Accelerating Leveled Fully Homomorphic Encryption Using GPU

Wei Wang, Zhilu Chen, and Xinming Huang
Department of Electrial and Computer Engineering
Worcester Polytechnic Institute, MA 01609, USA
{weiwang, zchen2, xhuang}@wpi.edu

***Abstract*—Gentry introduced the first plausible fully homomorphic encryption (FHE) scheme, which was considered a major breakthrough in cryptography. Several FHE schemes have been proposed to make FHE more efficient for practical applications since then. The leveled fully homomorphic scheme is among the most well-known schemes. In leveled FHE scheme, large-number matrix-vector multiplication is a crucial part of the encryption algorithm. In this paper, Chinese Remainder Theorem (CRT) is employed to reduce the computational complexity of the large-number element-by-element modular multiplication. The first step is called decomposition, in which each large-number element in the matrix and vector is decomposed into many small words. The next step is vector operation that performs the modular multiplications and additions of the decomposed small words. Finally the matrix-vector multiplication results can be obtained through reconstruction. We compare the CRT-based method with Number Theory Library (NTL), showing the proposed method is about 7.8 times faster when executing on CPU. In addition, it is observed that vector operation takes up to 99.6% of the total computation time and the reconstruction only takes 0.4%. Therefore GPU acceleration is employed to speed up the vector operations. Experiment results show that the GPU implementation of the CRT-based method is 35.2 times faster than the same method implemented on CPU and is 273.6 times faster than the NTL library on CPU.**

***Index Terms*—fully homomorphic encryption, Chinese remainder theorem, matrix-vector multiplication, CUDA**

I. Introduction

Gentry proposed the first plausible fully homomorphic encryption scheme, being one of the most significant advances in cryptography [1] in recent years. Fully homomorphic encryption (FHE) allows a computationally powerful server perform an arbitrary number of computations directly without revealing its secret key. Therefore, a practical FHE will provide an invaluable security application for emerging technologies such as cloud computing and cloud-based storage.

However, FHE is hard to have a practical application in real life due to its serious efficiency impediments. Several different FHE schemes has been proposed to make FHE more efficient [2][3][4][5]. The first implementation of a lattice-based FHE variant was reported by Gentry and Halevi [6]. Although it employs many impressive optimization methods to reduce the size of public key and improve the efficiency of primitives, the public key size is still very large about 17 Mega Bytes, encryption of one bit takes more than one second and recrypt primitive takes nearly half a minutes on a high-end Intel Xeon based server in the small setting case. We took an initial step to accelerate Gentry-Halevi scheme using the integer-FFT multiplication algorithm on NVIDIA C2050 GPU [7], which resulted about 7 times speedup compared the original work in [6]. Later, we managed to keep the calculation in FFT domain as reported in [8] and have about 342 times speedup for encryption, 15 times speedup for recryption and 7 times speedup for decryption. In a recent development, a more efficient FHE scheme called leveled fully homomorphic encryption without bootstrapping is reported in [2].

In this paper, we propose to accelerate the leveled FHE variant using NVIDIA GPU. In the leveled FHE scheme, the crucial operation for encryption is a large-number matrix-vector multiplication. The Chinese Remainder Theory (CRT) is applied to reduce the complexity of large-number modular multiplications. It includes three main steps. During the first step, CRT is used to decompose each large-number element into many small words, which is called the decompose process. The decompose process can be precomputed in the CPU. The second process is vector operation that performs modular multiplications and additions of all these small words. Finally, the final results can be reconstructed in the reconstruction process. In our observation, the vector operation process takes most of the computation time in CPU. So we implement this part in GPU while other computations remain in the CPU. A Compute Unified Device Architecture (CUDA) program [9] is developed to accelerate the computations by running it in many threads in parallel on GPU with many cores available.

The rest of the paper is organized as follows. In Section 2, we briefly review the leveled FHE scheme. The CRT-based method is described in Section 3. In Section 4, we present the method for GPU implementation. Section 5 gives the evaluation and experimental results.

II. Basic Leveled FHE Encryption Scheme

A fully homomorphic encryption scheme supports arbitrary binary operations on the ciphertexts similar to the operations on the original plaintexts without the knowledge of the encryption key: $E(x_1) \otimes E(x_2) = E(x_1 \times x_2)$ and $E(x_1) \oplus E(x_2) = E(x_1 + x_2)$, where $\otimes$ and $\oplus$ are two operations on the encryption space, while $\times$ and $+$ are the underlying operations on the message space. The basic leveled FHE encryption scheme works as follows [2].

1) E.Setup (1^λ, 1^μ, b): λ is the security parameter, representing 2^λ security against known attacks. Use the bit

$b \in \{0,1\}$ to select the parameters between a LWE-based scheme and RLWE-based scheme. Choose a μ-bit modulus M and choose the parameters $d = d(\lambda, \mu, b)$, $n = n(\lambda, \mu, b)$ and $\chi = n(\lambda, \mu, b)$ appropriately.
2) E.SecretKeyGen ($params$): Sample $\mathbf{s}' \leftarrow \chi^n$. Set $sk = \mathbf{s} \leftarrow (1, \mathbf{s}'[1], ..., \mathbf{s}'[n]) \in R_M^{n+1}$, which $R = R(\lambda)$ be a ring.
3) E.PublicKeyGen ($params$, sk): Generate (n+1)-column matrix $\mathbf{A}$' $\leftarrow R_M^{N\times n}$ uniformly and a vector $\mathbf{e} \leftarrow \chi^{\mathbf{N}}$ and set $\mathbf{b} \leftarrow \mathbf{A}'\mathbf{s}' + 2\mathbf{e}$. Set the public key $pk = \mathbf{A}$.
4) E.Enc($params$,pk,m): Assume the plaintext space is $R_2 = R/2R$. To encrypt a message $m \in R_2$, set $\mathbf{m} \leftarrow (m, 0, ..., 0) \in R_M^{n+1}$, sample $\mathbf{r} \leftarrow R_2^N$ and output the ciphertext $\mathbf{c} \leftarrow \mathbf{m} + \mathbf{A}^T\mathbf{r} \in R_M^{n+1}$.
5) E.Dec($params, pk, m$): Output $m \leftarrow [[\langle \mathbf{c}, \mathbf{s} \rangle]_M]_2$.

A recent work in [10] implemented the Advanced Encryption Standard (AES) homomorphically using this leveled FHE scheme, which took about 36 hours on a PC to evaluate a single AES encryption operation. It is too slow for any practical applications. In this implementation [10], for the smallest case (the depth $L = 10$) the dimension for the public key matrix is 9326, with the modulus q an odd number ranging from 512 to 2,048 bits. As shown above, the crucial part in encryption is a matrix-vector multiplication and decryption is actually a vector-vector multiplication. In this paper, we focus on accelerating the matrix-vector multiplication which is considered the most computation intensive part in the leveled FHE encryption scheme.

III. Software Implementation on CPU

A. CRT Representation

As mentioned in [10], the modulus is an odd number from 512 to 2,048 bits. For the matrix-vector multiplication, the computations are essentially large-number multiplications with each multiplicand in the size of 512 to 2,048 bits. This is similar to the modular multiplication in RSA. In this paper, we choose a medium size modulus of 1,024 bits for evaluation. The CRT method has been used widely in reducing the computational complexity for RSA encryption [11]. Hereby, we propose to apply the CRT method to the element-by-element modular multiplication and addition for the matrix-vector multiplication. We can choose a special odd number for M. When CRT is applied, it can be broken into 32 coprime pairwise modulies with each 32 bits.

B. Barrett Reduction

Initially, the 1,024-bit number is decomposed into 32 integers each with 32 bits during CRT decompose process. In the vector operation process, a modular reduction is required after each 32-bit by 32-bit multiplication. Thus an efficient modular multiplication is crucial for software implementation. Montgomery reduction [12] and the Barrett reduction algorithms [13] are the most popular modular reduction algorithms. Compared with Barrett reduction, Montgomery reduction needs extra computational steps to convert integers into Montgomery domain and later convert back from Montgomery domain. So Barrett method is employed for the modular reductions in this evaluation.

Given x, y and M with each n-bit number, the Barrett modular multiplication approach computes $r = xy \bmod M$. Barrett method shown in Algorithm 2 requires the precomputation of $\mu = \left\lfloor \frac{2^{2n}}{M} \right\rfloor$ ($n = \lceil \log_2(M) \rceil$) to reduce computation complexity.

C. Software Implementation

The matrix-vector multiplication involves a set of N dot-product operations if the matrix has N columns. The decompose process using CRT can be precomputed so we exclude the execution time of the decompose process in the evalu-

Algorithm 1 Dot Product Using Chinese Remainder Theorem

```
Procedure: $\mathbf{a} \times \mathbf{b} \bmod \mathbf{M} = a_0 b_0 + a_1 b_1 + ... + a_{N-1} b_{N-1}$.

Decompose: Let the numbers $m_0, ..., m_{k-1}$ be positive integers which are pairwise coprime with product $M = \prod_{i=0}^{k-1} m_i$. Thus the large numbers $a_0$,...,$a_{N-1}$and $b_0$,...,$b_{N-1}$ can be decomposed as follows. The decompose process can be precomputed.

$a_{0,0} = a_0 \bmod m_0$, $a_{0,1} = a_0 \bmod m_1$,..., $a_{0,k-1} = a_0 \bmod m_{k-1}$
$a_{1,0} = a_1 \bmod m_0$, $a_{1,1} = a_1 \bmod m_1$,..., $a_{1,k-1} = a_1 \bmod m_{k-1}$
    $\vdots$    $\vdots$
$a_{N-1,0} = a_{N-1} \bmod m_0$, $a_{N-1,1} = a_{N-1} \bmod m_1$,..., $a_{N-1,k-1} = a_{N-1} \bmod m_{k-1}$

$b_{0,0} = b_0 \bmod m_0$, $b_{0,1} = b_0 \bmod m_1$,..., $b_{0,k-1} = b_0 \bmod m_{k-1}$
$b_{1,0} = b_1 \bmod m_0$, $b_{1,1} = b_1 \bmod m_1$,..., $b_{1,k-1} = b_1 \bmod m_{k-1}$
    $\vdots$    $\vdots$
$b_{N-1,0} = b_{N-1} \bmod m_0$, $b_{N-1,1} = b_{N-1} \bmod m_1$,..., $b_{N-1,k-1} = b_{N-1} \bmod m_{k-1}$

Vector Operations:
$t_0 = (a_{0,0}b_{0,0} + a_{1,0}b_{1,0} + ... + a_{N-1,0}b_{N-1,0}) \bmod m_0$,
$t_1 = (a_{0,1}b_{0,1} + a_{1,1}b_{1,1} + ... + a_{N-1,1}b_{N-1,1}) \bmod m_1$,
    $\vdots$
$t_{k-1} = (a_{0,k-1}b_{0,k-1} + a_{1,k-1}b_{1,k-1} + ... + a_{N-1,k-1}b_{N-1,k-1}) \bmod m_{k-1}$,

Reconstruction: The dot product result can be reconstructed as follows.
$\mathbf{a} \times \mathbf{b} \bmod \mathrm{M} = \sum_{i=0}^{k-1} t_i v_i M_i$,
where $M_i = M/m_i$, and $v_i = M_i^{-1} \bmod m_i$.
```

Algorithm 2 Barrett Modular Multiplication

```
Procedure: $r = xy \bmod M \ (x < M, y < M)$
Precomputation: $n = \lceil \log_2(M) \rceil$,$\mu = \left\lfloor \frac{2^{2n}}{M} \right\rfloor$
Process:
$z = xy$;
$r = z - \left\lfloor \left\lfloor \frac{z}{2^n} \right\rfloor \frac{\mu}{2^n} \right\rfloor M$;
while $(r \geq M)$
    $r = r - M$;
end while;
retrun $r$;
end procedure
```

ation. We implement the matrix-vector multiplication using the CRT method using C/C++. We validate the results for our CRT implementation by comparing to the function of the large-number matrix-vector multiplication in NTL library[14]. Random numbers generated by C code are used as test vectors. From Table I, it shows that the CRT method is about 7.8 faster than the function in the NTL library when both executing on a PC. Also the vector operations take about 99.6% of the total computing time in the CRT method, which is the most computation-intensive part. As a result, we propose to use GPU to accelerate the vector operations, while leaving the reconstruction process and other remaining operations in the CPU.

Table I
PERFORMANCE COMPARISON AMONG NTL AND THE CRT METHOD

	Vector Operations	Reconstruction	Total
NTL library	555.4 sec	0	555.4 sec
CRT method	71.2 sec	0.343 sec	71.5 sec
Speedup	—-	—-	7.8

IV. GPU IMPLEMENTATION

Two CUDA kernel functions are developed to implement the steps of vector operations as in Algorithm 1. The first CUDA kernel function is kernel_BarretMulMod( ), which computes $r = xy \bmod M \ (x < M, y < M)$ described in Algorithm 2. To save memory space, the resulted matrix overwrites the input matrix since their dimensions are exactly the same. The other kernel function is kernel_addmodcal( ) used for modular additions. Both kernel functions use two-dimensional block and thread indexing, as explicitly parallel processing in the GPU.

The size of input matrix can be too large to fit into the GPU memory. For example, if the matrix has 9326×9326 elements, and each element is converted to 32 integer numbers each with 32-bits, then the memory size is 10.4 GB for this matrix only. To solve this problem, we divide the input data into several sections and keep them independent during computation. The GPU kernel function process one section of the data each time and the data of next section is transferred from host memory to GPU memory simultaneously. Thus the computation and data transfer are completely overlapped, thus the data transferring time is hidden. Based on our experiment for this particular case, this method can achieve a speedup of 1.96 in performance compared to non-overlapped GPU version shown in Table II. Fig 1 illustrates the process of the overlapped implementation. We allocate CUDA page-locked (pinned) memory for the input data to enable asynchronous data transfers. Two CUDA streams are created: one for computing and the other for data transfer. CPU and GPU synchronization is performed at the end of each section to ensure all computation and data transfer are completed. The pointer of the used data is exchanged with that of the newly imported data, making the two memory blocks ready for the next section.

Table II
PERFORMANCE COMPARISON AMONG OVERLAPPED GPU AND NON-OVERLAPPED GPU

	Vector Operations
None-overlapped GPU version	3.32 sec
Overlapped GPU version	1.69 sec
Speedup	1.96

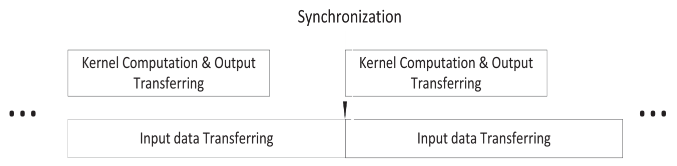
Synchronization
Kernel Computation & Output Transferring
Kernel Computation & Output Transferring
Input data Transferring
Input data Transferring
...
...

Figure 1. Overlapping computation and data transfer

V. EXPERIMENTAL RESULTS

As a case study, the CRT-based matrix-vector multiplication are evaluated on a desktop computer with Intel i5 3570K processor running at 3.4 GHz, 32 GB DDR3 RAM and one NVIDIA Tesla K20, which has 2,496 cores, 5GB DDR5 memory. Shoup's NTL library [14] is used for performance comparison and result validation.

Here we employ the smallest setting in [10] with a matrix dimension of 9,326 and the size of modulus M has 1,024 bits. In the CRT-based method, each 1,024 element is first decomposed into 32-bit small words. As mentioned in Section 3, our CRT-based matrix-vector multiplication is about 7.8 faster than the NTL library function on the CPU. Since the vector operation process takes 99.6% of the total calculation time, we use GPU to accelerate this vector operation process. When implemented on GPU, the vector operation process takes about 1.69 seconds which is 42.1 times faster than its implementation on CPU as shown in Table III. We compare the NTL-based calculation time on CPU, the CRT-based method on CPU, and the CRT-based method with GPU acceleration. The results are listed in Table IV and Fig. 2.

Table III
PERFORMANCE COMPARISON OF VECTOR OPERATION PROCESS AMONG CPU AND CPU PLUS GPU

	Vector Operations	Total Cal. Time
CPU	71.2 sec (CPU)	71.5 sec
CPU plus GPU	1.69 sec (GPU)	2.03 sec
Speedup	42.1	35.2

Table IV
PERFORMANCE COMPARISON AMONG NTL, CRT ON CPU AND CRT WITH GPU

	Calculation Time	Speedup
NTL on CPU	555.4 sec	1
the CRT-based method on CPU	71.5 sec	7.8
the CRT-based method with GPU	2.03 sec	273.6

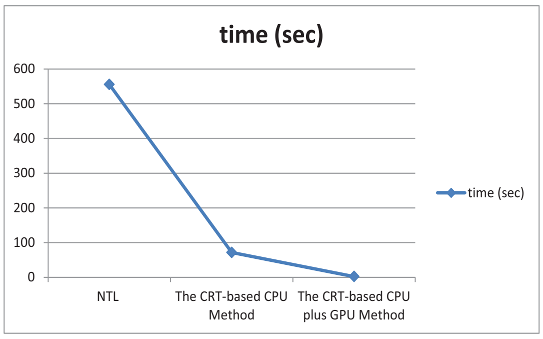

Figure 2. Execution time comparison

In the smallest case with the dimension of matrix 9,326 and the modulus 1,024 bits, it requires about 10.4 GB memory to store a matrix. From Table V, we can see that larger memory space is needed as the matrix dimension grows. Given the limitation of 32 GB RAM in the computer we use, only the small case is evaluated as an initial study.

Table V
MEMORY SPACE IN DIFFERENT SETTINGS

Matrix Dimension	Memory Space for one Matrix
9326	10.4 GB
19434	45.0 GB
29749	105.5 GB
40199	192.6 GB
50748	307.0 GB
61376	898.1 GB

VI. CONCLUSION

In this paper, the CRT method is used to implement the large-number matrix-vector multiplication. Compared to the NTL library function, the CRT-based method gains about 7.8 speedup when both executing on CPU. In order to further accelerate the matrix-vector multiplication, we use GPU to accelerate the vector operation process, which accounts for 99.6% of the total computation. In the GPU implementation, we manage to overlap the calculation process and data transfer process to improve the computation efficiency. Experimental results show the proposed CRT-based method with GPU implementation gains about 273.6 times speedup when compared with the NTL library function and 35.2 times speedup when compared with the same CRT-based method on CPU.

REFERENCES

[1] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proc. of the 41st Annual ACM Symposium on Theory of Computing*, 2009, pp. 169–178.
[2] 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.
[3] Z. Brakerski and V. Vaikuntanathan, "Efficient fully homomorphic encryption from (standard) lwe," in *Foundations of Computer Science (FOCS), 2011 IEEE 52nd Annual Symposium on*. IEEE, 2011, pp. 97–106.
[4] ——, "Fully homomorphic encryption from ring-lwe and security for key dependent messages," in *Advances in Cryptology–CRYPTO 2011*. Springer, 2011, pp. 505–524.
[5] 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 44th symposium on Theory of Computing*. ACM, 2012, pp. 1219–1234.
[6] C. Gentry and S. Halevi, "Implementing Gentry's fully-homomorphic encryption scheme," *Advances in Cryptology–EUROCRYPT 2011*, pp. 129–148, 2011.
[7] W. Wang, Y. Hu, L. Chen, X. Huang, and B. Sunar, "Accelerating fully homomorphic encryption using GPU," in *Proc. of 2012 IEEE Conference on High Performance Extreme Computing*. IEEE, 2012, pp. 1–5.
[8] ——, "Exploring the Feasibility of Fully Homomorphic Encryption," *IEEE Transactions on Computers*, 02 Aug. 2013.
[9] *CUDA C PROGRAMMING GUIDE*, 5th ed., NVIDIA, July 2013.
[10] C. Gentry, S. Halevi, and N. P. Smart, "Homomorphic evaluation of the aes circuit," in *Advances in Cryptology–CRYPTO 2012*. Springer, 2012, pp. 850–867.
[11] J. Grossschadl, "The chinese remainder theorem and its application in a high-speed rsa crypto chip," in *Computer Security Applications, 2000. ACSAC'00. 16th Annual Conference*. IEEE, 2000, pp. 384–393.
[12] P. Montgomery, "Modular Multiplication without Trial Division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, 1985.
[13] P. Barrett, "Implementing the Rivest Shamir and Adleman public key encryption algorithmstandard digital signal processor."
[14] V. Shoup, "NTL: A Library for Doing Number Theory," 2001.