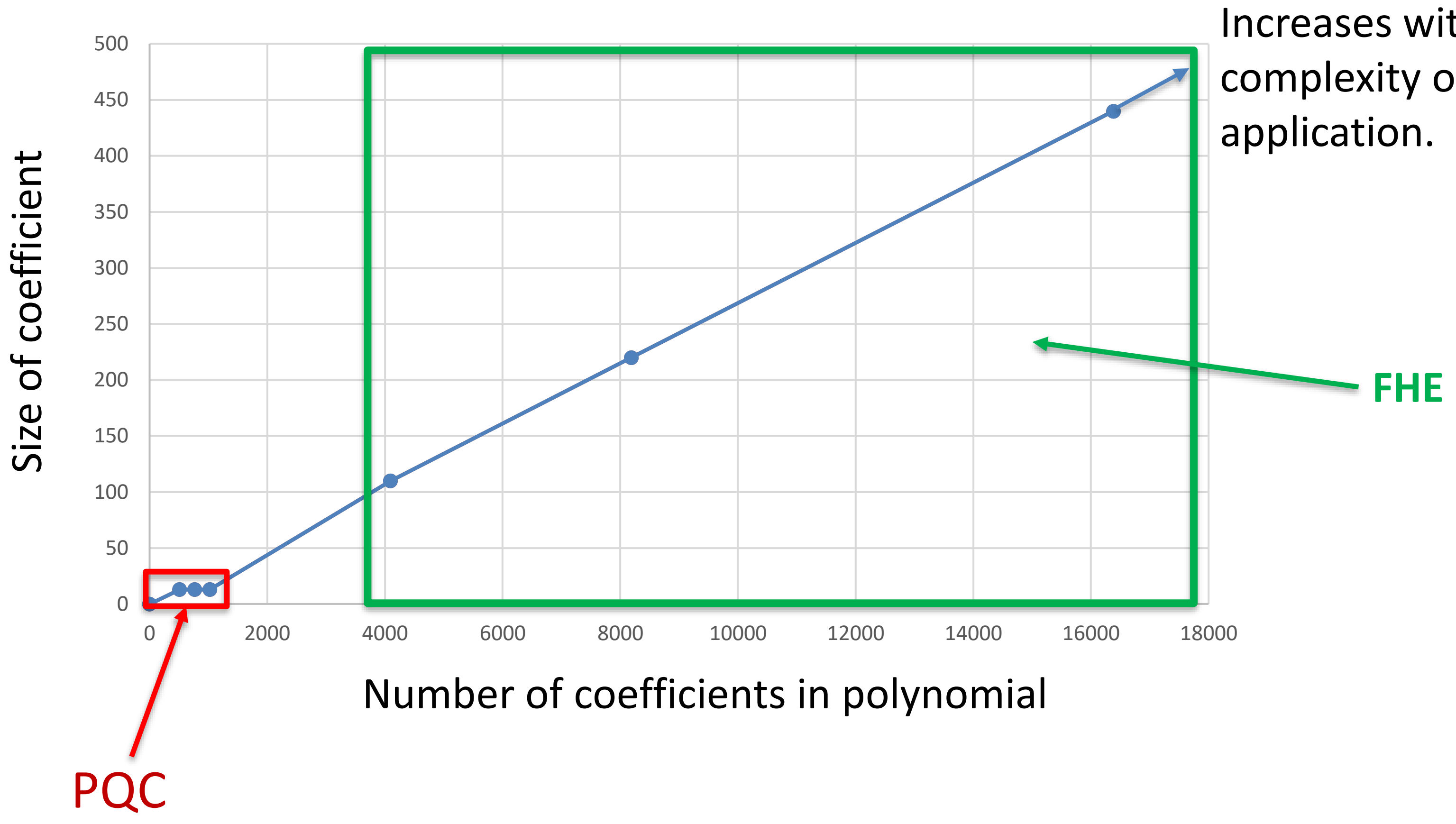


foo(data) $\longrightarrow$ *foo*(Enc(data))

Takes 1s

Takes 10^4 to 10^5 s

Parameters for PQC and FHE



Increases with complexity of application.

Like Public-key encryption,
FHE does lots of polynomial arithmetic.

How to design a hardware accelerator for FHE?

What makes implementation of FHE very challenging?

- Lots of polynomial arithmetic operations
  - Large degree polynomial arithmetic
  - Long integer arithmetic

- Big operands
  - Ciphertexts could be several MBs

- Memory management in HW accelerators
  - On-Chip memory is limited
  - Off-Chip data transfer is very slow

What makes implementation of FHE very challenging?

- Lots of polynomial arithmetic operations
  - Large degree polynomial arithmetic
  - Long integer arithmetic   This problem is solved using CRT

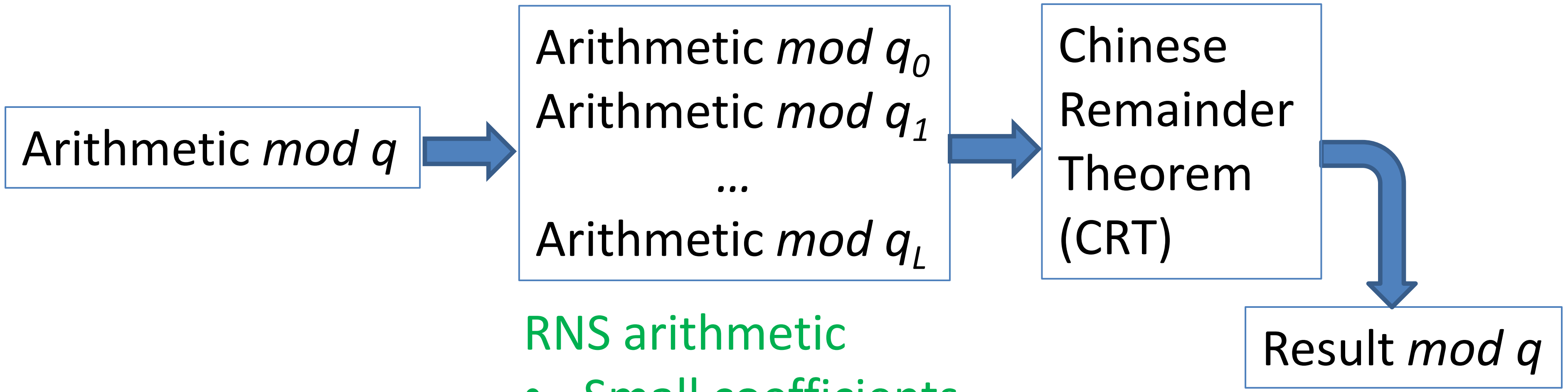
- Big operands
  - Ciphertexts could be several MBs

- Memory management in HW accelerators
  - On-Chip memory is limited
  - Off-Chip data transfer is very slow

Dealing with long-int coefficients using RNS

We can take a modulus $q = \prod_{0}^{L} q_i$ where q_i are coprime.

Then we can work with Residue Number System (RNS).



RNS arithmetic

- Small coefficients
- Parallel computation

What makes implementation of FHE very challenging?

- Lots of polynomial arithmetic operations
  - Large degree polynomial arithmetic
  - Long integer arithmetic

- Big operands
  - Ciphertexts could be several MBs

- Memory management in HW accelerators
  - On-Chip memory is limited
  - Off-Chip data transfer is very slow

How to multiply two very large polynomials?

- Schoolbook multiplication: $O(n^2)$
- Karatsuba multiplication: $O(n^{1.585})$
- Toom-Cook (generalization of Karatsuba)
- Fast Fourier Transform (FFT) multiplication: $O(n \log n)$

Which one is the best choice?

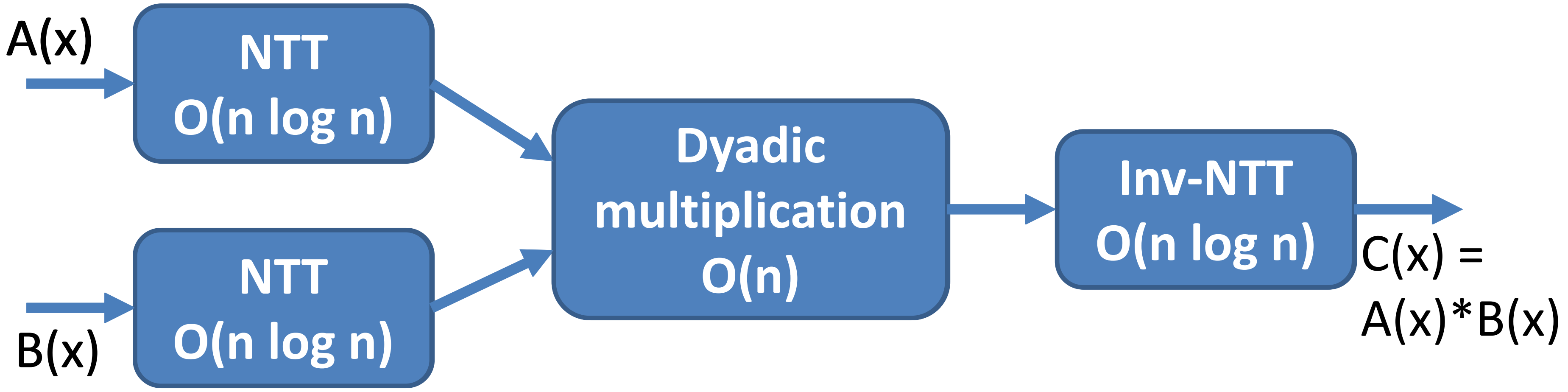
How to multiply two very large polynomials?

- Schoolbook multiplication: $O(n^2)$
- Karatsuba multiplication: $O(n^{1.585})$
- Toom-Cook (generalization of Karatsuba)
- **Fast Fourier Transform (FFT) multiplication: $O(n \log n)$**

Which one is the best choice?

Asymptotic complexity plays its role.

NTT-based Polynomial Multiplication



NTT or Number Theoretic Transform

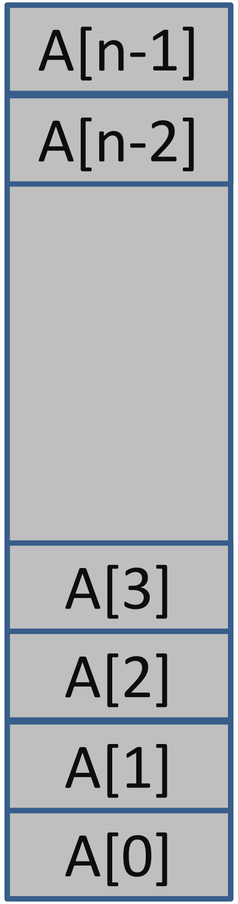
Let’s consider an application example.

Polynomial size $n = 2^{15}$

$\mathrm{Log}(q) = 60$

NTT and of a polynomial A[ ]

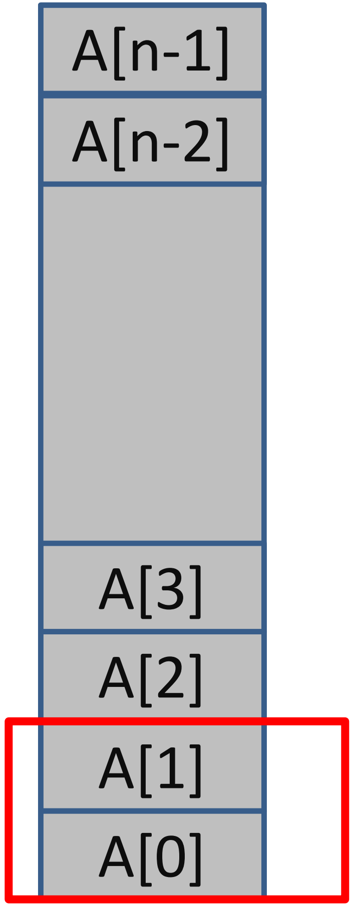
Simplified NTT loops



```
for(m=2; m<=n; m=2m){
   for(j=0; j<=m/2-1; j++){
      for(k=0; j<n; k=k+m){
         index = f(m, j, k);
         Butterfly(A[index],A[index+m/2]);
      }
   }
}
```

NTT and Memory access

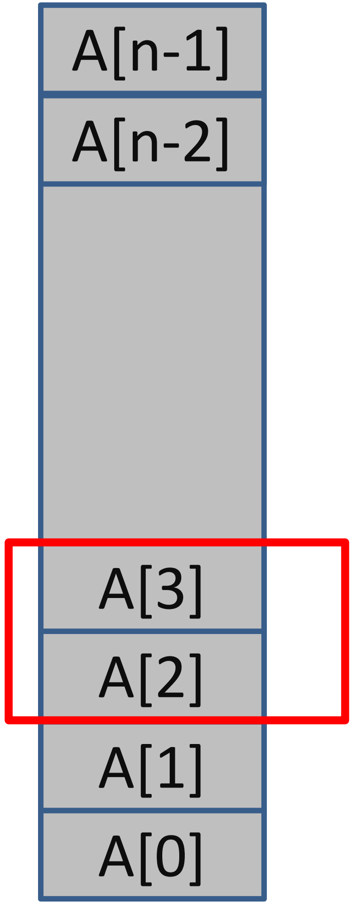
Simplified NTT loops



```
for(m=2; m<=n; m=2m){
   for(j=0; j<=m/2-1; j++){
      for(k=0; j<n; k=k+m){
         index = f(m, j, k);
         Butterfly(A[index],A[index+m/2]);
      }
   }
}
```

NTT starts with m=2
Butterfly(A[0], A[1])

NTT and Memory access



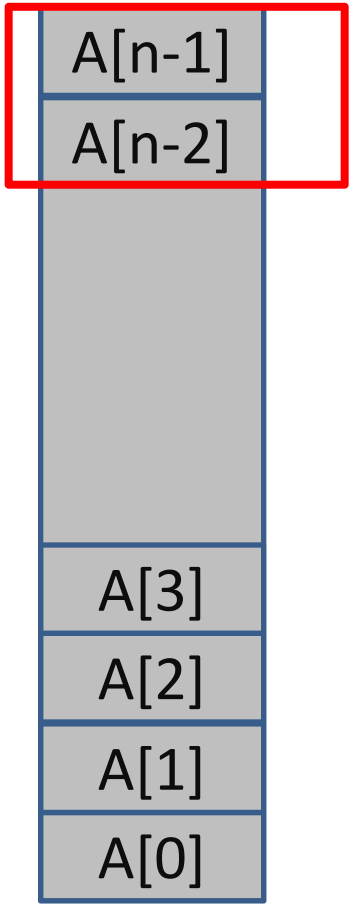
Simplified NTT loops

```
for(m=2; m<=n; m=2m){
   for(j=0; j<=m/2-1; j++){
      for(k=0; j<n; k=k+m){
         index = f(m, j, k);
         Butterfly(A[index],A[index+m/2]);
      }
   }
}
```

… with m=2

Butterfly(A[2], A[3])

NTT and Memory access

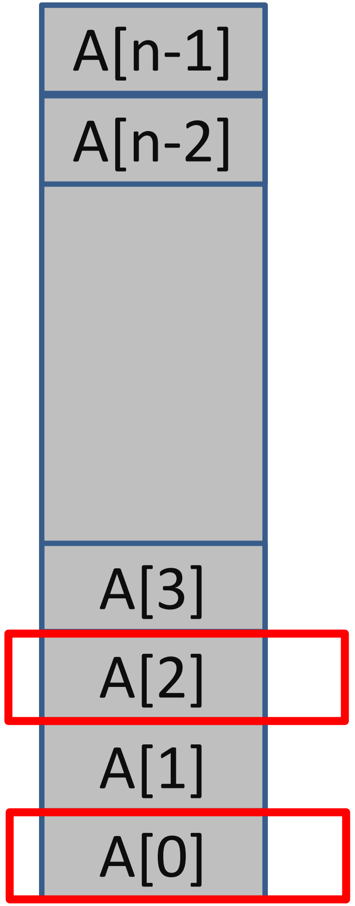


Simplified NTT loops

```
for(m=2; m<=n; m=2m){
   for(j=0; j<=m/2-1; j++){
      for(k=0; j<n; k=k+m){
         index = f(m, j, k);
         Butterfly(A[index],A[index+m/2]);
      }
   }
}
```

… with m=2, finally
Butterfly(A[n-2], A[n-1])

NTT and Memory access

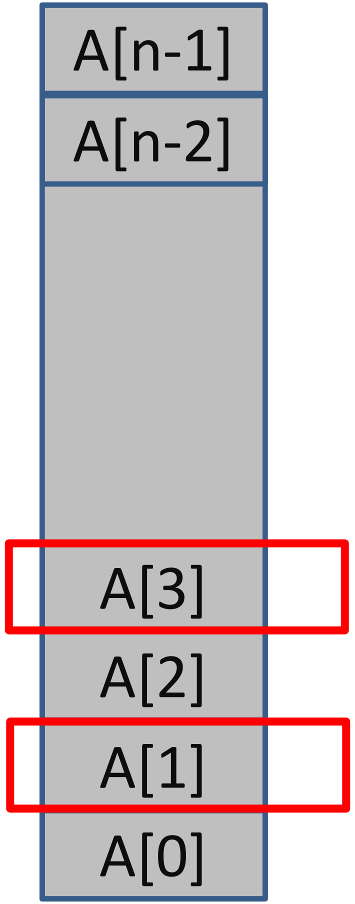


Simplified NTT loops

```
for(m=2; m<=n; m=2m){
   for(j=0; j<=m/2-1; j++){
      for(k=0; j<n; k=k+m){
         index = f(m, j, k);
         Butterfly(A[index],A[index+m/2]);
      }
   }
}
```

Next, m increments to m=4.
Butterfly(A[0], A[2]), Butterfly(A[4], A[6]) …

NTT and Memory access



Simplified NTT loops

```
for(m=2; m<=n; m=2m){
   for(j=0; j<=m/2-1; j++){
      for(k=0; j<n; k=k+m){
         index = f(m, j, k);
         Butterfly(A[index],A[index+m/2]);
      }
   }
}
```

Next, m increments to m=4.
Butterfly(A[1], A[3]), Butterfly(A[5], A[7]) …

Can we speedup polynomial multiplication using several NTT cores in parallel?

Answer: Yes

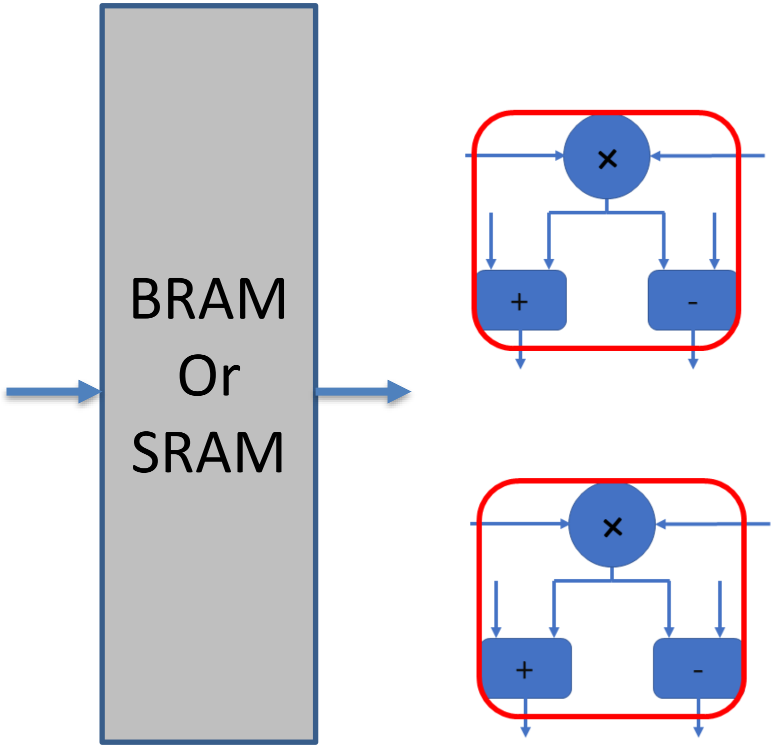
Can we speedup polynomial multiplication using several NTT cores in parallel?

Answer: Yes

Is parallel NTT easy to implement?

Answer: Complexity of implementation increases with number of cores

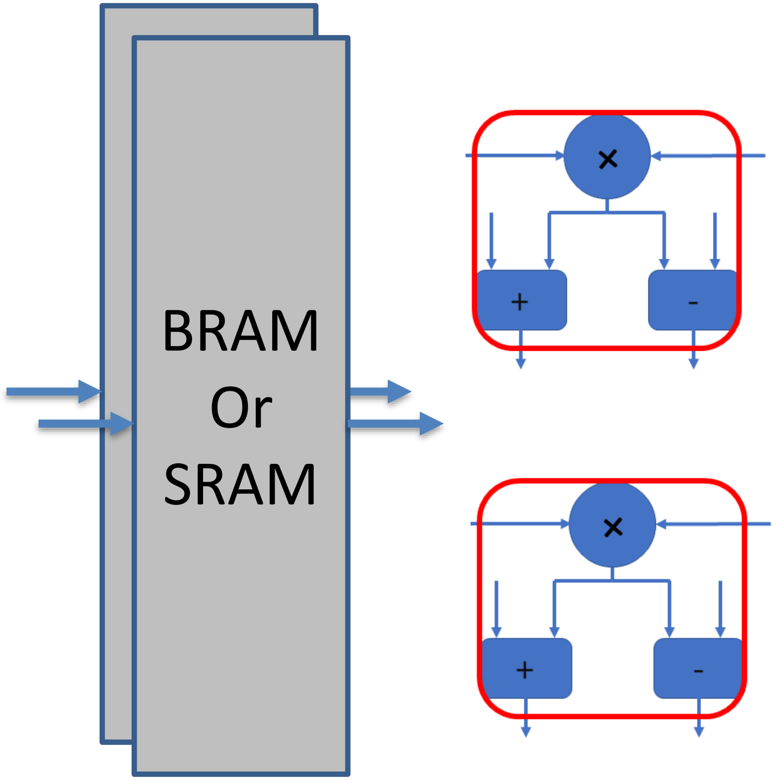
Parallel NTT
Challenge: Port limitation in BRAM or SRAM



Problem:

- One BRAM has only two ports.
- Each NTT core needs two ports

Parallel NTT
Challenge: Port limitation in BRAM or SRAM



Problem:

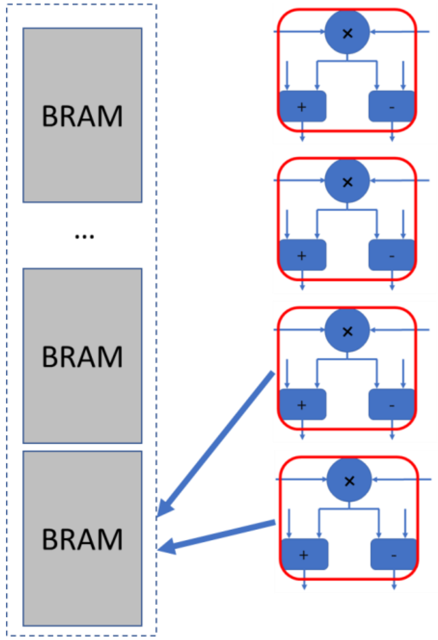
- One BRAM has only two ports.
- Each NTT core needs two ports

To get parallel NTT, designers instantiate *parallel* BRAMs in parallel.

Parallel NTT

Memory access conflict

- Two or more cores try to read/write the same BRAM element. But BRAM has a limited number of ports to satisfy one core.

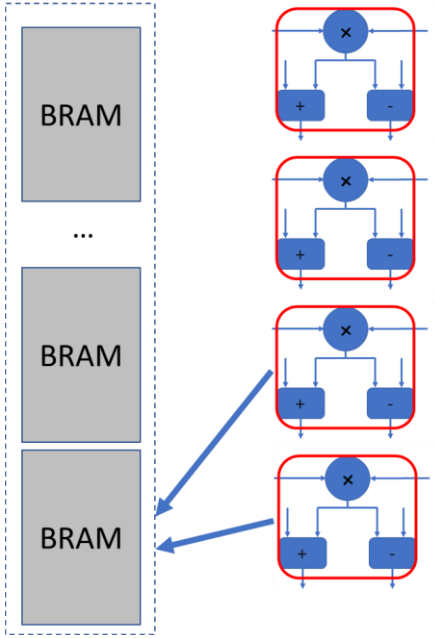


Two cores are trying to access the same BRAM.

Parallel NTT

Memory access conflict

- Two or more cores try to read/write the same BRAM element. But BRAM has a limited number of ports to satisfy one core.



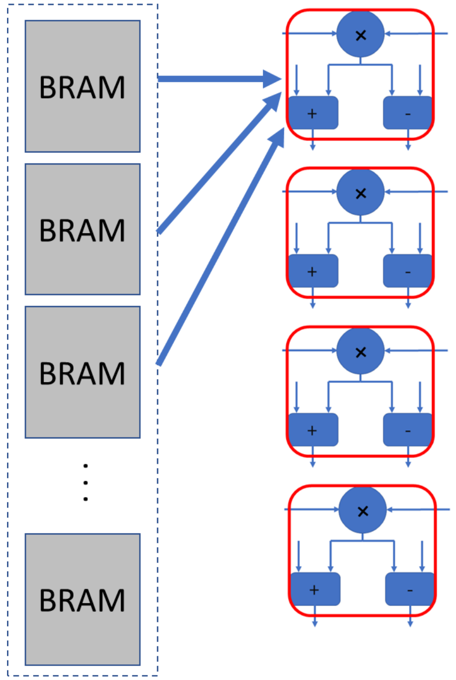
Solution: Cores generate addresses such that they are mutually exclusive.

Two cores are trying to access the same BRAM.

Parallel NTT

Long data routing

- Core requires data from distant BRAM memory
  - Long routing of data wires → slow clock frequency

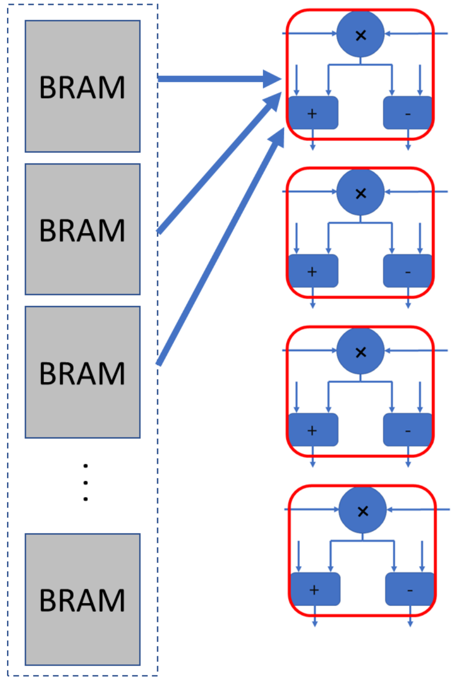


Core is reading data from far memory.

Parallel NTT

Long data routing

- Core requires data from distant BRAM memory
  - Long routing of data wires → slow clock frequency

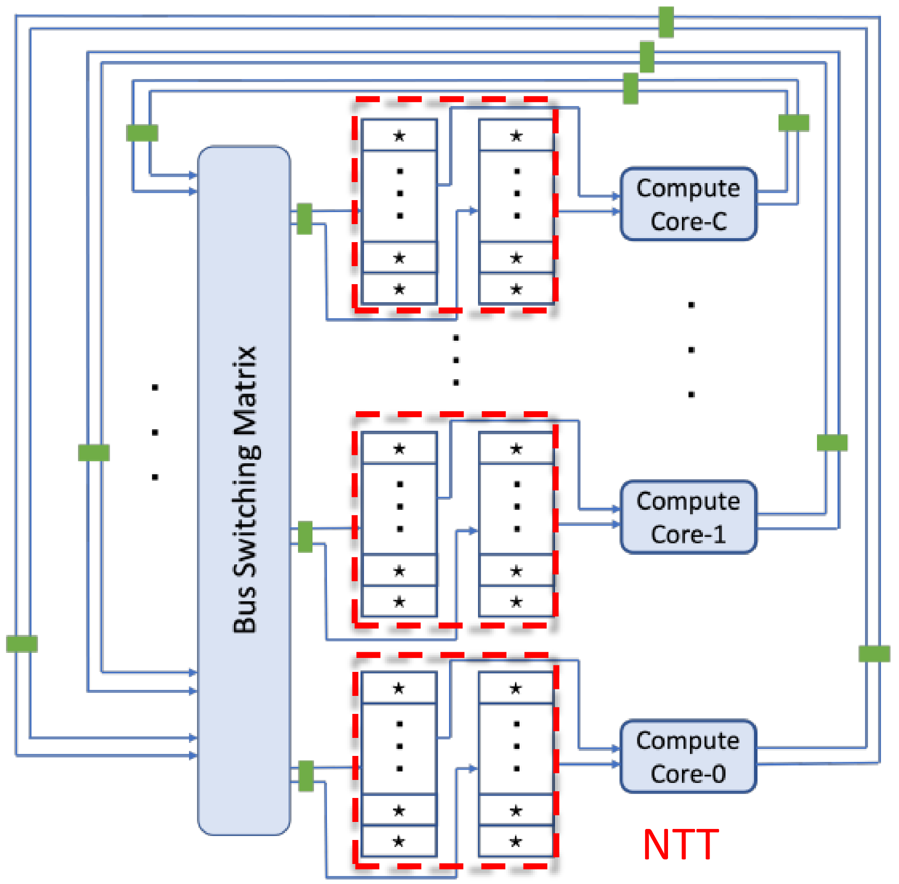


Core is reading data from far memory.

Solution: There is no easy solution to this problem.
Research papers propose localizing read or write (not both)

This paper localizes the read operation.

BRAM is exclusively read by only one core.



Wires to write coefficients to BRAMs. They are pipelined using layers of reg.

BRAMs

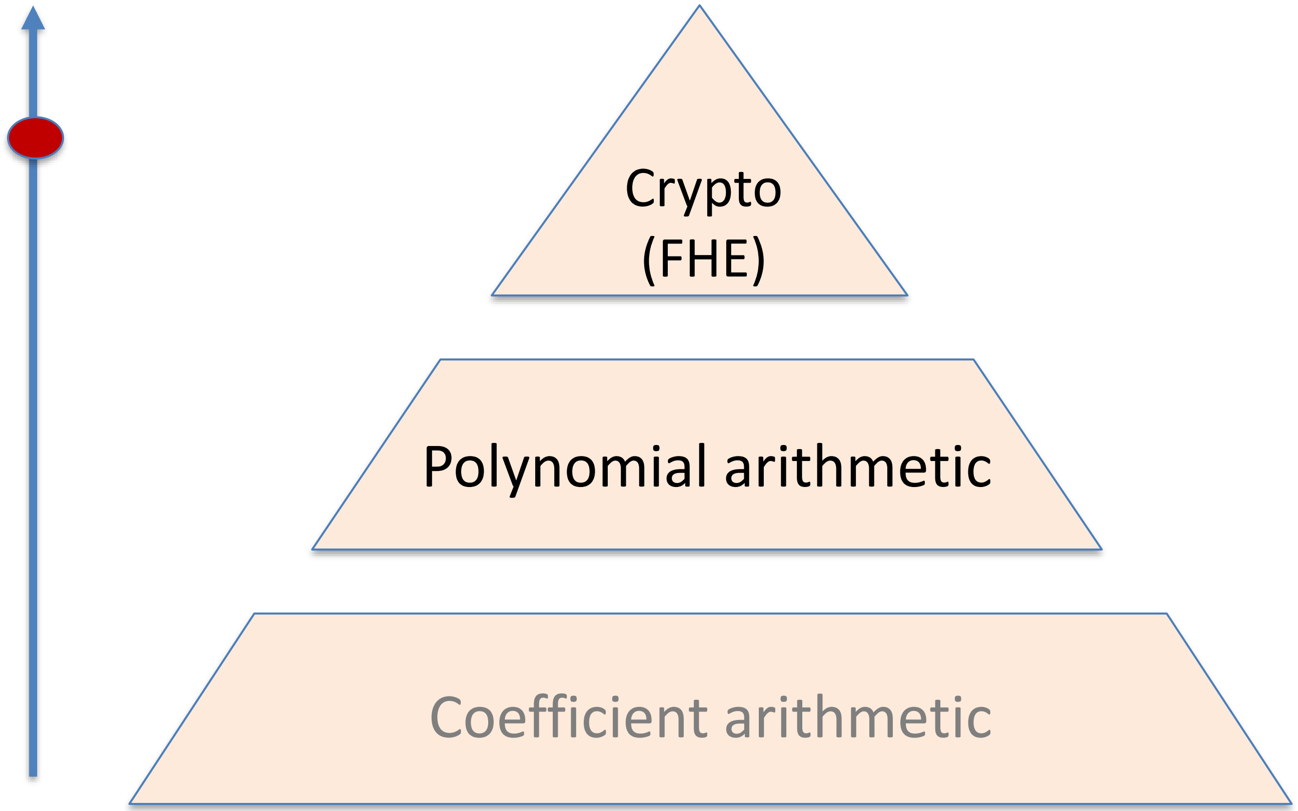
NTT Cores

🟩 Pipeline register

☆ Coefficient of a polynomial

Mert et al. “Medha: Microcoded Hardware Accelerator for computing on Encrypted Data”. TCHES 2023

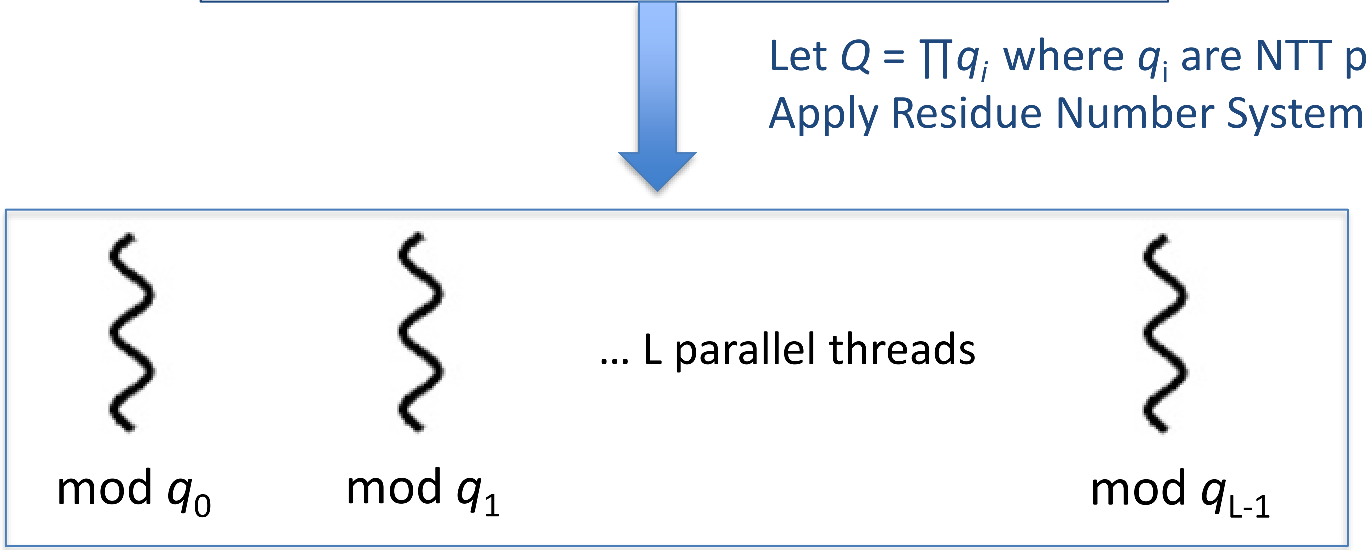
Next, FHE accelerator



High level computation flow

Ciphertexts are polynomials in $R_Q = Z_Q/\langle X^n + 1\rangle$
E.g., $\log(Q) = 500$, $n = 2^{15}$

Let $Q = \prod q_i$ where q_i are NTT primes.
Apply Residue Number System (RNS)



Each thread perform arithmetic in residue polynomial ring R_{qi}

High level computation flow

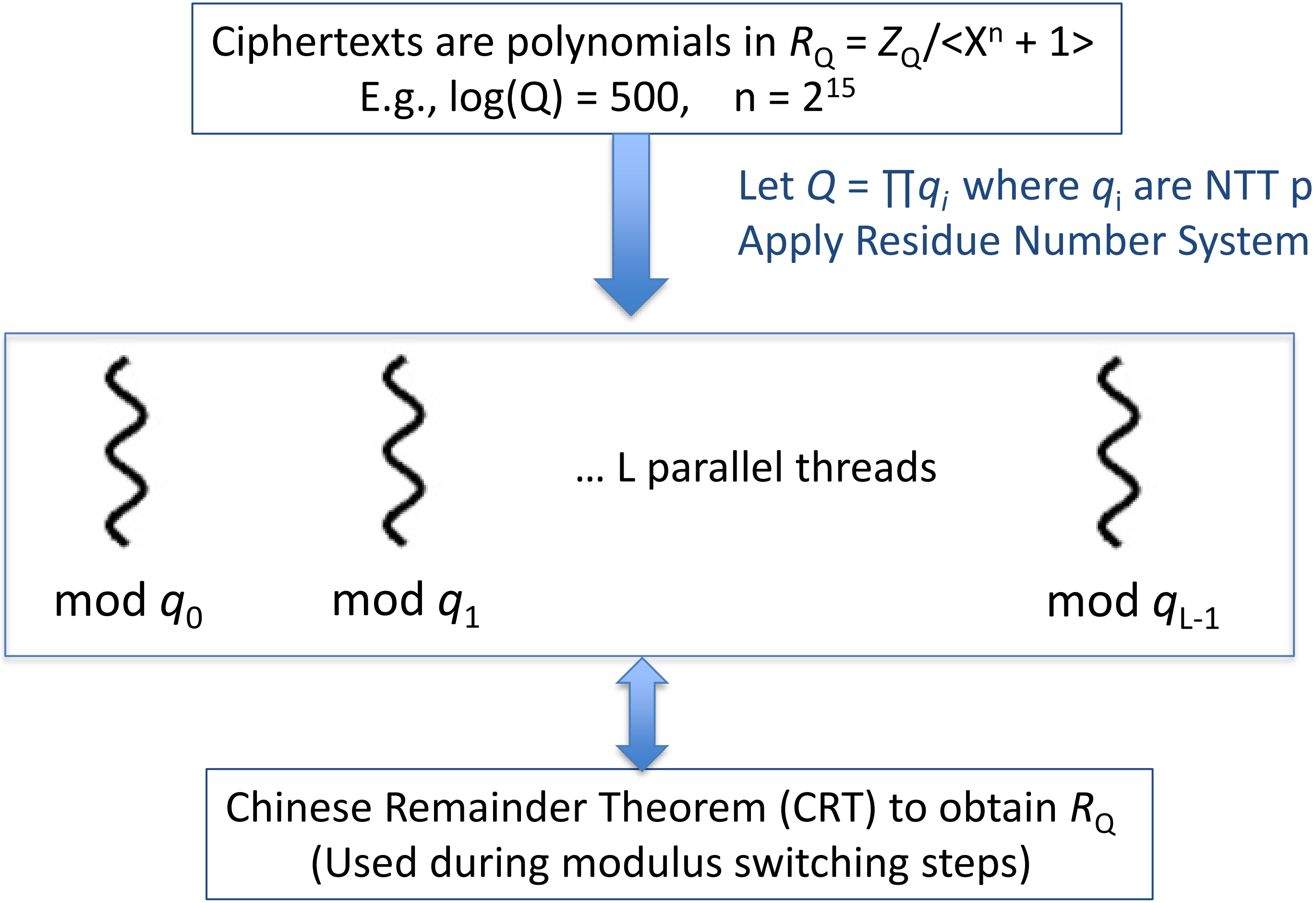
Ciphertexts are polynomials in $R_Q = Z_Q/\langle X^n + 1\rangle$
E.g., $\log(Q) = 500$, $n = 2^{15}$

Let $Q = \prod q_i$ where q_i are NTT primes.
Apply Residue Number System (RNS)

mod q_0 mod q_1 … L parallel threads mod q_{L-1}

NTTs, INTTs,
Coeff-wise add,
sub, mult, etc.

High level computation flow

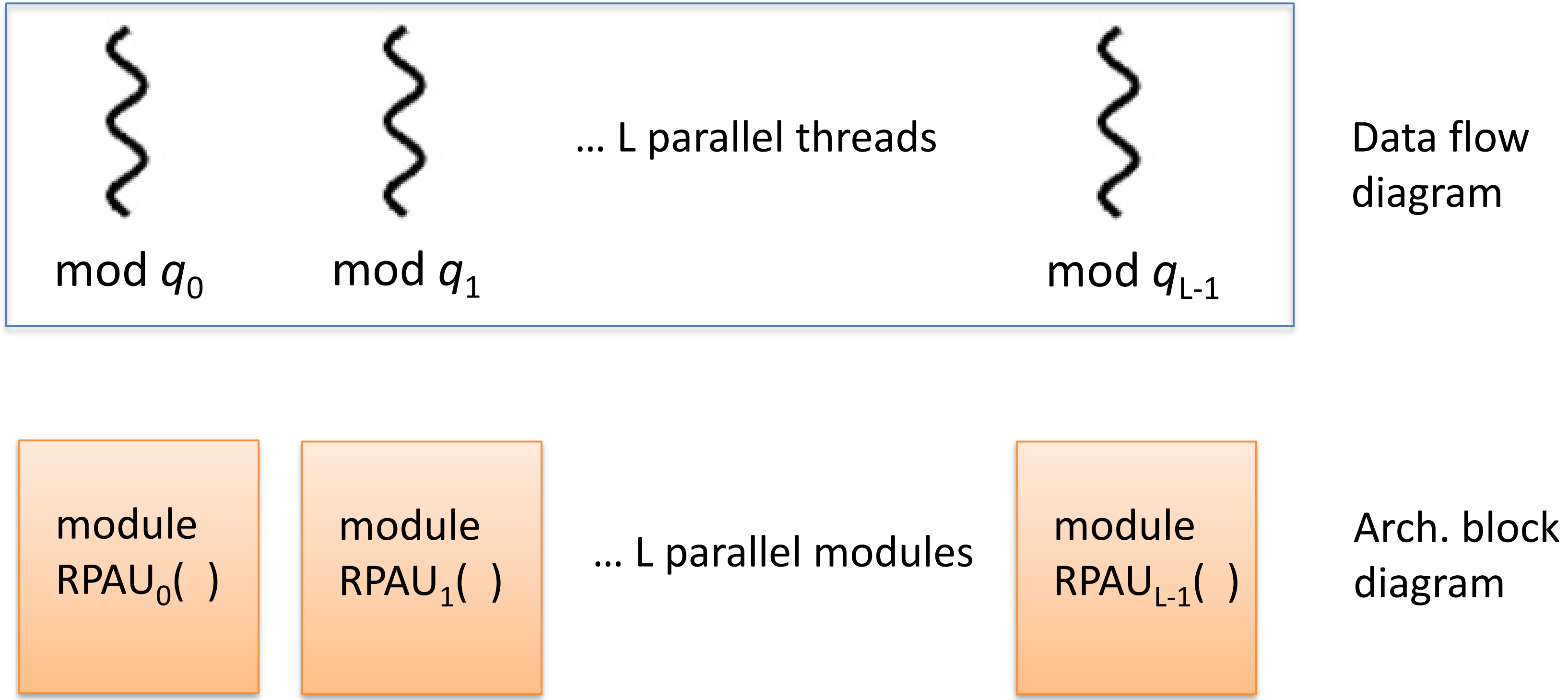


Ciphertexts are polynomials in $R_Q = Z_Q/\langle X^n + 1\rangle$
E.g., $\log(Q) = 500$, $n = 2^{15}$

Let $Q = \prod q_i$ where q_i are NTT primes.
Apply Residue Number System (RNS)

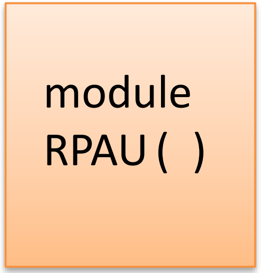
Chinese Remainder Theorem (CRT) to obtain R_Q
(Used during modulus switching steps)

High level accelerator architecture



*RPAU stands for Residue Polynomial Arithmetic Unit

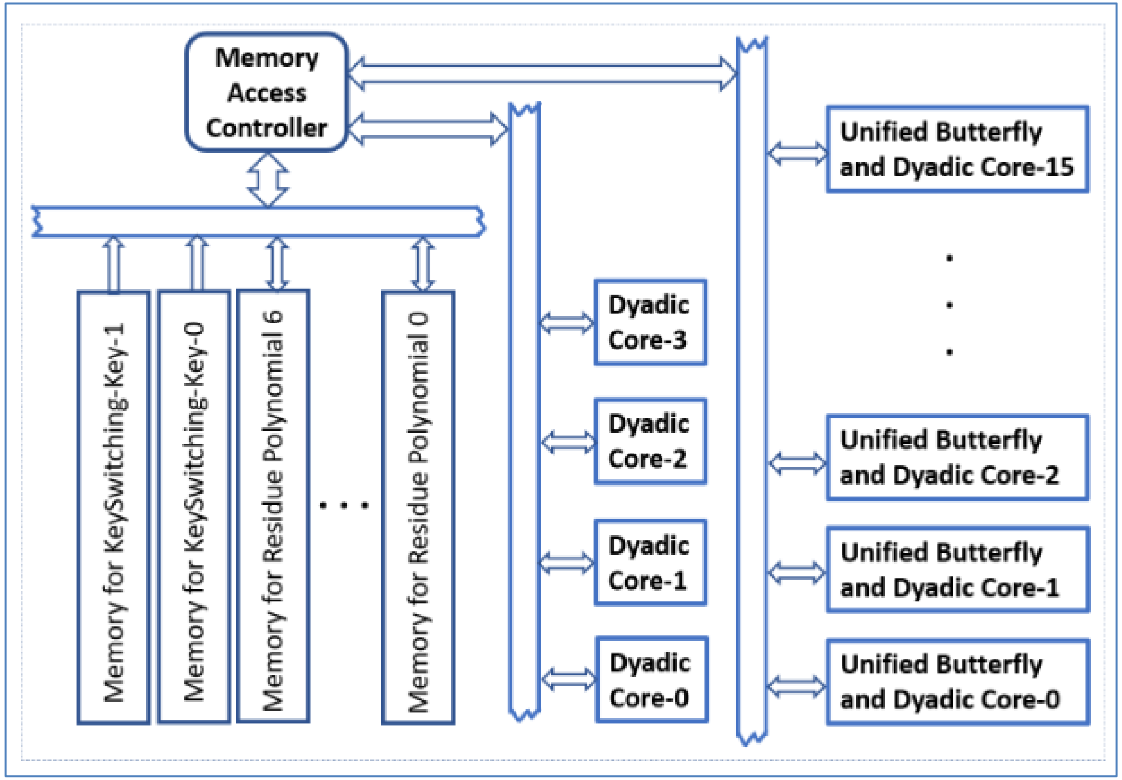
RPAU ( )



Each RPAU( ) module must support arithmetic modulo q_i

- NTT
- INTT
- Modular reduction by q_i
- Coefficient-wise modular addition
- Coefficient-wise modular multiplication

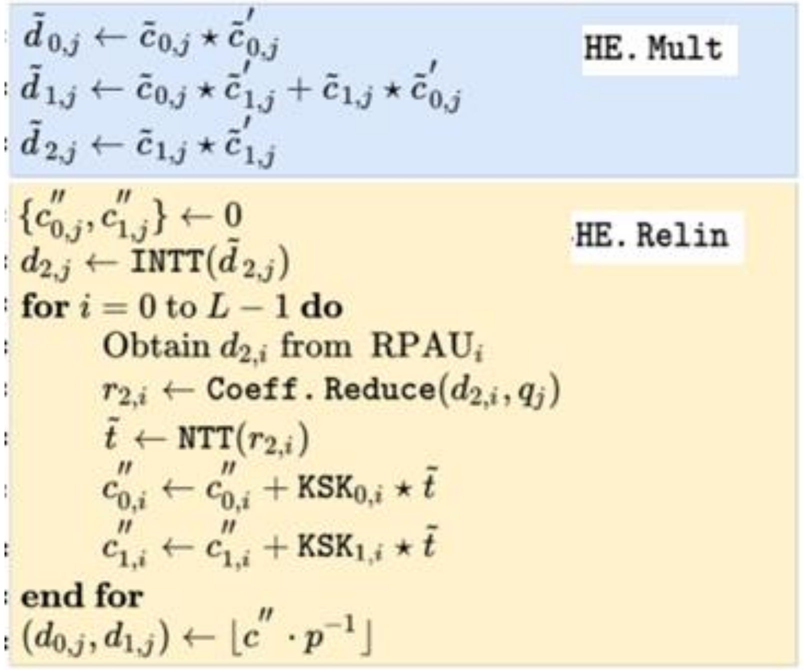
RPAU ( )


Memory Access Controller
Memory for KeySwitching-Key-1
Memory for KeySwitching-Key-0
Memory for Residue Polynomial 6
. . .
Memory for Residue Polynomial 0
Dyadic Core-3
Dyadic Core-2
Dyadic Core-1
Dyadic Core-0
Unified Butterfly and Dyadic Core-15
Unified Butterfly and Dyadic Core-2
Unified Butterfly and Dyadic Core-1
Unified Butterfly and Dyadic Core-0

Example RPAU. It uses 16 NTT butterfly cores and 4 coefficient-wise (dyadic) arithmetic cores. Polynomials are stored in ‘Memory’ made of BRAMs.

Mert et al. “Medha: Microcoded Hardware Accelerator for computing on Encrypted Data”. TCHES 2023

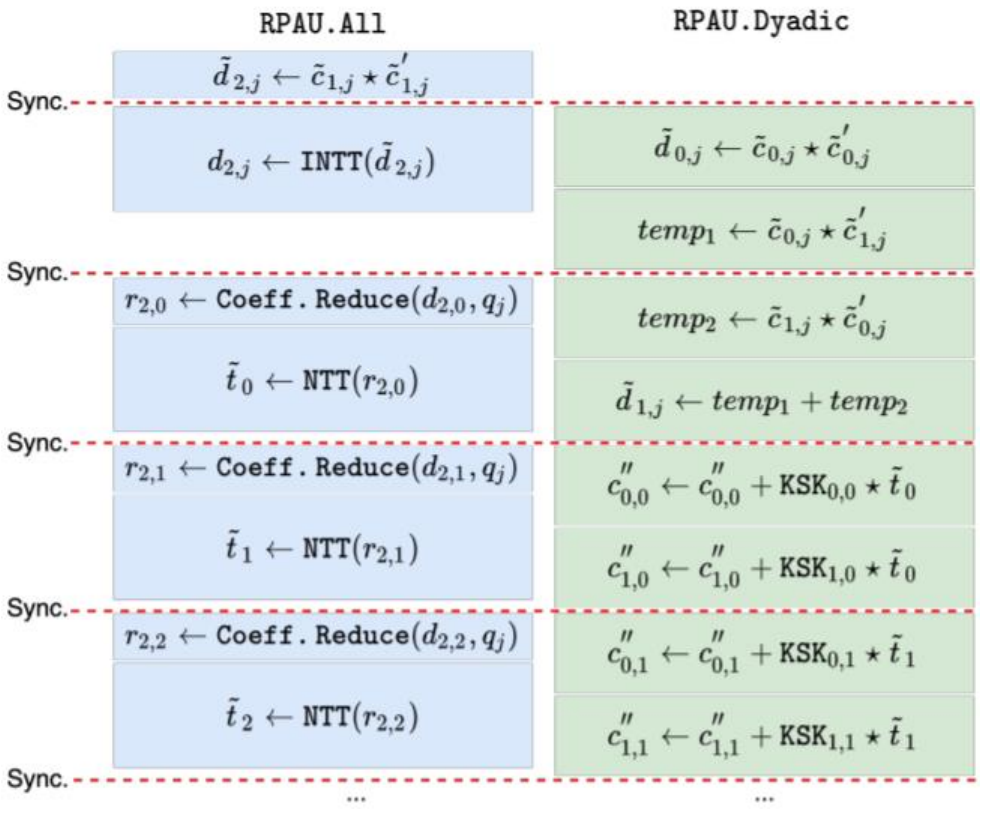
Instruction Parallelism in RPAU ( )



```
$\tilde{d}_{0,j} \leftarrow \tilde{c}_{0,j} \star \tilde{c}'_{0,j}$                HE.Mult
$\tilde{d}_{1,j} \leftarrow \tilde{c}_{0,j} \star \tilde{c}'_{1,j} + \tilde{c}_{1,j} \star \tilde{c}'_{0,j}$
$\tilde{d}_{2,j} \leftarrow \tilde{c}_{1,j} \star \tilde{c}'_{1,j}$
$\{c''_{0,j}, c''_{1,j}\} \leftarrow 0$                HE.Relin
$d_{2,j} \leftarrow \mathtt{INTT}(\tilde{d}_{2,j})$
for $i = 0$ to $L - 1$ do
    Obtain $d_{2,i}$ from RPAU$_i$
    $r_{2,i} \leftarrow \mathtt{Coeff.Reduce}(d_{2,i}, q_j)$
    $\tilde{t} \leftarrow \mathtt{NTT}(r_{2,i})$
    $c''_{0,i} \leftarrow c''_{0,i} + \mathtt{KSK}_{0,i} \star \tilde{t}$
    $c''_{1,i} \leftarrow c''_{1,i} + \mathtt{KSK}_{1,i} \star \tilde{t}$
end for
$(d_{0,j}, d_{1,j}) \leftarrow \lfloor c'' \cdot p^{-1} \rceil$
```

Homomorphic multiplication & key-switching.
(The most expensive operation)

Parallel execution of instructions

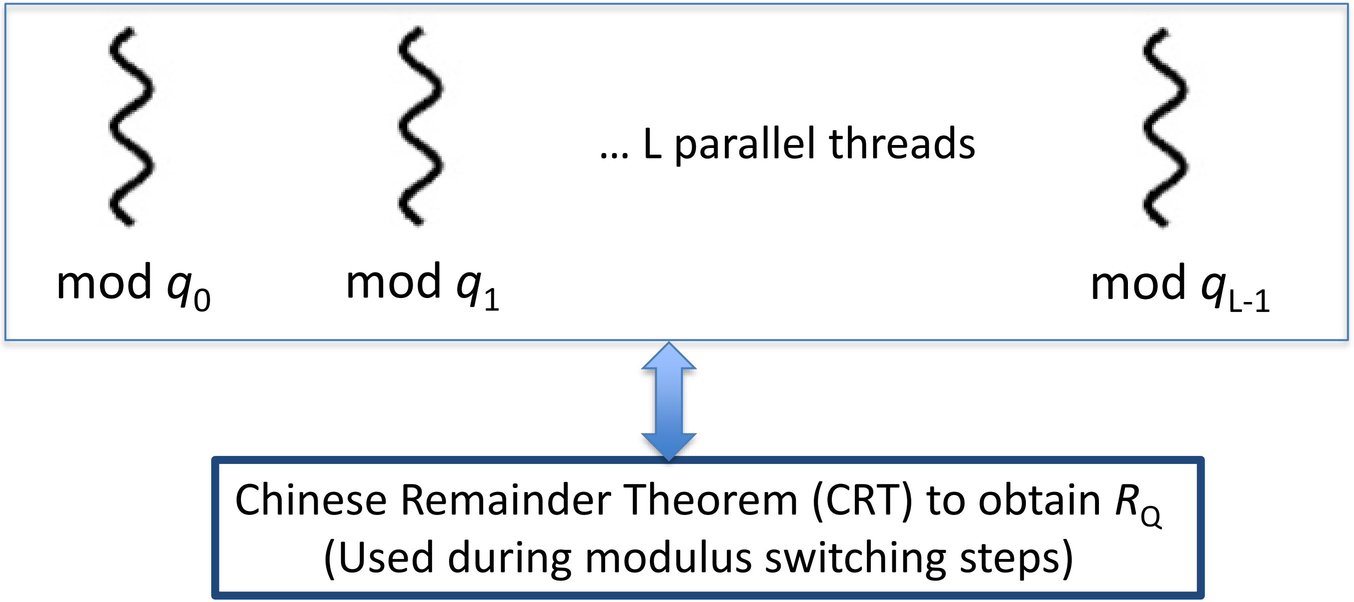

RPAU.All
RPAU.Dyadic
$\tilde{d}_{2,j} \leftarrow \tilde{c}_{1,j} \star \tilde{c}'_{1,j}$
Sync.
$d_{2,j} \leftarrow \mathtt{INTT}(\tilde{d}_{2,j})$
$\tilde{d}_{0,j} \leftarrow \tilde{c}_{0,j} \star \tilde{c}'_{0,j}$
$temp_1 \leftarrow \tilde{c}_{0,j} \star \tilde{c}'_{1,j}$
Sync.
$r_{2,0} \leftarrow \mathtt{Coeff.Reduce}(d_{2,0}, q_j)$
$\tilde{t}_0 \leftarrow \mathtt{NTT}(r_{2,0})$
$temp_2 \leftarrow \tilde{c}_{1,j} \star \tilde{c}'_{0,j}$
$\tilde{d}_{1,j} \leftarrow temp_1 + temp_2$
Sync.
$r_{2,1} \leftarrow \mathtt{Coeff.Reduce}(d_{2,1}, q_j)$
$\tilde{t}_1 \leftarrow \mathtt{NTT}(r_{2,1})$
$c''_{0,0} \leftarrow c''_{0,0} + \mathtt{KSK}_{0,0} \star \tilde{t}_0$
$c''_{1,0} \leftarrow c''_{1,0} + \mathtt{KSK}_{1,0} \star \tilde{t}_0$
Sync.
$r_{2,2} \leftarrow \mathtt{Coeff.Reduce}(d_{2,2}, q_j)$
$\tilde{t}_2 \leftarrow \mathtt{NTT}(r_{2,2})$
$c''_{0,1} \leftarrow c''_{0,1} + \mathtt{KSK}_{0,1} \star \tilde{t}_1$
$c''_{1,1} \leftarrow c''_{1,1} + \mathtt{KSK}_{1,1} \star \tilde{t}_1$
Sync.
...
...

This reduces 40% cycle count

Placement of RPAUs

CRT requires combining the residues.
→ Therefore, RPAUs need to communicate with each other

How to interconnect the RPAUs in large 3D FPGAs?

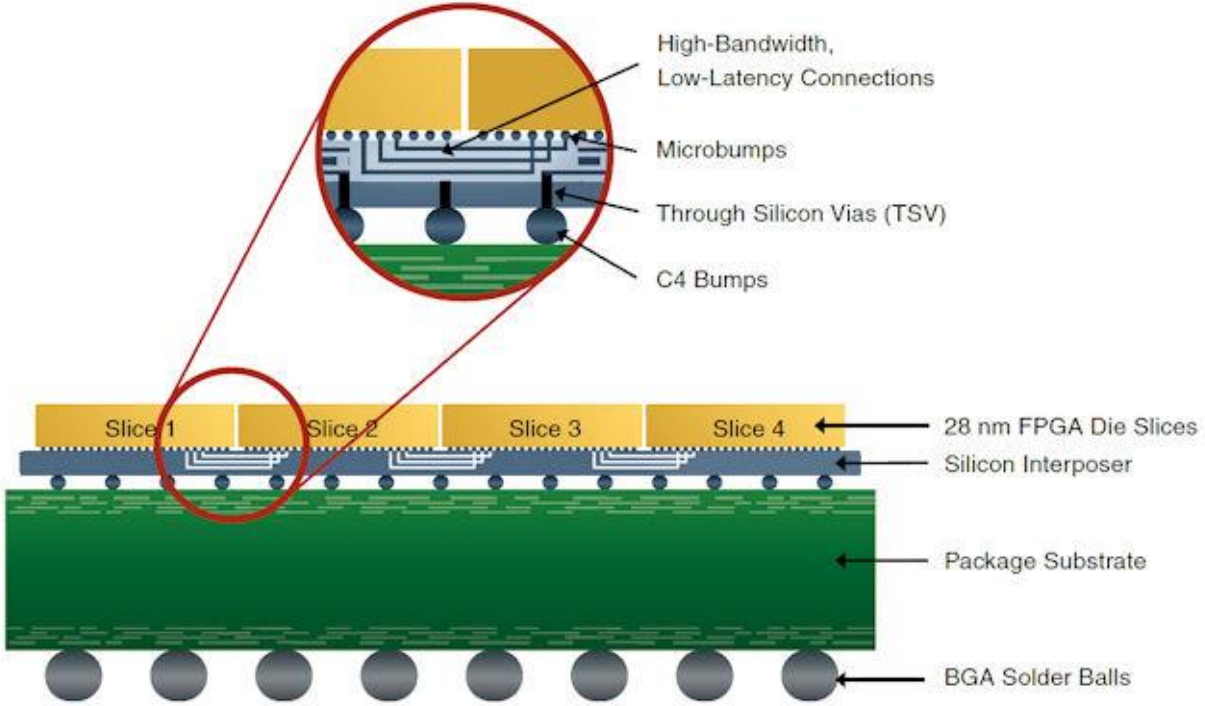


Chinese Remainder Theorem (CRT) to obtain R_Q
(Used during modulus switching steps)

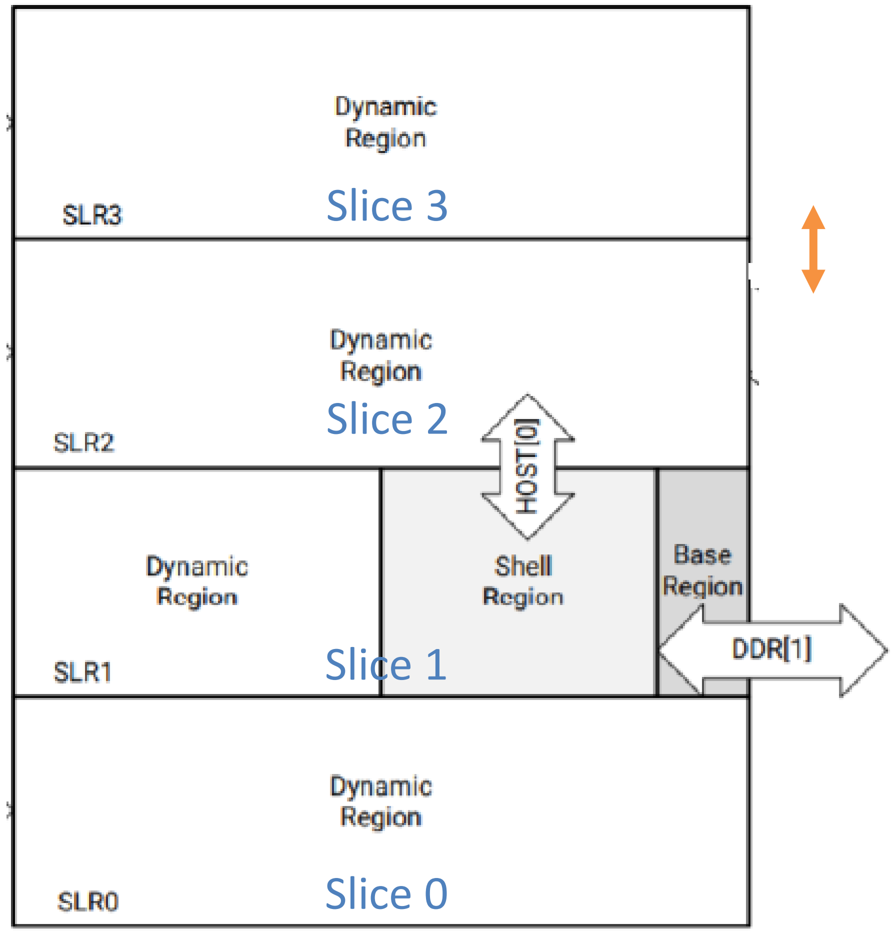
Large SLR FPGA

Large FPGAs are multi-die

- ➢ The FPGA is split into four SLRs.
- ➢ Connected by a limited number of wires.



Large SLR FPGA – top view



There are a limited number of interconnects.

Large design cannot be spread arbitrarily across SLRs.

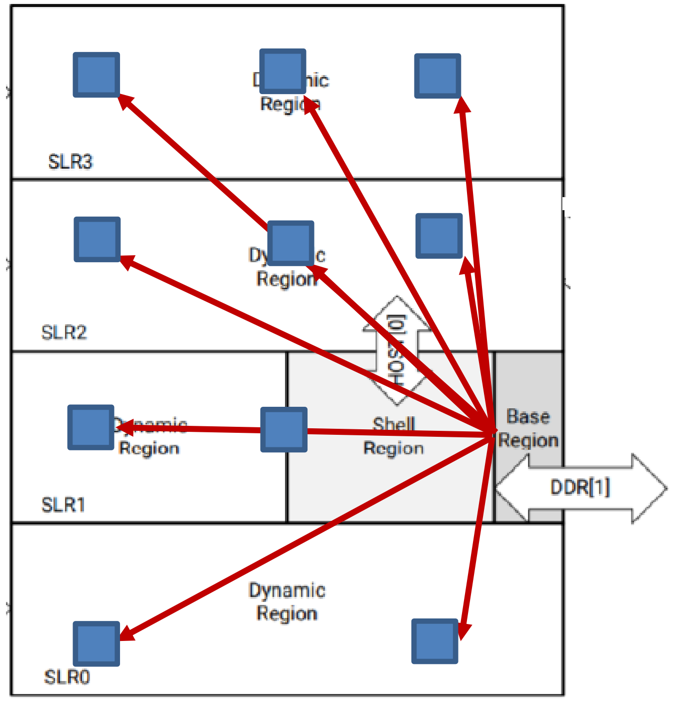
Xilinx Alveo U250 FPGA. This FPGA is 1000x larger than the FPGA used in this course.

Placement-friendly interconnection of RPAUs

- FPGA Constraints
  - ➢ The FPGA is split into four SLRs.
  - ➢ Connected by a limited number of wires.

- Some operations require exchanging the residue polynomials between RPAUs

- Naïve solution: A ”star-like” network

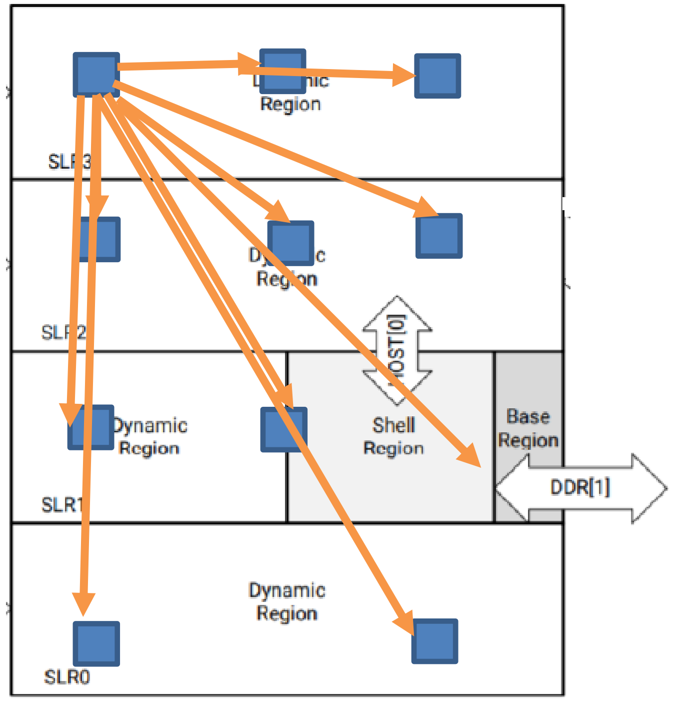


🟦 One RPAU

Placement-friendly interconnection of RPAUs

- FPGA Constraints
  - ➢ The FPGA is split into four SLRs.
  - ➢ Connected by a limited number of wires.
- Some operations require exchanging the residue polynomials between RPAUs
- Naïve solution: A ”star-like” network

Each RPAU has its own connections



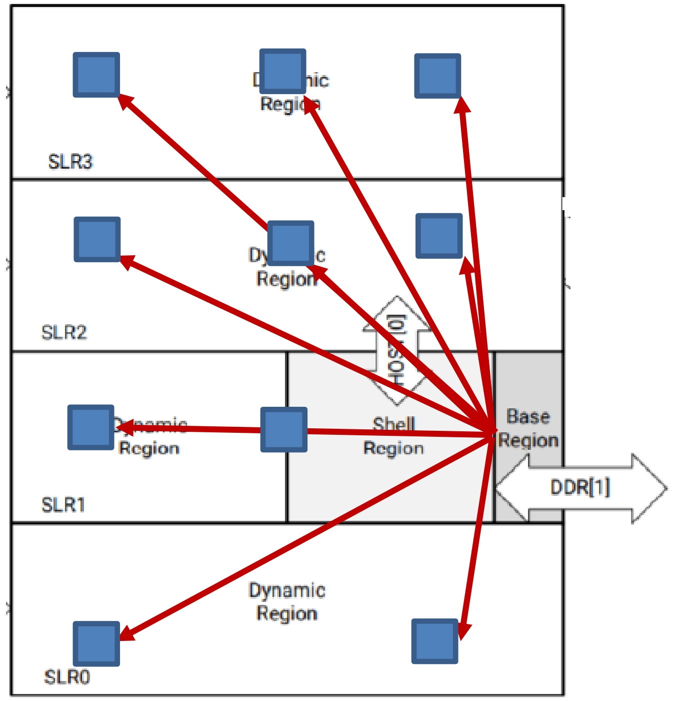
🟦 One RPAU

Placement-friendly interconnection of RPAUs

- FPGA Constraints
  - ➢ The FPGA is split into four SLRs.
  - ➢ Connected by a limited number of wires.

- Some operations require exchanging the residue polynomials between RPAUs

- Naïve solution: A ”star-like” network



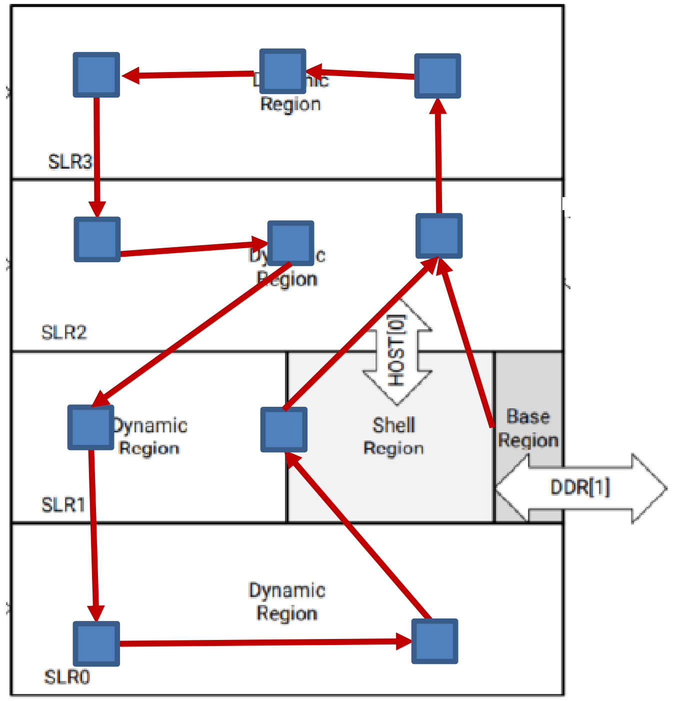
- Complicates the routing
- Large number of nets crossing the SLRs
- Reduces the clock frequency to around 50 MHz or less

Placement-friendly interconnection of RPAUs

- FPGA Constraints
  - The FPGA is split into four SLRs.
  - Connected by a limited number of wires.

- Some operations require exchanging the residue polynomials between RPAUs

- Solution: A "ring" interconnection of RPAUs



- Only two neighbour RPAUs are connected.
- Data sent to an RPAU through a chain of RPAUs.
- No additional computation overhead

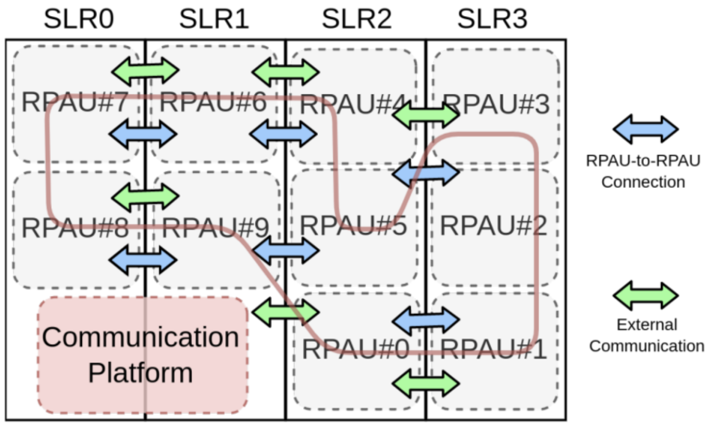
Mert et al. "Medha: Microcoded Hardware Accelerator for computing on Encrypted Data". TCHES 2023

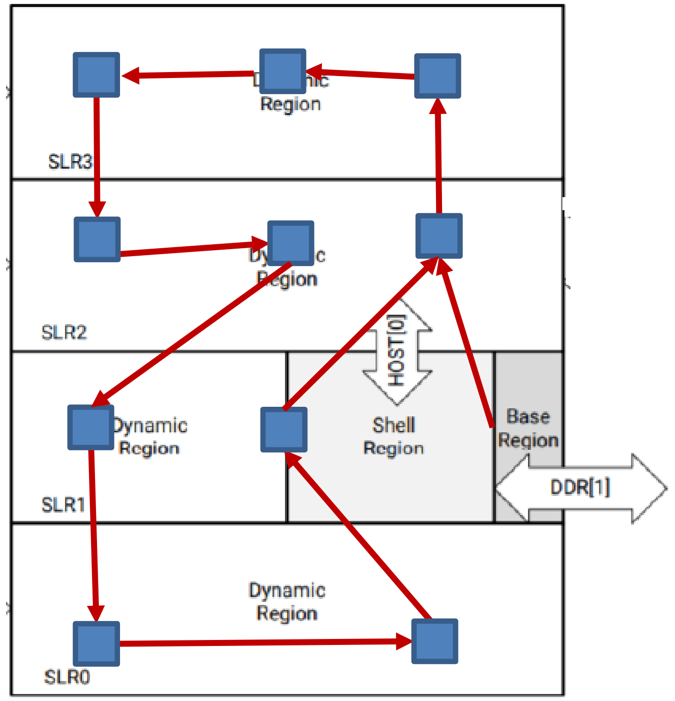
Placement-friendly interconnection of RPAUs

- FPGA Constraints
  - The FPGA is split into four SLRs.
  - Connected by a limited number of wires.

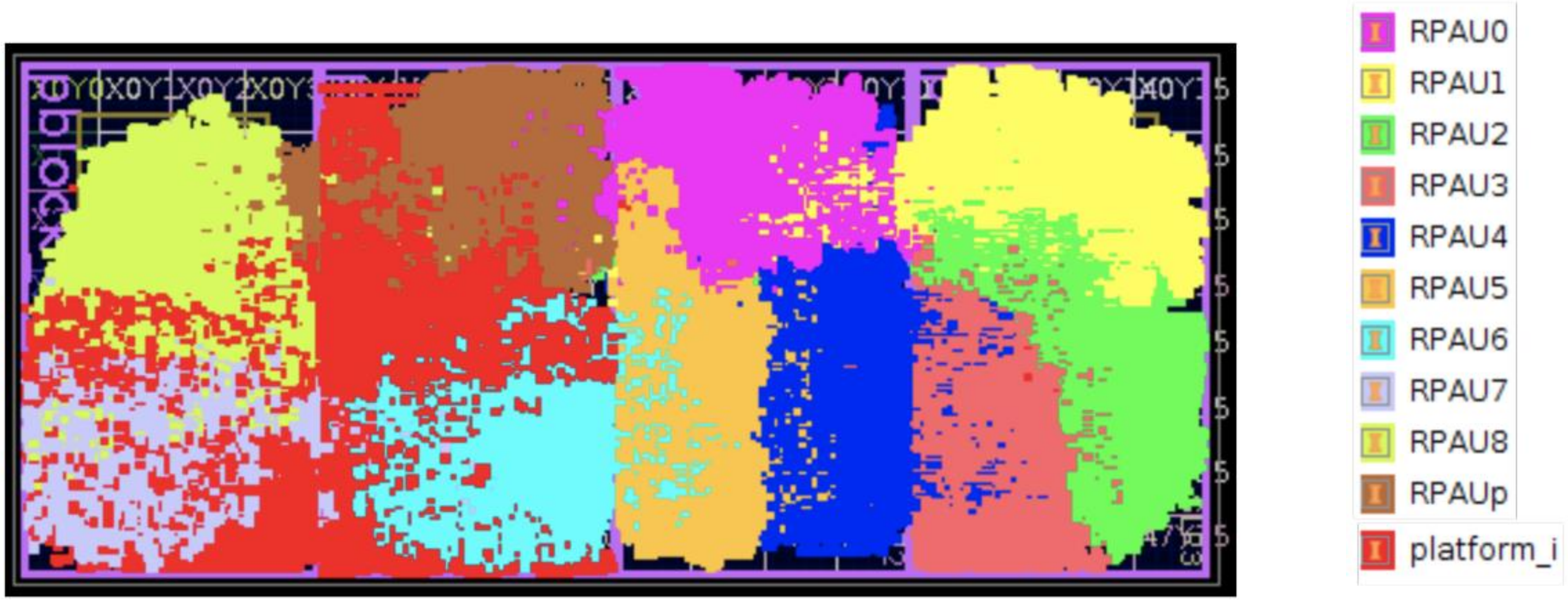
- Some operations require exchanging the residue polynomials between RPAUs

- Placement of 10 RPAUs using “ring” interconnect





Floorplan of the design



Full system overview

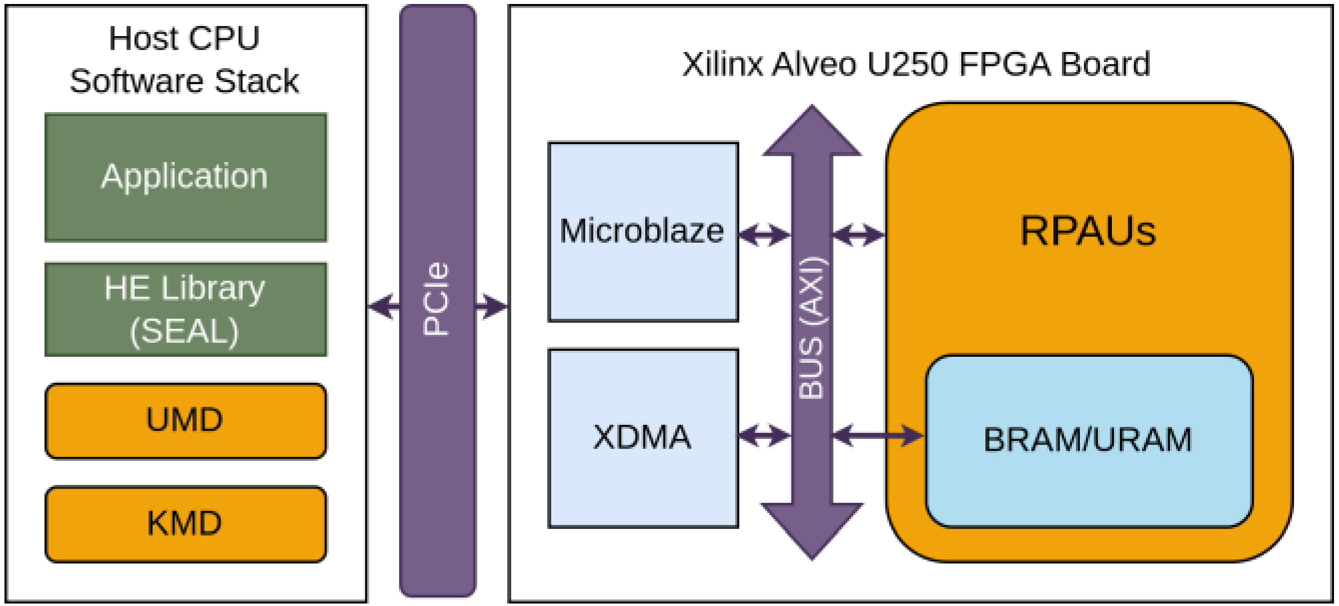


Figure 8: CPU-FPGA interface and software stack

FPGA is used as an accelerator card of a server. HW/SW codesign is used to run applications.

Mert et al. "Medha: Microcoded Hardware Accelerator for computing on Encrypted Data". TCHES 2023

FPGA Acceleration results

$$\begin{tikzcd}
\textit{foo}(\text{data}) \arrow[r] & \textit{foo}(\text{Enc}(\text{data})) \\
\text{Takes 1s} & \text{Takes } 10^4 \text{ to } 10^5 \text{ s} \arrow[d, "\text{Overhead down to}"] \\
 & 10^2 \text{ to } 10^3 \text{ s}
\end{tikzcd}$$

Mert et al. “Medha: Microcoded Hardware Accelerator for computing on Encrypted Data”. TCHES 2023

Our Group’s research: Open Problems in FHE

1. How to make hardware accelerators for larger parameter sets?

2. How to support different parameters?

3. How to support different FHE schemes?

4. How to implement FHE Bootstrapping?

5. From FPGA to ASIC accelerators
   - More parallel processing
   - Custom memory
   - Higher clock frequency and lower power consumption