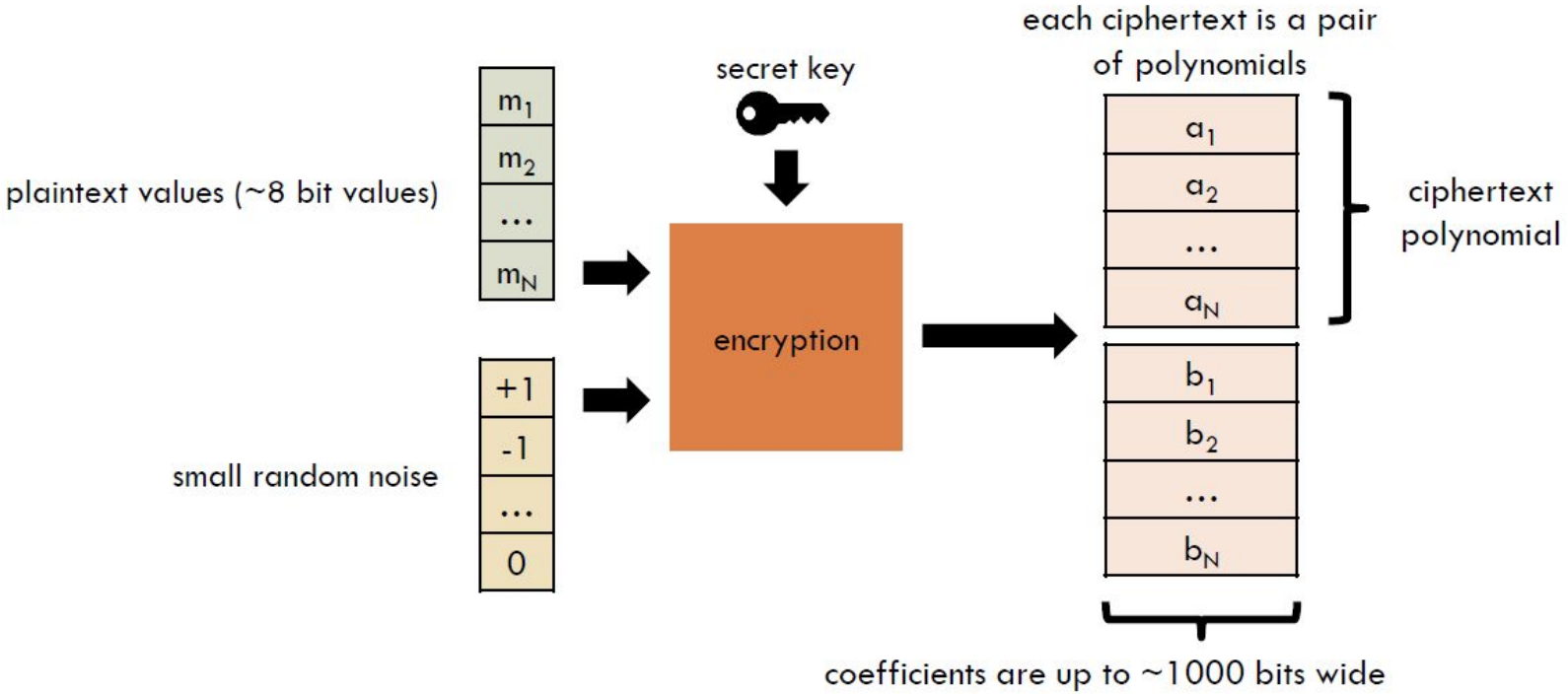
Fully Homomorphic Encryption

Presenter: Yihan Jiang, Yahya Alhinai

Research Motivation and Goal

- Long term project introduced by Jianming about Fully Homomorphic Encryption.
- We will present the initial works for this long term project as the final project.
- A survey paper about the bottleneck of existing fully homomorphic encryption designs. (By September 2022)
- An accelerator on HLS for the full homomorphic encryption based on the profiling results. (During Fall 2022)

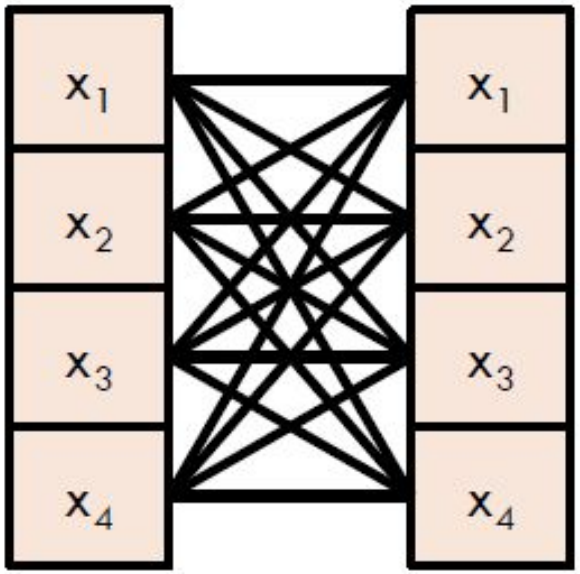
Encryption


each ciphertext is a pair of polynomials
secret key
plaintext values (~8 bit values)
m_1
m_2
...
m_N
small random noise
+1
-1
...
0
encryption
a_1
a_2
...
a_N
ciphertext polynomial
b_1
b_2
...
b_N
coefficients are up to ~1000 bits wide

Multiplying Polynomials

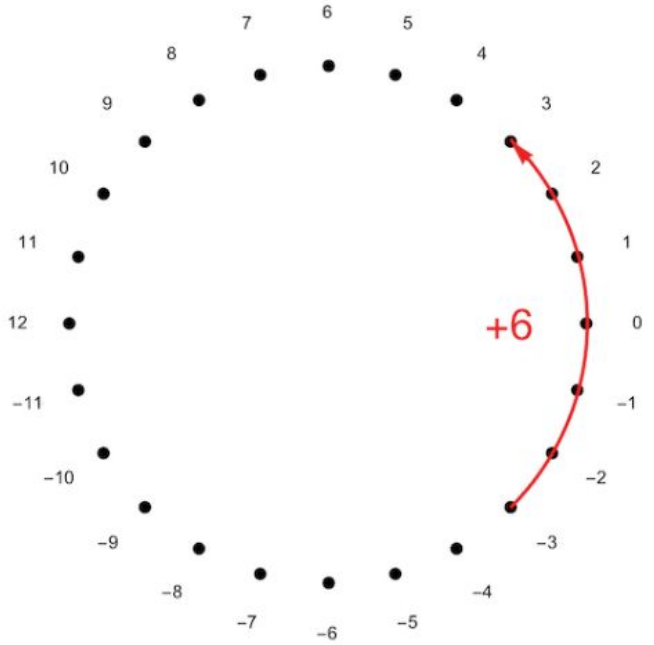
Fast implementations of homomorphic encryption schemes depend on efficient polynomial arithmetic.

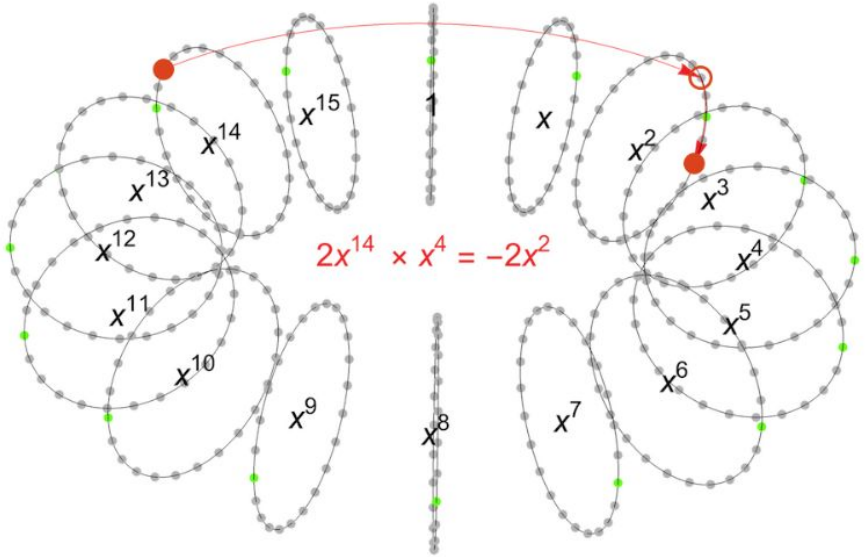
The most time-consuming operation in polynomial arithmetic is high-order polynomial multiplication.



Naively, this takes $O(n^2)$ multiplications

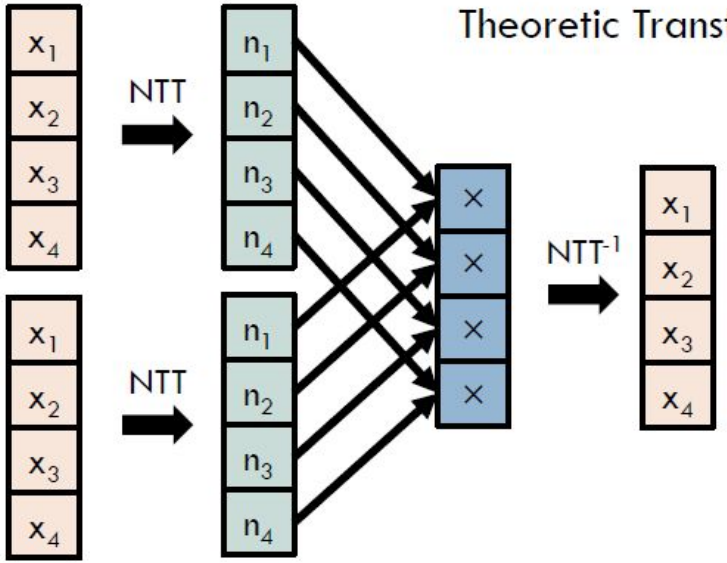
Polynomial Rings





Number-Theoretic Transform (NTT)

Instead, we can use the Number Theoretic Transform (NTT)



NTTs and NTT^{-1} each take O(nlogn) multiplies, making the whole operation O(nlogn)

HLS Implementation

The NTT designed in this application has 10 stages and each stage has its own unique number of loop variable. For this reason, stage special cycle numbers belonging to 10 stages are determined for 10 different functions, and each stage is divided into 3 separate stages as reading index, calculation, and modular calculation.

```
// stage0
read_index_0(a_index,b_index);
b_calculation_0(out_y,xin,a_index,b_index);

// stage 1
 read_index_1(a_index,b_index);
 b_calculation_1(b_out, out_y,b_index, xx);
 mod_cal_all(out_x, out_y,a_index,b_index, b_out);

//stage2
 read_index_2(a_index,b_index);
 b_calculation_2( b_out, out_x,b_index, xx);
 mod_cal_all(out_y, out_x,a_index,b_index, b_out);


 //stage3
 read_index_3(a_index,b_index);
 b_calculation_3( b_out, out_y,b_index, xx);
 mod_cal_all(out_x, out_y,a_index,b_index, b_out);

 //stage4
 read_index_4(a_index,b_index);
 b_calculation_4( b_out, out_x,b_index, xx);
 mod_cal_all(out_y, out_x,a_index,b_index, b_out);
```

```
void b_calculation_4(uint64_t b_out[512],uint64_t x[1024],int b_index[512],uint64_t xx[1024]){


    int wbarr = 15;
    uint64_t wb = 1;
    int count=0;
    b_calculation_4: for (int t = 0; t < 16; t++) {
                for (int trans = 0; trans < 32; trans++) {

                    b_out[count] = c(x[b_index[count]], wb);
                     count++;


            }

            wb = xx[wbarr];
            wbarr++;

        }
```

```
void read_index_5(int a_index[512],int b_index[512]){

    int count=0;

        int trans_size=64;
        read_index_5:for (int t = 0; t < 32; t++) {
                    for (int trans = 0; trans <16; trans++) {

                        int i = trans * trans_size + t;

                        int j = i + (trans_size >> 1);
                             a_index[count]=i;
                             b_index[count]=j;
        count++;
                    }}

}
```

```
void mod_cal_all(uint64_t out_x[1024],uint64_t x[1024],int a_index[512],int b_index[512],uint64_t b_out[512]){

        mod_cal_1: for (int t = 0; t < 512; t++) {
         #pragma HLS unroll factor=4

                    out_x[a_index[t]] = mod_add(x[a_index[t]], b_out[t], 0xffffffff00000001);
                    out_x[b_index[t]] = mod_sub(x[ int a_index[512]                t], 0xffffffff00000001);
                }
                    Press 'F2' for focus


}
```

Results

Testing **1024-wide vector** with **32-bit**

Baseline: SEAL software 46.4µs

Vitis HLS: 4.316µs (speedup x10.7)

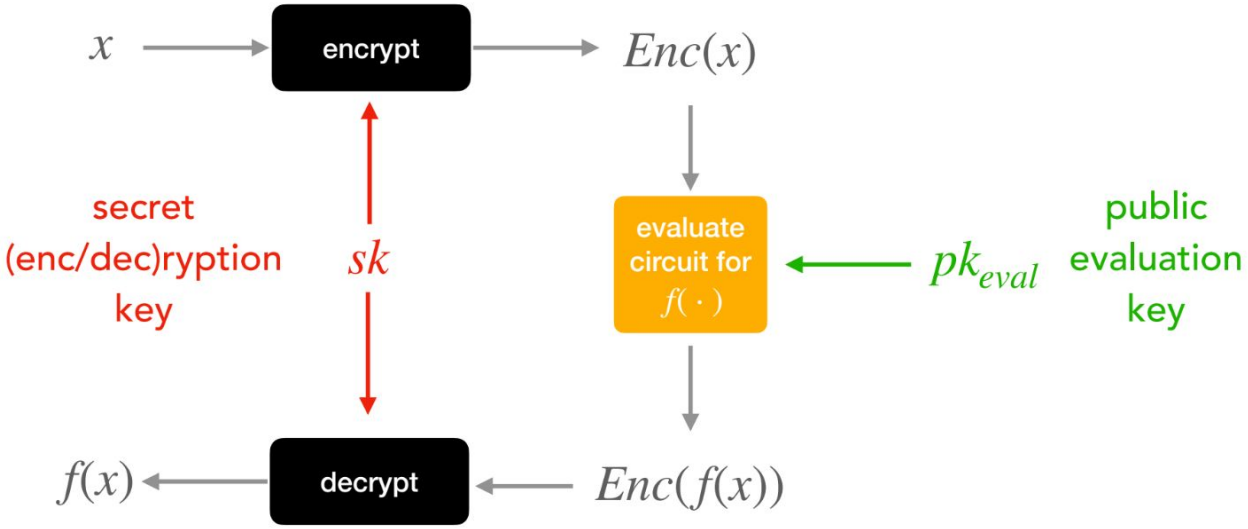
	Latency(µs)	BRAM_18K	DSP	FF	LUT
Original	29.74	73%	27%	1.1%	9%
Pipeline output in NTT + Parallel tail factor of 4	9.19	10%	21%	22%	41%
Optimized load and save functions (#pragma HLS INLINE)	8.37	47%	33%	1.1%	43%
Optimized load and save (#pragma HLS INLINE) + Optimized partial buff store pipeline and parallel tail factor of 2	7.57	47%	34%	1.1%	40%
Optimized load and save (#pragma HLS INLINE) + Optimized partial buff store pipeline and parallel tail factor of 2 + Pipeline output and Parallel tail factor of 4	**4.316**	52%	132%	1.1%	48%

General Introduction of FHE

- Symmetric FHE
- Asymmetric FHE
- Strengths
- Challenges

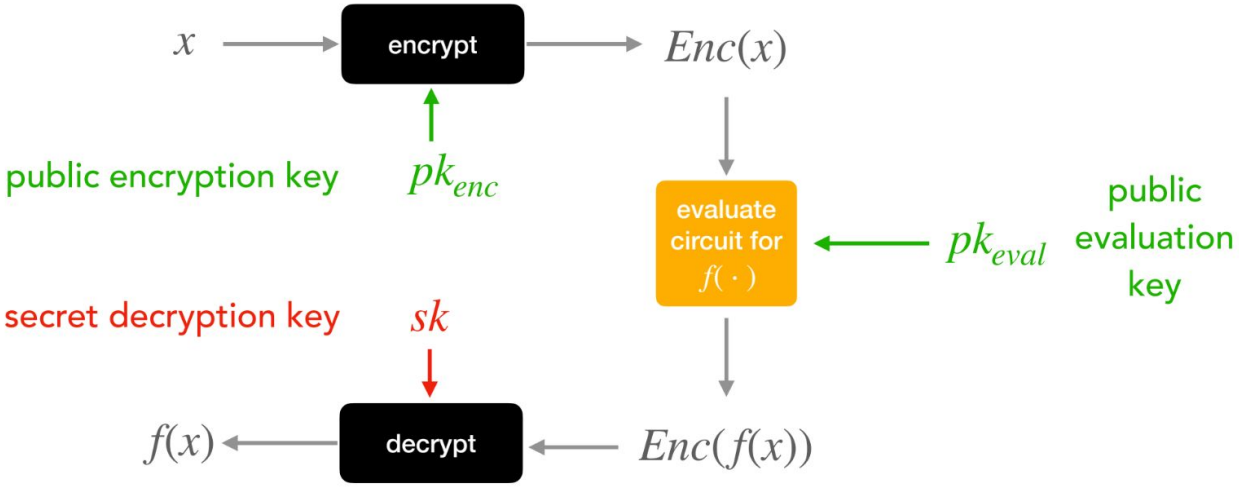
General Introduction of FHE

- **Symmetric FHE**
- Asymmetric FHE
- Strengths
- Challenges



General Introduction of FHE

- Symmetric FHE
- **Asymmetric FHE**
- Strengths
- Challenges



General Introduction of FHE

- Symmetric FHE
- Asymmetric FHE
- **Strengths**
- Challenges

- Companies could offer services without having access to their users' data.
- Localization becomes unnecessary. Companies could save a lot on cloud infrastructure costs.

General Introduction of FHE

- Symmetric FHE
- Asymmetric FHE
- Strengths
- **Challenges**

- Computation complexity.
- Early FHE schemes were about 10^9 times slower than performing computations on unencrypted data.
- Recent advanced FHE versions are still 1000x to 100000x slower than unencrypted computation when executed in carefully optimized software.

GAZELLE: a low-latency framework for secure neural network inference

- Major Contributions

  1) Employs a much simpler packed additively homomorphic encryption (PAHE) scheme, which can support very fast matrix-vector multiplications and convolutions.

  2) Combine Lattice-based AHE schemes with powerful features (SIMD evaluation and automorphisms). Make them the ideal tools for common linear-algebraic computations.

F1 - the first programmable FHE accelerator

- Major Contributions

  1) Unlike prior accelerators that built fixed pipelines tailored to specific FHE schemes and parameters, F1 allows the same hardware to accelerate all operations within arbitrary FHE programs, and even multiple FHE schemes.

  2) Achieve ASIC-level performance without sacrificing programmability.

- Weaknesses

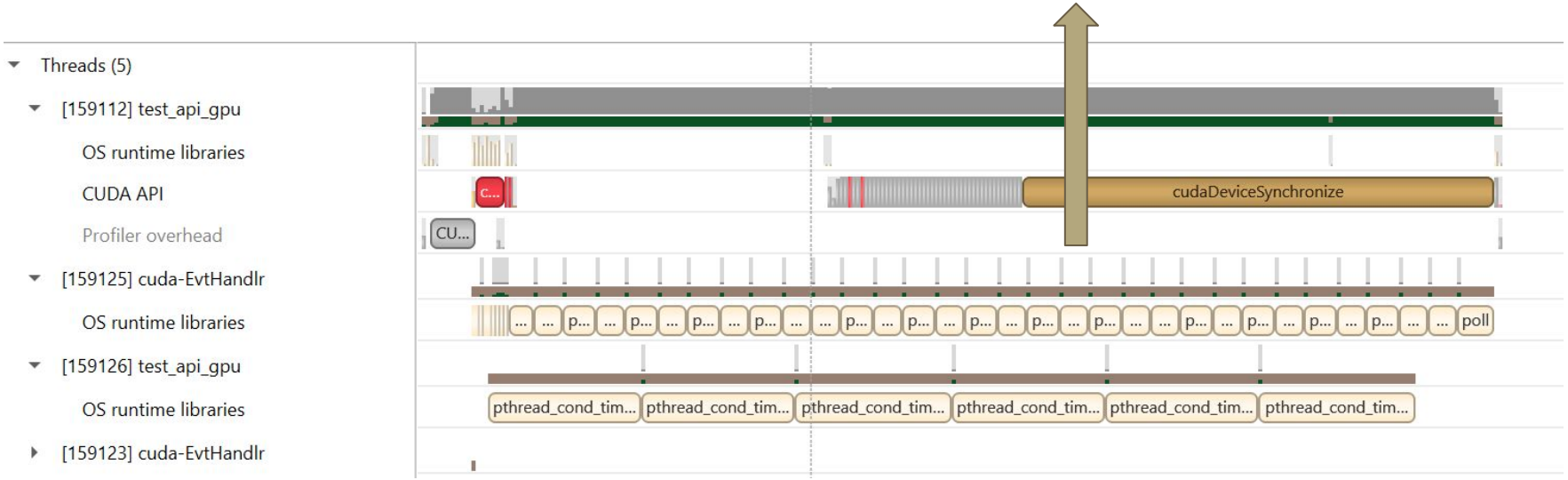
  1) The design only focused on improving the overhead created by data movement because it is the major issue. It is still worth to explore some other aspects. (ex: a better FU utilization design).

cuFHE: an open-source library for FHE on CUDA-enabled GPU

- General
- GPU

cuFHE

- **General**
- GPU



cuFHE

- General
- **GPU**