Accelerating Homomorphic Encryption

Accelerator Architecture

Noelle Crawford[*]
noellec3@illinois.edu
University of Illinois at Urbana-Champaign

Marcanthony Huang[*]
mg25@illinois.edu
University of Illinois at Urbana-Champaign

Gregory Jun[*]
hgjun2@illinois.edu
University of Illinois at Urbana-Champaign

Arpan Raja[*]
arpanr2@illinois.edu
University of Illinois at Urbana-Champaign

Wesley Wu[*]
wwu70@illinois.edu
University of Illinois at Urbana-Champaign

ABSTRACT

Fully Homomorphic Encryption (FHE) enables mathematical transformations on encrypted data to enable direct computation on sensitive information. High-performance implementation of these techniques, including specialized hardware, is required for FHE adoption. This requires that the main bottlenecks, in particular polynomial multiplication, need to be very efficient. A widely adopted optimization for FHE schemes uses Number Theoretic Transform (NTT), which is similar to the fast Fourier Transform (FFT), to speed up the computation of polynomial multiplication. However, such a method has a high memory access overhead and parallelism barriers caused by rigid data dependencies. We propose a hardware accelerator architecture that seeks to improve the performance of polynomial multiplication around these issues. Other polynomial multiplication methodologies will also be considered, such as Karasuba's algorithm as well as the use of convolutional systolic arrays.

1 INTRODUCTION

In a world becoming increasing more connected through IoT technology there has been a growing fear regarding the personal privacy and security of the sensitive information. For example in the medical field it is necessary patient information to be sent to a server in order to be processed to calculate health related risks. Unfortunately there is no secure way to ensure the data is protected. Even if the data is encrypted the server performing computation needs to decrypt the data, only protecting the data fro middle man attacks. There is no way to ensure security from a comprised or malicious cloud computer. To illustrate this point; myChart, a healthcare service that holds onto personal medical information of patients for hospitals, has come under fire for selling private information to different social media companies. Patients have had little to no ability to fight back or even hold these people responsible, creating a need for ensured patient privacy. In order to try and solve this problem recent researchers have suggested a new form of encryption known as Homomorphic encryption.

Homomorphic encryption is a relatively new style of encryption in which operations performed on the encrypted data are correctly propagated when the result is decrypted. Specifically, fully homomorphic encryption schemes (FHE) support the computation of any homomorphic function, including the common arithmetic operations ($+, -, \times$).

Unfortunately this form of encryption results in even the most basic computation taking significantly computational resources. In its current state Homomorphic encryption can be millions of times slower resulting in it being unusable. To alleviate this concern we are suggesting a novel accelerator in order to make the computation on encrypted data to be closer to its trivial counterpart.

2 BACKGROUND

Multiple schemes implementing forms of homomorphic encryption exist. For this course, we will base our algorithm on the BFV homomorphic encryption scheme [1]. The algorithm can be divided into the following parts:

(1) **Represent integer operands as polynomials.** For this scheme, we use a binary counting system to represent our operands. For any integer k there is a unique representation as a polynomial $f(x) = \sum_{i \in \mathbb{N}} b_i x^i$, where $b_i \in \{0, 1\}$, $B = \{i \mid b_i = 1\}$ such that $\sum_{i \in B} 2^i = k$.

```
>>> f = BFV.IntEncode(36)
* f(x): 0 + 0*x^1 + 1*x^2 + 0*x^3 + 0*x^4 + 1*x^5 + 0*x^6
    + 0*x^7 + ...
```

(2) **Encrypt polynomials to generate ciphertext.** The general process of encrypting the polynomials is as follows. First, the secret key (sk) and public key (pk) are generated, with sk being a randomized polynomial, and $pk = (-(a * sk + e), a)$ where a, e are randomized polynomials. Then, for every polynomial message a user wants to decrypt, three more randomized polynomials are sampled (u, e_1, e_2) and used to compute $ct = ([p_0 \cdot u + e_1 + \Delta \cdot m]_q, [p_1 \cdot u + e_2]_q)$, where Δ is a parameterized scaling factor. The security of the BFV encryption scheme is guaranteed by a reduction to the ring learning with errors (RLWE) problem, so here e_1, e_2 provide the error 'noise' that prevents an observer from deducing the original data.

```
>>> ct1 = BFV.Encrypt(f)
* ct1[0]: 96933556 + 113071479*x^1 + 99610246*x^2
    + 69795252*x^3 + 41625873*x^4 + 52960741*x^5
    + 58991265*x^6 + 11425165*x^7 + ...
* ct1[1]: 95729437 + 97548977*x^1 + 96082620*x^2
```

[*] All authors contributed equally to this work. (Alphabetical order)

```
    + 20280378*x^3 + 11209257*x^4 + 48489375*x^5
    + 126626183*x^6 + 123631717*x^7 + ...
```

(3) **Perform computations.** This is the most time consuming part of the process, and where we aim to focus our efforts on accelerating. The BFV paper implements three homomorphic operations: addition, subtraction, and multiplication. Addition and subtraction are directly implemented as the corresponding polynomial operations, that is, to calculate $a \pm b$, one can simply calculate $f_{ct}(x) = f_a(x) \pm f_b(x)$. Simple multiplication can be implemented similarly, so $f_{ct}(x) = f_a(x) \cdot f_b x$, where $f_a(x) = (c_{a0} + c_{a1} \cdot x)$ and $f_a(x) = (c_{b0} + c_{b1} \cdot x)$. Notice that the corresponding output will be of the form $f_{ct}(x) = (c'_0 + c'_1 \cdot s + c'_2 \cdot s^2)$, a ciphertext tuple with 3 terms.

```
>>> ct_mul = BFV.HomomorphicMultiplication(ct1, ct2)
* ct_mul[0] :125079869 + 27681377*x^1 + 102746964*x^2
    + 24688598*x^3 + 31211437*x^4 + 16998782*x^5
    + 72481926*x^6 + 44003211*x^7 + ...
* ct_mul[1] :89160054 + 557156*x^1 + 1658775*x^2
    + 8245064*x^3 + 50189253*x^4 + 99013323*x^5
    + 115397445*x^6 + 25523533*x^7 + ...
* ct_mul[2] :85458786 + 16612315*x^1 + 70099140*x^2
    + 91969849*x^3 + 61811045*x^4 + 61146088*x^5
    + 124566650*x^6 + 52629530*x^7 + ...
```

(4) **Relinearization** Since the output of a multiplication operation is a ciphertext with 3 terms as opposed to 2, the output cannot be directly decrypted or operated on. To solve this, the BFV paper employs a technique called 'relinearization', which transforms the output into a ciphertext package of two polynomials. Consider some output ciphertext $ct_i = (c_0, c_1, c_s)$, then this represents a value $[c_0 + c_1 \cdot s + c_2 \cdot s^2]_q$. We can then solve for a pair $ct_f = (c_{0f}, c_{1f})$ such that $ct_f = [c_{0f} + c_{1f} \cdot s + r]_q$ for some small r.

```
>>> ct_mul = BFV.Relinearization(ct_mul)
* ct_mul[0] :90700989 + 10538200*x^1 + 25462045*x^2
    + 125874164*x^3 + 107475309*x^4 + 12560202*x^5
    + 86606375*x^6 + 94588971*x^7 + ...
* ct_mul[1] :90801717 + 120114893*x^1 + 79551519*x^2
    + 45379272*x^3 + 89211627*x^4 + 11961111*x^5
    + 119335190*x^6 + 359410*x^7 + ...
```

(5) **Decrypt resulting polynomial and recover integer result.** Once the desired operations have been performed, the ciphertext can be decrypted to recover and decode the original polynomial and value. The decrypted polynomial can be calculated as $f_m(x) = [\lfloor \frac{t \cdot [c_0 + c_1 \cdot s]_q}{q} \rceil]_q$, and then decoded by evaluating the summation as described in the integer encoding step.

In general, since we are operating on polynomials, most operations should be trivial to parallelize in our accelerator. However, our focus for this project will specifically be on accelerating the multiplication operation through the efficient use and optimization of NTTs (number theoretical transform). Prior work has shown that multiplication operations consume up to 70% of execution time [2]., making it an excellent candidate for the focus of our project.

3 KERNELS

NTT Kernel Code:

```
void naive_ntt(int * a, int * b, int n, int k, int mod) {
    int q = n * k + 1;
    assert(is_prime(q) == 1);
    int x = primitive_root(q);
    int w = power(x, k, mod);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            b[i] += a[j] * power(w, i * j, mod);
        }
        b[i] %= mod;
    }
}
void naive_intt(int * a, int * b, int n, int k, int mod) {
    int q = n * k + 1;
    assert(is_prime(q) == 1);
    int x = primitive_root(q);
    int w = power(x, k, mod);
    int mod_inverse_w = find_mod_inverse(w, mod);
    int mod_inverse_n = find_mod_inverse(n, mod);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            b[i] += a[j] * power(mod_inverse_w, i * j, mod);
        }
        b[i] %= mod;
        b[i] *= mod_inverse_n;
        b[i] %= mod;
    }
}

void ntt_multiply(int * a, int * b, int * c, int n, int k, int mod) {
    for (int i = 0; i < n; i++) {
        c[i] = (a[i] * b[i]) % mod;
    }
}
```

Polynomial multiplication calculations account for a majority of FHE execution, with some sources stating as much as 70%. A naive implementation of polynomial multiplication leads to a time complexity of O(N^2), where N is the degree of the polynomials being multiplied. A popular algorithm-level optimization for polynomial multiplication is the Fast Fourier Transform (FFT)/Number Theoretic Transform (NTT). In order to handle extremely large degree polynomials, numerous accelerators have attempted to speed up this method, which offers one of the lowest algorithmic complexities in polynomial multiplication. However, NTT has a high memory access overhead and parallelism barriers caused by rigid data dependencies. Other approaches to multiplying polynomials, such Karatsuba's algorithm or a convolutional systolic array of processing elements, have been suggested. When compared to NTT, these techniques handle lower degree polynomial multiplications and use simpler accelerator designs to accommodate lighter workloads. In comparison to NTT accelerators, these techniques utilize less hardware resources, which increases the potential of accelerated encryption on embedded devices or less expensive FPGAs. These options will be further investigated in the development of our accelerator.

The NTT is essentially a generalization of the DFT, except instead of using twiddle factors in the complex plane, it generates twiddle factors from a quotient ring, chosen by the user.

Calculating the NTT is as follows:

(1) **Obtain input polynomial.** Obtain the polynomial of length n in vector form.

(2) **Select minimum mod base M.** Choose a minimum working modulus M s.t. $1 \leq n < M$ and every value in the polynomial vector is in the range $[0, M)$.
(3) **Select coprime number N.** Let $N = kn + 1$ using some arbitrary $k \geq 1$ s.t. $N \geq M$ and N is prime.
(4) **Find generator of $\mathbb{Z}_N$.** Find all of the prime factors of $N-1$, then guess an arbitrary integer a s.t. $\forall$ prime factors p of $N-1$, $a^{\frac{N-1}{p}} \neg \equiv 1 \; mod \; N$.
(5) **Calculate primitive root ω.** Calculate $\omega = g^k \equiv r \; mod \; N$. This will be the primitive root of unity we'll use as the twiddle factor for the transform.
(6) **Compute output vector Y.** Using the twiddle factor and input polynomial, calculate the output vector Y using the formula as follows:
$$Y(i) = \sum_{j=0}^{n-1} X(i) * \omega^{i*j} \qquad \forall \; i \in [0, n)$$

Calculating the inverse NTT is as similar to the forward NTT, except using the multiplicative inverse mod N from the forward:

(1) **Find multiplicative inverse ω^{-1}.** Find a $\omega^{-1} \in (0, N]$ s.t. $\omega * \omega^{-1} \equiv 1 \; mod \; N$
(2) **Computer unscaled input vector Xn.** Using the following formula, compute Xn:
$$X(i) * n = \sum_{j=0}^{n-1} Y(i) * \omega^{-i*j} \qquad \forall \; i \in [0, n)$$
(3) **Re-scale the input vector.** Find n^{-1} s.t. $n{*}n^{-1} \equiv 1 \; mod \; N$, then element-wise multiply each element of Xn by n^{-1}

To multiply two polynomials in the NTT frequency domain, simply perform an element-wise multiplication of the transformed polynomials modulo N, then perform an inverse NTT to get the result back to polynomial space.

For sufficiently big operands, the Karatsuba algorithm provides a strategy for polynomial multiplication that enables it to outperform the naive method. The procedure starts by splitting up polynomials A and B into smaller polynomials called A_H, A_L, B_H, and B_L, respectively. Therefore, the operands can be written as follows:
$$A = A_L + A_H x^{n/2}, B = B_L + B_H x^{n/2}$$
The multiplication of the two terms leads to this expansion:
$$A * B = A_H B_H x^n + (A_L B_H + A_H B_L) x^{n/2} + A_L B_L$$
The middle term can be reduced:
$$A_L B_H + A_H B_L = (A_L + A_H)(B_H + B_L) - A_L B_L - A_H B_H$$
Since multiplications $A_L B_L$ and $A_H B_H$ can be reused after being computed, one multiplication has been eliminated, but two additions and one subtraction have been added. When one addition and two subtractions execute more quickly than one multiplication due to the size of the operands, this is a better option compared to the naive approach. The Karatsuba algorithm is of the order O($n^{1.58}$) in terms of complexity, which is higher than NTT. However, less operations and hardware resources are used for Karatsuba when the degrees of the polynomials are lower. Such an approach, and past techniques used to accelerate this procedure, will also be further considered in the design of our accelerator.

```c
#define POLYNOMIAL_SIZE 4096

// typedef uint32_t coeff_prec;
// typedef uint64_t coeff_prec;
typedef unsigned __int128 coeff_prec;

void multiply(coeff_prec A[], coeff_prec B[], coeff_prec C[]) {
    for (int i = 0; i < POLYNOMIAL_SIZE; i++)
        for (int j = 0; j < POLYNOMIAL_SIZE; j++)
            C[i + j] += A[i] * B[j];
}
```

Listing 1: Naive C representation of Polynomial Multiplication

4 ESTIMATED RESULTS

This section presents an estimation of the potential performance uplift we can achieve when using an accelerator to accelerate the computationally bound polynomial multiplication with a wide data type of 128 bits. For the baseline, we implemented an algorithmic description of the polynomial multiplication shown in Listing 1.

We measured the number of instructions needed using the "perf_event_open" system call. We measured the number of instructions required on two systems using two different ISA, one x86 and the other ARM, and the results are summarized in Table 1. As can be seen, the computation for polynomial multiplication is in the order of hundreds of millions of instructions. Furthermore, as we increased the bit precision of the coefficients, we expected the number of instructions to increase. This fact is reflected in the table, although in the Raspberry PI system, uint128 was not supported.

The key kernel that we are trying to accelerate has a high degree of parallelism due to the polynomial multiplication not having any dependencies. Thus based on the current CPU implementation we have on hand, assuming infinite area and memory bandwidth, we should be able to compute the polynomial multiplication under 32 clock cycles, the maximum time needed for a pipeline phase of a RISC-V CPU. However, in reality, our design is limited by area and memory bandwidth; thus, the maximum speed-up we can achieve will depend on this constraint. Furthermore, the design space of the polynomial multiplication can be extended to the algorithmic level, where kernels like the NTT algorithm can be used to approximate the polynomial multiplication.

All things considered, we are aiming for a design that is able to accelerate the polynomial multiplication by 10X. To achieve this goal, we are currently exploring many different methods for computing polynomial multiplication, as introduced in the previous section. Furthermore, to address multiple limiting factors, such as the memory wall, we are exploring caching methods to hide the unavoidable memory latency. At the high level, we are in the process of optimizing the algorithm of the homomorphic multiplication, identifying sections where loop fusion can be employed to simplify the computation and also simplify the hardware design. The true limiting factor of our design would be the simulation capacity, and for the scope of this project, we aim to scale the design – decreasing the polynomial size and bit precision so that the design cycle is in the order of hours rather than days.

Coefficient Precision	x86 - AMD 1700X	ARM - Rasp 4
uint32	553676865	536920098
uint64	553676860	637583393
uint128	822112321	N/A

Table 1: Instructions Counts

5 CONCLUSION AND FUTURE WORK

Homomorphic encryption can revolutionize the security field if we can solve the computation problem of cyphertexts. Thus we propose a method for accelerating the key limiting factor of homomorphic encryption, the polynomial multiplication. We hope that by being able to efficiently manipulate ciphertexts, we can, in the future, explore the design space of the application of homomorphic encryption using our accelerator to train and use deep neural network entirely in the encrypted space.

REFERENCES

[1] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat practical fully homomorphic encryption. *Cryptology ePrint Archive* (2012).

[2] Jongmin Kim, Gwangho Lee, Sangpyo Kim, Gina Sohn, Minsoo Rhu, John Kim, and Jung Ho Ahn. 2022. Ark: Fully homomorphic encryption accelerator with runtime data generation and inter-operation key reuse. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1237–1254.