Accelerating BGV Scheme of Fully Homomorphic Encryption Using GPUs

by

Jiyang Dong

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Electrical & Computer Engineering

by

April 2016

APPROVED:

Professor Xinming Huang, Major Thesis Advisor

Professor Yehia Massoud, Head of Department

Abstract

After the first plausible fully homomorphic encryption (FHE) scheme designed by Gentry, interests of a building a practical scheme in FHE has kept increasing. This paper presents an engineering study of accelerating the FHE with BGV scheme and proves the feasibility of implement certain parts of HElib on GPU. The BGV scheme is a RLWE-based FHE scheme, which introduces a set of algorithms in polynomial arithmetic. The encryption scheme is implemented in finite field. Therefore, acceleration of the large polynomial arithmetic with efficient modular reduction is the most crucial part of our research efforts. Note that our implementation does not include the noise management yet. Hence all the work is still in the stage of somewhat homomorphic encryption, namely SWHE. Finally, our implementation of the encryption procedure, when comparing with HElib compiled by 9.3.0 version NTL library on Xeon CPU, has achieved 3.4x speedup on the platform with GTX 780ti GPU.

Acknowledgments

First I would like to express my gratitude to my professor, Xinming Huang, who guid me to finish this thesis topic. He provided me the right direction to dig in and also narrowed vision from the vast amount relative and previous work which provide me good inspiration. Professor Huang provided me not only invaluable suggestions but also well-performing devices and sufficient supplies which supported me to finish this thesis.

Also I am really grateful to professor Erkan Tüzel and his students Kingsley, James Leonard for sharing their computation platform with me when the original computation device was accidentally flushed, and helping me get started on their platform.

Thanks to Wei Wang who shared the algorithm of his previous work to reference and also show me good direction in much more details. Dai Wei, a member of Vernam Group of WPI, who is mainly working for the implementation of cuHE library of NTRU scheme, have given me invaluable recomendation and suggestions since the NTRU scheme is also a RLWE-based scheme. Also I would not be able to build appropriate background for FHE in a short time without his help.

I also need to thanks other students in the laboratory, during my project working they disscussed with me and gave me brilliant suggestions. I really glad to know them as friends in my life. Lastly I want to thank my paraents for the nature encouragement from them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Since Gentry constructed the first plausible scheme in 2009, fully homomorphic encryption (FHE) has been a prevalence in the area of cryptography. A consequent possible application of practical FHE scheme is cloud computing which allows a untrusted cloud to perform computations directly on ciphertexts for arbitrary times without "leaking" (decryption) the original data from clients. Later the customers can obtain their desired results by simply applying the decryption to the output sent back from the cloud. Therefore during the entire operations on the cloud server, no one other than the client is able to access the original plaintext, the user's privacy is ensured. Because of this invaluable feature, FHE has been increasingly attracting the attentions from both academia and industry.

1.1 Different FHE implementations

Most cryptography schemes is based on mathematical problems which is proved hard to solve through contemporary computational power. The basic idea to accomplish FHE scheme is to introduce noise purposely to the ciphertext, which correlates the security to certain existing mathematical problems, to make it difficult for adversary

to extract the valid data from noisy ciphertexts. The first FHE scheme introduced by Gentry is using ideal lattices wherein the security of the scheme is based on the hardness of lattice problem [1]. In 2010, Gentry [2] proved that the security of his FHE scheme can be reduced to the worst-case of hardness of lattice problems.

Similarly, Coron *et al.* proposed a FHE scheme over integers [3]; Brakerski and Vaikuntanathan presented another efficient FHE scheme which is based on learning with errors (LWE) problem [4]. At this moment, the LWE based schemes are considered the most efficient among all prevailing FHE schemes. One of them is designed by Brakerski, Gentry and Vaikuntanathan, which is widely known as BGV scheme [5]. Another is designed by Lopez, Tromer and Vaikuntanathan, known as LTV scheme [6]. The LTV scheme is also known as NTRU-based FHE scheme. Both BGV and LTV are LWE-based schemes, and one should be aware that they all work in polynomial rings, which requires vast and large-degree polynomial arithmetic.

1.2 Other important concepts in LWE-based schemes

1.2.1 Somewhat Homomorphic Encryption (SWHE)

As we mentioned earlier that almost all the FHE schemes is implemented by "introducing noise to the plaintext". One can recover the message by given the secret key. However, the ciphertexts after evaluation may have a correctness problem due to the growth of noise contained in the ciphertexts. Gentry invented a procedure called "bootstrapping" which is to "refresh" the ciphertext to ensure the correctness of decryption [1]. Since Gentry called the encryption scheme before bootstrapping "somewhat" homomorphic, this SWHE term remains in all the later works. The bootstrapping is viewed as a way of noise management and the later works also introduced other ways of noise management. Therefore, all the scheme of FHE

should contain a SWHE which supports only limited depth of evaluation circuit, plus proper noise management in the cyphertext.

1.2.2 Batching

Note that the LWE-based schemes are working on polynomial rings. The SWHE contained in such schemes uses polynomial rings with the form of

$$R_p = \mathbb{Z}[X]/(F(X), p) \tag{1.1}$$

as the plaintext space, where $F(X)$ is a cyclotomic polynomial (which can be seen as the "polynomial version of prime") ring and p is a prime integer. Thus, the plaintext, which is usually an one-bit number, is encrypted into ciphertext in a respective "field" defined by equation (1.1). Then Smart and Vecauteren introduced the concept of SIMD [7] and indicated that one can "split" the original plaintext space into a vector of smaller spaces with the help of Chinese Remainder Theorem (CRT) on polynomials. The original cyclotomic polynomial in the plaintext space can be written into:

$$\Phi_m(X) = F(X) = F_1(X) \cdot F_2(X) \cdots F_l(X) (mod\, p) \tag{1.2}$$

where $F_n(X)$ can be viewed as a plaintext space "slot". Consequently, one can "encode" a vector of plaintext bits, instead of a single bit, to respective slots described by equation (1.2). Such "encoding" procedure is called "batching". Thus one can batch a vector of plaintext bits and encrypt them into one ciphertext so that the evaluation is operated in parallel by applying component-wise operations to ciphertexts.

1.2.3 Ring Learning With Errors Problem (RLWE)

In 2010, Lyubaskevsky, Peikert and Regev [8] introduced the ring-LWE problem. In the BGV scheme, Brakerski, Gentry and Vaikuntanathan used a special case of the RLWE for FHE [5] which will be explained in the later section. For readers who first know the words of RLWE, the general idea is that random linear equations, when introduced and mixed with a small amount of noise sampled from a certain distribution, are not distinguishable from a real uniform random distribution. Consequently one can not find the true information from the noisy polynomials in a certain ring.

Chapter 2

Related Work

This work is focusing on accelerating and re-engineering the HElib which is a C++ implementation of BGV scheme. There are already several previous works which have achieved decent performance.

2.1 First attempt of accelerating FHE scheme

The first attempt to accelerate a FHE scheme is on both FPGA and GPU platform, which is implemented by Wang *et al.* following the scheme introduced by Gentry and Halevi [9]. This scheme is a lattice-based scheme with the following encryption procedure:

$$c = [u(r)]_d = [b + 2\sum_{i=1}^{n-1} u_i r^i] \tag{2.1}$$

where d and r are public key. We can see from the equation (2.1) that one need to evaluate the inner product of vector (with all 0/1 elements) u and r^i. After the encryption, one bit of plaintext will be encrypted to a large integer which could be as large as 768 000-bits when a dimension of 2048 lattice is applied [10]. Therefore the most crucial part that impacts the efficiency is modular multiplication among

large integers [11].

According to [11], to achieve acceleration of large integer multiplication, they employed the Strassen's multiplication algorithm introduced by Emmart and Weems [12] with the help of finite-field FFT:

Strassen multiplication for integers

1. Given 2 number A and B, view each digit of them as an sample in FFT algorithm
2. Apply FFT to both A B into FFT(A) and FFT(B) respectively
3. Multiply the result of FFT in component-wise: C[i] = FFT(A)[i] * FFT(B)[i]
4. Compute the inverse FFT to the product C abouve: C' = iFFT(C)
5. Resolve the carries(where b is the base):
   for each element in C' **do**
   if C'[i] >= b **do**
   C'[i+1] += C'[i] **div** b;
   C'[i] = C'[i] **mod** b;
   end if
   end for
6. **return** C'[i];

Figure 2.1: Strassen FFT multiplication for integers

Because the Step 3 in Figure 2.1 is a component-wise multiplication in the Strassen FFT multiplication algorithm, the advantage of parallelism computing of GPU can be employed to achieve much better performance.

Niall *et al.* [12] found a specific prime, p = 0xFFFFFFFF00000001 for the FFT procedure mentioned above in finite field. This prime actually belong to a set of primes called solinas primes which have similar properties. Now we focus on the prime 0xFFFFFFFF00000001 used by Niall and explain what special properties does this prime have. Below are the definition of k-point FFT and iFFT in finite

field:

$$X(i) = \sum_{j=0}^{k-1} x_j (r_k)^{ij} (mod\, p) \tag{2.2}$$

$$x(j) = -k^{-1} \sum_{j=0}^{k-1} X_i (r_k)^{-ij} (mod\, p) \tag{2.3}$$

where r_k is the k^{th} primitive root and p is a prime for the definition of finite field. When a solinas prime is introduced here, in this case 0xFFFFFFFF00000001, for 64-point or less FFTs, the roots of unity are powers of two, e.g. $r_{64} = 2^3$ (Since under the finite field p, $(2^3)^{64} mod\, p = 1$, which is exactly the definition of a root of unity). Thus, under the finite field for 0xFFFFFFFF00000001, all the multiplications in the FFT procedure can be accomplished by shifting bits (multiplication with power-of-two numbers) which is a much simpler operation than large number multiplications, especially for FPGA devices.

2.2 Acceleration of LTV FHE scheme

LTV scheme is also a LWE based FHE scheme [6], which is much similar to BGV scheme. Dai *et al.* has proved that the Strassen FFT multiplication algorithm is also feasible when the case comes to polynomials, in which the LWE (or RLWE) problem is employed. In this case all the operations are performed in a plaintext space defined by a polynomial ring:

$$R_p = \mathbb{Z}[X]/(F(X), p) \tag{2.4}$$

More specifically, the $F(X)$ is defined in the form of $x^n + 1$, where n is the "length" (degree) of a polynomial. Different from scheme based on lattice, the one-bit message is actually "embedded" into a polynomial of Definition (2.4); Then the

key way to achieve better performance is to accelerate the operations of polynomials with a high degree.

To accelerate high-degree polynomial multiplications, the Strassen multiplication need to be modified slightly as in Figure 2.2. (Note that due to the scenario of FHE, we often assume that two polynomial operands are in the same degree n)

Strassen multiplication for polynomials

1. Whenever there two polynomial operands A and B exist, expand them to $2n$-degree polynomials by padding 0s to the coefficients. View each coefficient as an sample of FFT.
2. Then simply aply FFT to both a and b: $A = FFT(A)$, $B = FFT(B)$;
3. Multiply the result of FFT in component-wise: $C[i] = FFT(A)[i] * FFT(B)[i]$
4. Compute the inverse FFT to the product C above: $C' = iFFT(C)$
5. Resolve the carries
6. return C'[i]

Figure 2.2: Strassen multiplication for polynomials

There is a slight difference between a large integer multiplication algorithm and a high-degree polynomial. The former viewed each bit of a large integer as a sample of FFT operation and the other viewed a coefficient of a polynomial as a sample.

More specifically, the FFT or Number Theoretic Transform (NTT) algorithm applied here is the Cooley-Tukey FFT algorithm. The NTT algorithm used in [13] is a radix 64 FFT as in Figure 2.3.

NTT (Cooley-Tukey) algorithm

1. Split N samples into a vector of 4096(64 * 64) -sample rows.(4096 rows, N/4096 columns)
2. **for** each set of 4096 samples **do**
3. Split 4096 samples into a vector of 64-sample rows(64 rows, 64 columns)
4. **for** each 64 samples **do**
5. 64-point NTT
6. **end for**
7. Transpose
8. Multiply twiddle factors(4096-point)
9. **for** each 64-sample rows(after transpose operation) **do**
10. 64-point NTT
11. **end for**
12. **end for**
13. Transpose
14. Multiply twiddle factors(N-point)
15. **for** 4096 columns **do**
16. N/4096-point NTT
17. **end for**

Figure 2.3: Algorithm of NTT(FFT) used in cuHE

Note that the 64-point NTT is built with 8-point FFTs in assembly code on GPU provide by Niall *et al.* such that the multiplications in the FFT are indeed replaced with only shifting and addition operations. In a addition, Dai *et al.* also built a complete scheme on GPU, cuHE [14], to utilize the computation power of GPU heavily. But generally the most crucial part of this scheme affected by serial

processing is still the multiplication of high-degree polynomial [13]. This NTT algorithm used in cuHE only supports FFTs in the length of 16384, 32768 or 65536.

Chapter 3

Accelerating The HElib

HElib [15] is an open source library project implemented in C++, which follows the BGV scheme. The HElib has implemented the numerical and mathematical theory employed in BGV and as well as the higher level applications. They have lower layers filled with math structures and classes which could be an analogy to a lower-level architecture as "assembly language", providing a "platform" for the crypto scheme [16]. Higher layers, called "crypto layers", provide the classes and methods required by FHE scheme, e.g. ***KeySwitching*** , ***FHEcontext***, ***EncryptedArray*** etc.

Similar to other hardware implementations, we focus on the FFT/iFFT procedure and relative index utilities and polynomial arithmetics too, all of which lied in lower-level layers.

3.1 Arbitrary-length Bluestein FFT

Each scheme and accelerating method mentioned above focus on implementation of FFT algorithm for large-size multiplications during the encryption and evaluation. Roughly, this is resulted from the Gentry's original design since all the encryption

schemes keep themselves secure by introducing noise to the plaintext which lead to the huge size of encrypted data.

Although the NTT and FFT are different structurally, in concept they are the same and they both employ Cooley-Tukey algorithm. My accelerating technique also focus on FFT, however, employed the Bluestein FFT procedure instead. Although it is easy to find that the original implementation in HElib also harness the Bluestein FFT algorithm, we actually choose it with decent comparison and concern since not all implementation on CPU are transportable to a GPU platform.

Bluestein FFT was presented by Bluestein in [17]. Now we can have a introduction to Bluestein's algorithm. Recall that

$$X(i) = \sum_{j=0}^{k-1} x_j (r_k)^{ij} (mod\, p) \tag{3.1}$$

Multiply $(r_k)^{-\frac{1}{2}i^2}$with both side of equation (3.1), it will become:

$$(r_k)^{-\frac{1}{2}i^2} X(i) = \sum_{j=0}^{k-1} x_j (r_k)^{\frac{1}{2}(2ij-i^2)} (mod\, p) \tag{3.2}$$

Since $2ij - i^2 = -(j-i)^2 + j^2$, the equation (3.2) would be respectively:

$$\begin{aligned} (r_k)^{-\frac{1}{2}i^2} X(i) &= \sum_{j=0}^{k-1} x_j (r_k)^{\frac{1}{2}(-(j-i)^2+j^2)} (mod\, p) && (3.3) \\ &= \sum_{j=0}^{k-1} x_j (r_k)^{\frac{1}{2}j^2} (r_k)^{-\frac{1}{2}(j-i)^2} (mod\, p) && (3.4) \end{aligned}$$

Therefore

$$X(i) = (r_k)^{\frac{1}{2}i^2} \sum_{j=0}^{k-1} x_j (r_k)^{\frac{1}{2}j^2} (r_k)^{-\frac{1}{2}(j-i)^2} (mod\, p) \tag{3.5}$$

Note that the right side of equation (3.4) is actually the convolution of $x(n){r_k}^{\frac{1}{2}n^2}$

and ${r_k}^{-\frac{1}{2}m^2}$, where $n = 0, 1, 2, ..., k-1$,$m = -k+1, -k+2, ..., -1, 0, 1, 2, ..., k-1$ for a k-point FFT. In another words, using Bluestein's algorithm, the FFT procedure is eventually turned to the convolution operation.

We can observe that, the second term of convolution, ${r_k}^{-\frac{1}{2}m^2}$ contains no parameters relative to the input x of the FFT. Thus we can simply put it into the "precomputation" and only execute it once. In this way, we can accelerate the the entire Bluestein FFT procedure. Here we also need to employ the computation power of GPU and it is a rule of thumb that when coding in CUDA [nvidia2015c], precomputation is a frequently-used step which could make the memory accessing much more efficient.

In particular, the convolution operation can be also accelerated by a pair of FFT and iFFT procedure and it is similar to the Strassen multiplication with the only slight difference that it does not require the step of resolving carriers. Here it may be confusing that the Bluestein's algorithm is optimizing FFT using another pair of FFTs! We shall analyze this step by step. First, Bluestein has proved that the FFT in his algorithm has the complexity of $N\,log\,N$ which is the same as the Cooley-Tucky algorithm. In addition, the procedure to accelerate the convolution operation is a well-developed process and has been widely applied to different fields. That is, roughly, we could actually "lower the hardness" of computation during the FFT procedure by putting steps into precomputation.

The Bluestein's FFT have certain benefits. CUDA is a platform for parallel computing invented by Nvidia [18]. However, that doesn't mean the operation could be pure in parallel when running on device. On the contrast one should always keep cautious to the memory management such as reducing the inter-transportation between Host and Device and avoid conflict of on device memory as much as possible to guarantee parallelism. There are good methodologies and suggestions from

the guide book from Nvidia [19]. For example, as we described above, we don't need to resolve the carriers by continuously executing $C'[i+1] += C'[i]\ div\ b$; $C'[i] = C'[i]\ mod\ b$, which is called data dependency. Such a dependency would extremely impact the parallelism. Similarly, the Transpose steps in the Cooley-Tucky NTT algorithm would also slow down the GPU execution or cause more restrictive memory management and complex implementation.

In conclusion, the Bluestein FFT is implemented in following steps (all in finite field)

NTT (Cooley-Tukey) algorithm

1. Apply FFT to sequence ${r_k}^{-\frac{1}{2}m^2}$(extended to 2k length by padding 0s) and get output: $R_b = \text{FFT}({r_k}^{-\frac{1}{2}m^2})$ (stage of precomputation)
2. Apply FFT to sequence $x(n){r_k}^{\frac{1}{2}n^2}$(extended to 2k length by padding 0s) and get output :$R_a = \text{FFT}(x(n){r_k}^{\frac{1}{2}n^2})$
3. Do element-wise multiplication between R_a and R_b :$R_c[i] = R_a[i] * R_b[i]$
4. Apply iFFT to R_c : $R'_c = \text{iFFT}(R_c)$
5. Element-wise multiplication with $(r_k)^{\frac{1}{2}i^2}$: $R'_c[i+k-1] * (r_k)^{\frac{1}{2}i^2}$

Figure 3.1: Bluestein FFT

When comparing to the Cooley-Tukey algorithm, it has the following benefits: no execution steps in which elements have dependencies on each other, the data path is cleaner (excluding transposing or carrier resolving), easy for memory management and implementation, and benefiting a lot from precomputation. Thus the Bluestein FFT algorithm is naturally GPU-friendly because of such properties. In addition, there is also a bonus that when harnessing the Bluestein FFT procedure, we can have k in arbitrary length, whereas the NTT algorithm supports only 16384, 32768 or 65536 of length k.

3.2 Double-CRT and Barrett modular reduction

When the plaintext space is factored into lower-degree fields, the BGV introduced the CRT. To achieve modulus switching, BGV scheme employ CRT second time on ciphertext space and build a "modulus chain" containing a vector of modulus to represent the sub-field generated by CRT. HElib also constructed a class, ***DoubleCRT*** to keep all unities for modulus switching.

As discussed above, the BGV scheme take good advantage of the FFT procedure to accelerate the polynomial operation. HElib makes every polynomial ready-to-evaluate before anything is evaluated. Therefore, HElib puts every polynomials into a "double-CRT representation" by applying finite field FFTs on each finite field represented by respective moduli in modulus chain (all operations are under $(mod\, q_i)$).

Therefore, a efficient modular reduction, especially for modular multiplication is also required for the density of modular reduction operations. Wang *et al.* have presented that they harnessed the Barrett Modular reduction [20] and we dicide to follow this modular algorithm with introducing the power of solinas prime.

In the previous section, we present how one can take advantage of prime P, 0xFFFFFFFF00000001. In fact, more generally, this prime can also speed up the fundamental multiplication. Here the prime is large enough hence the operands are in $\mathbb{Z}/P\mathbb{Z}$. Then the product can be written in the form of

$$2^{96}a + 2^{64}b + 2^{32}c + d \tag{3.6}$$

where a, b, c and d are all 32-bit values. Note that for prime 0xFFFFFFFF00000001 $2^{96} = -1\ (mod\, p)$, $2^{64} = 2^{32} - 1(\ mod\, p)$, equation (3.6) can be written as

$$
\begin{aligned}
product &= -1(a) + (2^{32} - 1)b + 2^{32}c + d \\
&= 2^{32}(b + c) - b - a + d
\end{aligned} \tag{3.7}
$$

We can obtain a fast fundamental multiplication by shifting 32 bits (multiplication with 2^{32}) and 32-bit subtractions and additions. Replacing the generic multiplication using the notation ***ModPMul*** and similarly, we also replacing addition and subtractions by ***ModPAdd*** and ***ModPSub***. Note that ***ModPAdd***, ***ModPMul*** and ***ModPSub*** are implemented by assembly code on GPU to explore the power of parallel computation as much as possible.

Barrett modular multiplication

1. $n = \lceil log_2(q) \rceil, invQ = \lfloor \frac{2^{2n}}{q} \rfloor$ (Precomputation)
2. $t = ModPMul(a,\ b)$
3. $h = t/2^n$
4. $y = ModPMul(h,\ invQ)/2^n$
5. $z = ModPMul(y,\ q)/2^n$
6. $r = t - z$
7. **While** (r $\geq$ q) **do**
8. $\quad r = ModPSub(r,\ q)$
9. **return** r

Figure 3.2: Algorithm of Barrett multiplication

The Step 7 and 8 in Barrett multiplication algorithm is a loop and Wang presented that in their scenario that while loop would not last long. However, in our case, when applying some of the test cases for HElib data stream, certain inputs do

cause very long execution time and we decided to replace the steps seven and eight by a generic modular operation symbol, *"%"*.

3.3 Mapping to $\mathbb{Z}/\mathbf{m}\mathbb{Z}^*$

After finite field Bluestein FFT procedure finished with help of Barrett reductions, one still need to discard some of the outputs [15]and permutate to maintain a consecutive vector of the rest of the outputs, which is directed by corresponding indexing. This step is to make sure that when all the polynomial operands transformed in to FFT domain, the elements should be all in $\mathbb{Z}/m\mathbb{Z}^*$.

Mapping to $\mathbb{Z}/m\mathbb{Z}^*$ in HElib

1. **for** each element in length m **do**
2. **if** the ith element is in $\mathbb{Z}/m\mathbb{Z}^*$
3. put it into output vector
4. **end if**
5. **end for**

Figure 3.3: Mapping to $\mathbb{Z}/m\mathbb{Z}^*$ in HElib

Obviously this algorithm could be a simple code segment in CPU, however, it will impact the entire GPU implementation since the traverse-and-recombine procedure will cause dependency among the treads on CUDA device. At first glance, this procedure is not able to be avoided efficiently. We setup an auxiliary **dropFlags** vector with the same length of input, which indicate that the elements should be dropped or not, by the flag variables 1/0 respectively. With the help of GPU, therefore, we introduce an offset function to help with this mapping procedure.

Preparing step in CPU

1. In CPU, setup a vector which indicates that whether a respective element should be dropped, $dropFlags[]$.
2. Transporting the vector to GPU

Mapping to $\mathbb{Z}/m\mathbb{Z}^*$ in GPU

1. Setup register variable $offset$ for each thread;
2. **for** each, i^{th} , element(thread) in the vector **do**
3. **for** all $offset$ on thread index$\geq i$ **do**
4. **if** index $== i$ **do**
5. $offset = 0$
6. **else do**
7. $offset = offset + 1$
8. **end if**
9. **end for**
10. **end for**
11. Permutation: put each i^{th} FFT output into the$(i - offset)^{th}$ result vector.

Figure 3.4: Mapping to $\mathbb{Z}/m\mathbb{Z}^*$on GPU

However, in practice, we observe that the indexes on which the elements should be dropped actually have a certain periodical pattern. With this observation, we decide to make this step further optimized. When knowing the period, the mapping procedure is relatively simple to implement and less complex.

Preparing step in CPU

1. In CPU, determine the period and use it as a parameter,P, send to GPU

Mapping to $\mathbb{Z}/m\mathbb{Z}^*$in GPU

1. Use modular P to deter
2. Setup register variable *offset* for each thread;
3. **for** each, i^{th} , element(thread) in the vector **do**
4. Modular P to determine the wether the i^{th} elemetn should be dropped
5. **if** dropped do
6. $offset = 0$
7. **else do**
8. $offset = i/period + 1$
9. **end if**
10. **end for**
11. Permutation: put each i^{th} FFT output into the$(i - offset)^{th}$ result vector.

Figure 3.5: Mapping to $\mathbb{Z}/m\mathbb{Z}^*$on GPU

Once the $\mathbb{Z}/m\mathbb{Z}^*$ is settled, the vector $dropFlags$ and the period would not change so the preparing steps above can be put into the precomputation step. Although, according to our observation, the index of dropped element $dropFlags$ always exist in a specific period, we are not able to prove it mathematically, hence we decide retain the former one above as a backup option for readers whereas the latter one is used for performance enhancement.

3.4 Streaming operation

To harness the computational power of GPU, roughly, one need to first define several kernels with certain functions or implementations, then send data from CPU (host) to GPU (device) and let those kernels process the data according to their own definition. The GPU would run the kernels as much as possible to keep the parallelism and high efficiency. Hence once the algorithm is modified with enough parallelism and have the memory properly managed, one can get promising performance.

However, the CUDA do provide us a stream option which is used to handle the potential overlap part of different kernels and memory-copy operations. As mentioned in section 3.2, we also keep the double-CRT representation structure in GPU, which means each polynomial need to be FFT transformed to several finite fields. To lower the dependency and harness the parallelism, each polynomial in specific finite field locates in separate parts memory and is copied to respective parts of memory in GPU. This memory-copy operation here can actually overlap with subsequent kernels.

This optimization needs to be cautious considering the effect on the default stream [19]. When a certain procedure is running on GPU, if no stream is specified, all the executions happen in the default stream, which have the highest priority among all the streams. Therefore, when specified streams occurs, one needs to rearrange the kernels and host codes to avoid the streamed kernels to be interrupted by the default stream. After CUDA 7.0, the default stream can also take part in the concurrency with other streams with the flag "**–default-stream per-thread**" specified during the compilation of CUDA codes. Figure (3.6) shows the timing of the kernels running on GPU without the overlapping stream. Comparing with Figure 3.6, Figure 3.7 shows the streamed ***Memcpy*** operation. It is obvious in

Figure 3.7 that the most expensive steps on GPU have a large overlapping time period with other kernels, which is much more efficient.

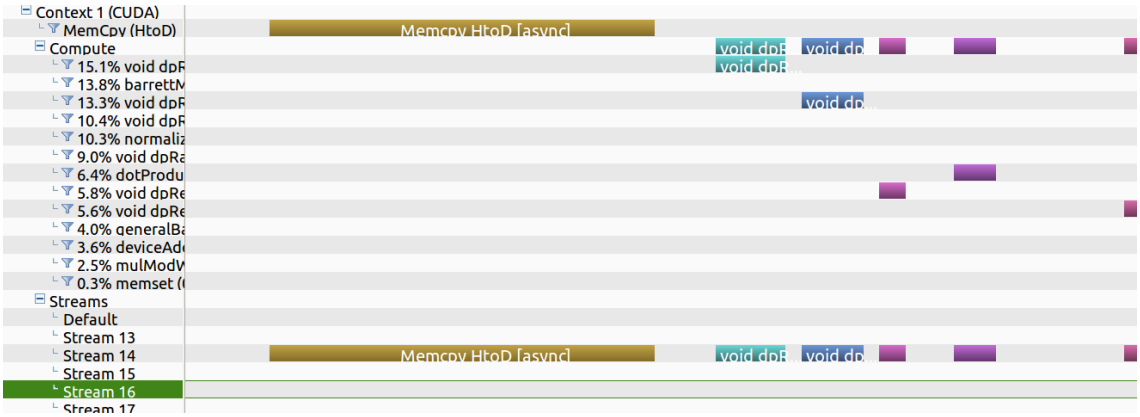

Figure 3.6: Kernels without stream overlapping

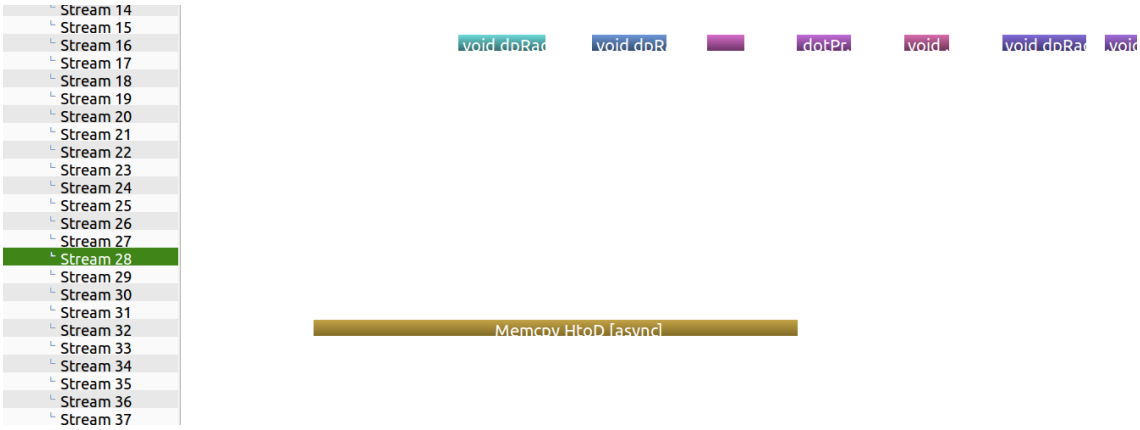

Figure 3.7: The streamed overlapping kernels

3.5 cuFFT library

In Section 3.1, we present that the Bluestein FFT can actually employ the FFT algorithm to accelerate the convolution step in the procedure. Therefore the NTT kernel designed by Dai for NTRU based FHE is also a good practice here for accelerating the Bluestein FFT algorithm. Observe that CUDA provides its own FFT for

complex numbers. We also tried this built-in function and discovered that cuFFT have an extremely powerful performance compared with the NTT algorithm. Even the NTT algorithm is already promising enough, the efficiency of NTT is still based on the large data amount of FHE, which is the parallelism provided by GPU when multiple kernels are lunched in the GPU device. The performance of a single NTT kernel doesn't have any advantage compared with the one provided by NTL library [21] in CPU. According to our observation on our platform, the cuFFT library, can actually provide more than ten times the performance of a single NTT kernel with the same volume of input data applied.

However, this built-in function of cuFFT has a precision issue. We have observed that the results output from cuFFT always have, in certain range, difference with the NTT ones. Recall that, the basic idea of the FHE scheme is to introduce noise mask to the original plaintext to maintain the security and homomorphism. Without using cuFFT, the noise still grow with evaluations. It is natural to have a conjecture that this precision issue can be viewed as another noise growth out of the steps of evaluation.

Ducas and Micciancio have focused on the accelerating the bootstrapping in HElib. In their implementation, they used cuFFT library too and claimed that this precision issue "doesn't necessary break the correctness of the scheme" [22]. They presented that the error growth during the FFT procedure is $O(\sqrt{logN})$ and their implementation worked correctly when they were performing the ***Homomorphic NAND*** and ***Refresh*** operations. Considering the huge advantage of the performance, we decide to employ the cuFFT instead of NTT. However, we are not able to theoretically prove the correctness during the fully homomorphic evaluation, so we decide to leave this result at SWHE stage.

We have a noteworthy observation in details for HElib: HElib supports the

option that using only half number of CRT primes for ciphertext space, which means there would be $L/2+1$ small primes for modulus switching, where L is the level of evaluation defined by user. This option is set as default when compiling the HElib. However, when the using the default setting, it leads to a problem that some of the primes have a size more than 32-bits. Another option is that, if a **-DNO_HALF_SIZE_PRIME** compiling flag is defined, there would be L different small primes, which are all less than 32-bits, for the modulus switching step.

Note that even the cuFFT does provide the 64-bit double precision, the subsequent complex operations do not support the numbers that are larger than 32 bits. Therefore, for simplicity of the implementation, we strongly recommend that, when compiling the HElib, the **-DNO_HALF_SIZE_PRIME** flag should be claimed.

3.6 Summary

Now that we have discussed about all the details included in the HElib encryption acceleration. In this section, the conceptual details of encryption and the complete system diagram of our GPU implementation will be presented. To encrypt a message bit $m \in R_2$, we need to extend the one-bit m to a vector $\boldsymbol{m}= (m, 0, ..., 0) \in R_q^{n+1}$ and the output ciphertext is:

$$c = m + 2 \cdot e + A^T \cdot r \in R_q^{n+1} \tag{3.8}$$

where $\mathbf{r} \leftarrow \chi$ is sampled from a noise distribution which have coefficients ± 1, 0 with probability 1/4 and 1/2 respectively. A is public key combined from vector $\boldsymbol{b} = \mathbf{Bt} + \mathbf{2e}$ followed by n columns of -$\boldsymbol{B}$. $\boldsymbol{B}$ is a matrix sampled uniformly, $\boldsymbol{B} \leftarrow R_q^{N \times n}$. The $\boldsymbol{t}$ is sampled by $\boldsymbol{t} \leftarrow \chi^n$ which is to generate the secret key, $sk = \boldsymbol{s} \leftarrow (1, \mathbf{t}[1], ..., \mathbf{t}[n]) \in R_q^{n+1}$.

Here we shall describe the relationship between the BGV encryption to RLWE. First, the scenario above is based on GLWE and one can accomplish a scheme based on RLWE by simply setting $N = 1$. When $\boldsymbol{s}$ is drawn uniformly and $e_i \leftarrow \chi$, RLWE problem is that samples (a_i, b_i) uniformly from R_q^2 and another bunch samples (a_i, b_i) where $b_i = a_i \cdot s + e_i$ are computational indistinguishable. Thus in this case, an attacker can not distinguish the public key, A^T, from a uniform in R_q^2 and also not able to recover the message m containing in the ciphertext.

However, using the secret key, "Bob" the receiver of the ciphertext, can recover this bit of message correctly. First, note that the crucial observation is $A \cdot s = 2e$. Consequently, the decryption is to compute the innerproduct of the ciphertext $\boldsymbol{c}$ and the secret key ***s followed by*** modular reduction by modulus q and *2* serially:

$$[[< c, s >]_q]_2 = [[(m^T + r^T A) \cdot s]_q]_2 = [[m + 2r^T e]_q]_2 = [m + 2r^T e]_2 = m \qquad (3.9)$$

Since in equation (3.9), the second term is much smaller than the modulus q, the whole scheme remains correct. If that condition is not satisfied, the decryption will be ruined. In particular, this term, $\mathbf{2r^Te}$, is actually relative to the error, which is the reason that why one need to keep the homomorphism by error management. Without error management, one will loose the correctness quickly during the evaluation and that is why this encryption is called "somewhat homomorphic encryption"(SWHE). Note that the number 2 in the above equation is the plaintext space. If we replace the number '*2*' , in $\boldsymbol{b = Bt + 2e}$ and equation (3.9), by a prime p, the scheme can supports other native plaintext space p instead of *2* .

Since the BGV scheme provides a SIMD option, after the batching step, the encryption procedure is slightly different.

First set up an "empty" ciphertext, $(c_0^{'}, c_1^{'})$, which means the "encryption of zeros". Define Q as the product of all the primes in the modulus chain for ciphertext space. A random low-norm polynomial $r \in R_Q$ with ± 1 and 0 coefficients. Then sample another low-norm error polynomials $(e_0, e_1) \in R_Q$ which follows Gaussian distribution with variance σ^2. The computation for canonical ciphertext is as follow

$$\overrightarrow{c} = (c_0, c_1) := r \cdot (c_0^{'}, c_1^{'}) + p \cdot (e0, e1) + plaintext \tag{3.10}$$

.

The implementation of the encryption in equation (3.10) can is shown in Figure (3.8).

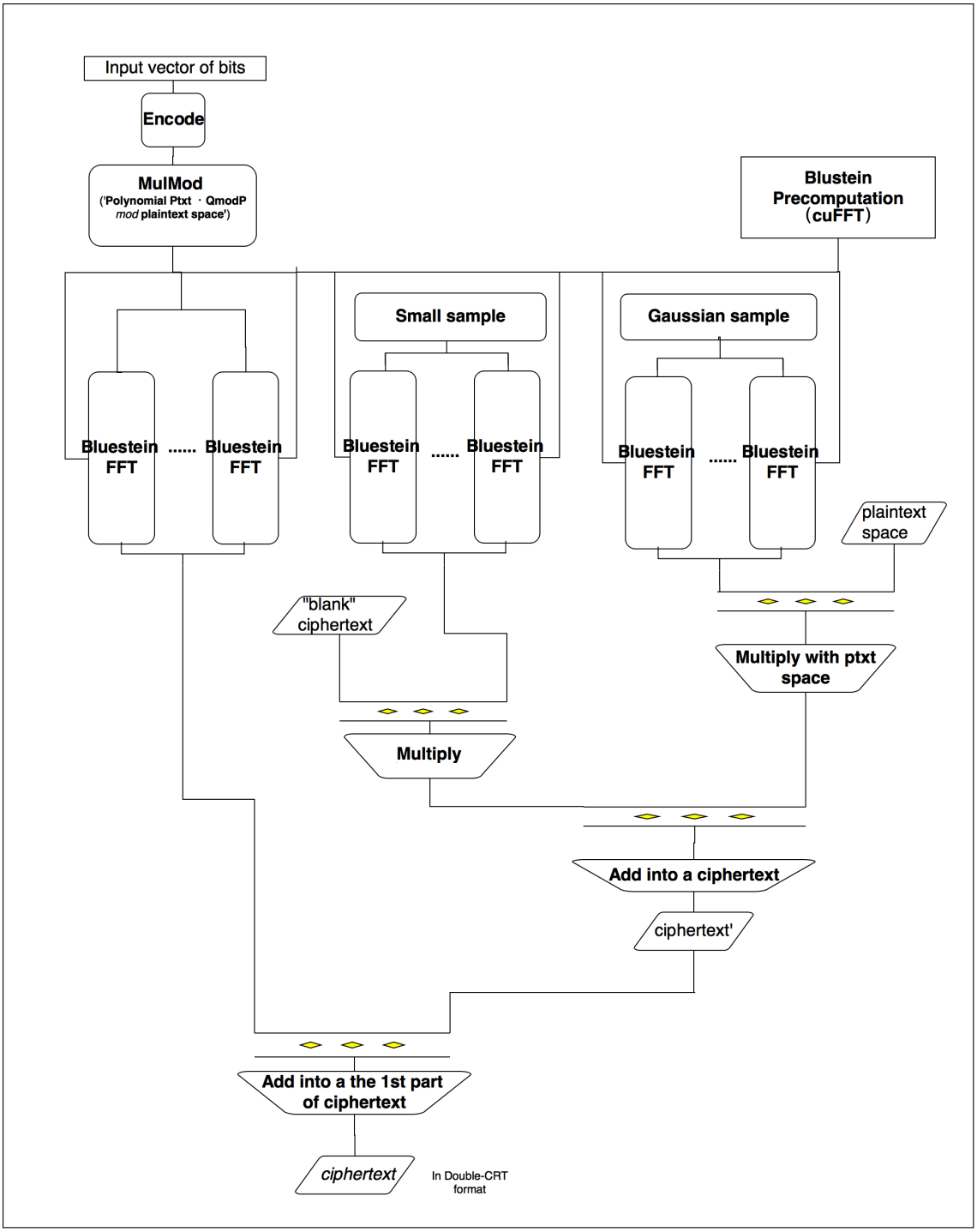

Figure 3.8: The encryption implementation on GPU

Chapter 4

Experimental Results

In Chapter 2, we have introduced some variants of GPU implementations for FHE schemes. However, different acceleration schemes are based on various assumptions, parameter settings, notations and even different scenarios. For example, Wang *et al.* implemented their integer-base scheme for one-bit encryption, and compared its performance with the original C/C++ code from IBM Research. Leo *et al.* decided to implement the bootstrapping procedure due to the lack of efficiency of certain bootstrapping part on HElib. However, they only focus on the scale of one-bit but the HElib supports SIMD (batching) techniques. Dai *et al.* have developed their own entire NTRU library on GPU.

Our acceleration performance on GPU is compared with the corresponding functions on HElib and the defined parameters are exactly the same as HElib, which could help readers to verify the feasibility of GPU acceleration for BGV scheme.

HElib does support other plaintext field with p larger than *2* so we choose $p = 5$ and $r = 1$ as the native plaintext space. Then we setup the parameter, Hamming weight of secret key, to 64. In this section, all the results below are generated on the platform with Intel Xeon e5620 CPU and GTX 780Ti with CUDA 7.5 installed.

The NTL included by HElib is on version 9.3.0.

4.1 Comparison of cuFFT/NTT procedure

In Section 3.5, we insist that it is worthy to harness the cuFFT library provided by Nvidia, considering its astonishing performance. Here we setup the security parameter, k, in HElib for 128, 256 and 512 respectively (which means we have $k - bit$ security). Then, we extract the performance of certain Bluestein FFT algorithm implemented by different version of FFT implementations and have them shown below.

Security	cuFFT(sec)	NTT(on GPU) (sec)	Speedup
128	0.001	0.065989	65.98
256	0.002	0.06499	32.49
512	0.003	0.146978	48.99

Table 4.1: Bluestein FFT performance comparison (with both cuFFT and NTT on GPU)

Security	cuFFT(sec)	FFT using NTL(on CPU) (sec)	Speedup
128	0.001	0.005999	6
256	0.002	0.005999	3
512	0.003	0.010998	3.67

Table 4.2: Bluestein FFT performance comparison (with cuFFT on GPU and FFT using NTL library on CPU)

As shown in Table 4.1, when comparing with NTT on GPU, the cuFFT has promising speedup when it comes to the single kernel. However, if the precision must be preserved, the NTT algorithm can ensure exactly the same behaviors as the ones in HElib. Note that when multiple NTT algorithm kernels are launched, the performance can slow down according to the algorithm in the previous work [14].

4.2 Comparison of encryption procedure

In Chapter 1, we introduce the batching technique, which allow users to encrypt multiple bits of plaintext in a SIMD pattern which is considered the parallel method. This technique requires the definition of both the plaintext space and "sub-space" generated by polynomial CRT. More specifically, HElib has a function, ***encode***, to batch multiple bits of plaintext. In our implementation, we leave this part remaining on CPU and our CUDA implementation on GPU takes a polynomial as the input.

There is no common measurement standard for the performance. Ducas et al and Sung Lee *et al.* implemented the bootstrapping which takes one-bit data as input even their work is relative to HElib or BGV which supports SIMD technique; According to Dai's work, their cuHE GPU library is with SIMD included but also taking a polynomial as their input. Therefore, our performance comparison includes two version. One is performing with the ***encode*** function included and the other one is without the ***encode*** function.

Security	NTL(9.3.0)(sec)	GPU (sec)	Speedup
128	0.192971	0.077989	2.47
256	0.205969	0.090986	2.26
512	0.451931	0.160976	2.8

Table 4.3: Speedup of encryption **including** *encode*

Security	NTL(9.3.0)(sec)	GPU(sec)	Speedup
128	0.166975	0.051991	3.2
256	0.172973	0.058992	2.9
512	0.389941	0.096985	4.0

Table 4.4: Speedup of encryption **excluding** *encode*

To make this comparison more straightforward, when the security parameter in HElib is set to 128, 256 and 512, the length of FFTs employed by the Bluestein algorithm during the encryption steps are 32768, 32768 and 65536 respectively.

Comparing with HElib, our encryption with *encode* function, that takes a vector of bits as input, is 2.5 times faster. The other one without the *encode* function that takes a polynomial as the input is 3.36 times faster. Our implementations have resulted a good performance and they still have more potential. Since our goal is to re-engineering the HElib, which requires interfaces between HElib and our implementation. According to Figure 4.1, the kernels are in parallel execution pattern and the execution time (including all the memory-copy functions which is known as the most expensive operation when switching to GPU platform) is relatively small. Thus, we have to tolerant a large part of "overhead" caused by the interface.

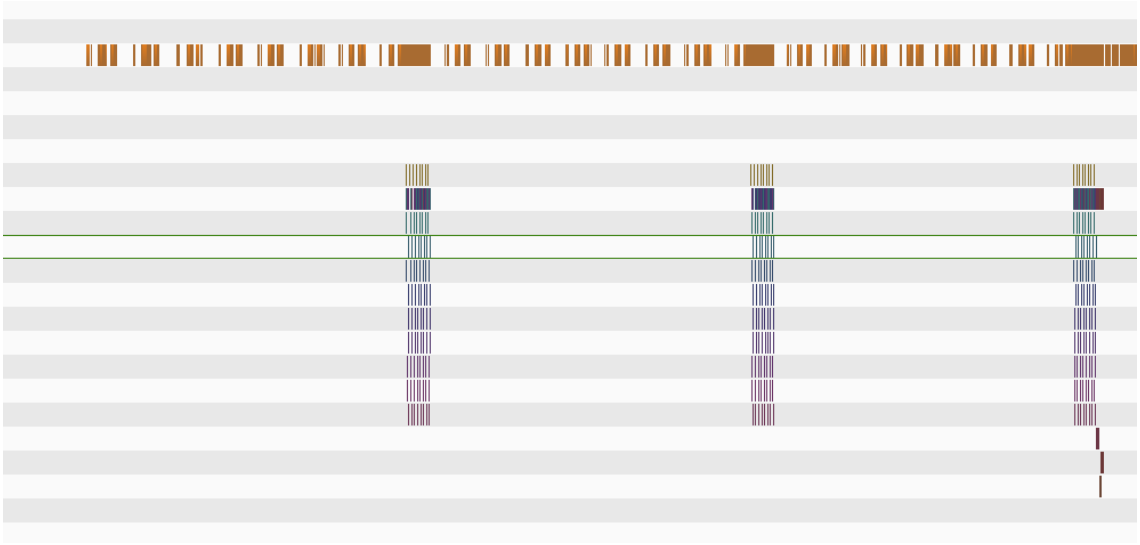

Figure 4.1: Results from Nvidia visual profiler

Chapter 5

Future Work

5.1 The precision issue of cuFFT

As we presented in previous chapters, cuFFT have a huge advantage on performance through the trade-off of precision. Somewhat unfortunately, we think that the precision issue cannot be simply ignored because most of the FHE schemes are based on the idea of introducing noise "mask" to the plaintext and they evaluate noisy ciphertext with correctness. However, at the time of writing this thesis, there is no elaborative discussion and analysis for the issue that how exactly harmful this precision issue would be when introducing cuFFT to the scenario of BGV or other FHE schemes.

The part which most likely would be affected is the level of evaluation. According to BGV scheme, the decryption would still valid as long as the magnitude of noise is smaller that $q/2$, where the q is the odd prime number used to define the polynomial fields. Therefore if the "base" of the noise has increased during the encryption, then the later evaluation, which grow the noise each time, may reach the noise limit.

5.2 Fully homomorphic encryption

Our implementation work is actually at the stage of SWHE. Other stages of fully homomorphic encryption are about the noise management features, e.g. bootstrapping and modulus switching. Only with these techniques applied, the scheme is fully homomorphic since the noise management would allow the user to evaluate the ciphertext in arbitrary depth.

5.2.1 Modulus switching

Our implementation have all the modulus needed during the modulus switching. When there is a need to apply the modulus switching, the following procedure should be applied.

1. First, define a parameter $\Delta = q/q'$ where q is the current modulus and q' is another smaller modulus.
2. When applying to a ciphertext, do $\overrightarrow{\gamma} = \overrightarrow{c}\ mod\Delta$
3. Ensure the coefficients in$\overrightarrow{\gamma}$ is divisible by p where p is the plaintext space modulus by adding or subtracting multiple Δ.
4. Creat another ciphertext $\overrightarrow{c}'$ and $\overrightarrow{c}' = \overrightarrow{c} - \overrightarrow{\gamma}$
5. Then output $\overrightarrow{c}' = \overrightarrow{c}'/\Delta$

Figure 5.1: Modulus switching

5.2.2 Bootstrapping.

Although the BGV scheme is designed for FHE without bootstrapping, it introduces bootstrapping as a form of optimization for their scheme. The bootstrapping procedure has several advantages [5]:

1. The BGV scheme has to specify the levels of evaluations in advance. This would cause a problem for implementation of hardware acceleration when more eval-

uations, which exceed the level limit, need to be performed. In this scenario, more primes need to be added for further evaluation. Such a scenario would negatively affect the parallelism because when a prime is added outside the original defined levels, most likely the hardware platform need to be designed again. Thus, usually hardware implementations do not have good flexibility for this situation if there is no bootstrapping feature in such implementations.

2. Bootstrapping make the ciphertexts shorter, and [22][23] have shown the feasibility of accelerating the bootstrapping. Ducas *et al.* claimed that their bootstrapping can be finished in less than a second. Therefore it is possible to design a more flexible scheme with bootstrapping on hardware platform.

5.3 Extend the implementation to the complete BGV scheme

From our experimental results, the major limitation of our implementation is that when implementing certain parts of functions, the performance suffers a lot from the GPU/CPU interface. Another LWE-based and NTRU based FHE scheme which has a lot of similarities with BGV scheme has been implemented by Dai *et al.* as cuHE which is open source on Github. So it is possible to implement the complete HElib on GPU including the modulus switching and re-linearization on GPU. We suggest that the HElib would achieve better performance when more modules, e.g. SWHE including batching and noise management procedures, are implemented on GPU.

Bibliography

[1] C. Gentry *et al.*, "Fully homomorphic encryption using ideal lattices." in *STOC*, vol. 9, 2009, pp. 169–178.

[2] M. Van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, "Fully homomorphic encryption over the integers," in *Advances in cryptology–EUROCRYPT 2010*. Springer, 2010, pp. 24–43.

[3] J.-S. Coron, A. Mandal, D. Naccache, and M. Tibouchi, "Fully homomorphic encryption over the integers with shorter public keys," in *Advances in Cryptology–CRYPTO 2011*. Springer, 2011, pp. 487–504.

[4] Z. Brakerski and V. Vaikuntanathan, "Efficient fully homomorphic encryption from (standard) lwe," *SIAM Journal on Computing*, vol. 43, no. 2, pp. 831–871, 2014.

[5] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," in *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*. ACM, 2012, pp. 309–325.

[6] A. López-Alt, E. Tromer, and V. Vaikuntanathan, "On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption," in *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*. ACM, 2012, pp. 1219–1234.

[7] N. P. Smart and F. Vercauteren, "Fully homomorphic simd operations," *Designs, codes and cryptography*, vol. 71, no. 1, pp. 57–81, 2014.

[8] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," *Journal of the ACM (JACM)*, vol. 60, no. 6, p. 43, 2013.

[9] C. Gentry and S. Halevi, "Implementing gentry's fully-homomorphic encryption scheme," in *Advances in Cryptology–EUROCRYPT 2011*. Springer, 2011, pp. 129–148.

[10] W. Wang, X. Huang, N. Emmart, and C. Weems, "Vlsi design of a large-number multiplier for fully homomorphic encryption," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 22, no. 9, pp. 1879–1887, 2014.

[11] W. Wang, Y. Hu, L. Chen, X. Huang, and B. Sunar, "Accelerating fully homomorphic encryption using gpu," in *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*. IEEE, 2012, pp. 1–5.

[12] N. Emmart and C. Weems, "High precision integer multiplication with a gpu," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*. IEEE, 2011, pp. 1781–1787.

[13] W. Dai, Y. Doroz, and B. Sunar, "Accelerating ntru based homomorphic encryption using gpus," in *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*. IEEE, 2014, pp. 1–6.

[14] W. Dai and B. Sunar, "cuhe: A homomorphic encryption accelerator library," in *Cryptography and Information Security in the Balkans*. Springer, 2015, pp. 169–186.

[15] S. Halevi and V. Shoup, "Helib, homomorphic encryption library," *Internet Source*, 2012.

[16] ——, "Algorithms in helib," in *Advances in Cryptology–CRYPTO 2014*. Springer, 2014, pp. 554–571.

[17] L. I. Bluestein, "A linear filtering approach to the computation of discrete fourier transform," *Audio and Electroacoustics, IEEE Transactions on*, vol. 18, no. 4, pp. 451–455, 1970.

[18] C. Nvidia, "Compute unified device architecture programming guide," 2007.

[19] C. NVIDIA, "C programming guide, version 7.5," 2015.

[20] P. Barrett, "Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor." Springer, 1986.

[21] V. Shoup *et al.*, "Ntl: A library for doing number theory," 2001.

[22] L. Ducas and D. Micciancio, "Fhew: Bootstrapping homomorphic encryption in less than a second," in *Advances in Cryptology–EUROCRYPT 2015*. Springer, 2015, pp. 617–640.

[23] M. S. Lee, Y. Lee, J. H. Cheon, and Y. Paek, "Accelerating bootstrapping in fhew using gpus," in *Application-specific Systems, Architectures and Processors (ASAP), 2015 IEEE 26th International Conference on*. IEEE, 2015, pp. 128–135.