Efficient Implementation of Paillier On GPUs

Xiongwen Guo #1

# *School of Information, Renmin University of China*
Beijing, China
1 xiongwenguo@ruc.edu.cn

***Abstract*—Homomorphic Encryption (HE) is used in many fieldsis due to its support for certain computable functions on encrypted data. Compared to conventional encryption schemes, HE allows a third party to work on the encrypted data without need to decrypt it. Based on the type and number of operations that are permitted, homomorphic encryption can be categorized into three classes[1]: Partially Homomorphic Encryption (PHE), Somewhat Homomorphic Encryption (SWHE) and Fully Homomorphic Encryption (FHE). Paillier is a widely-used PHE scheme. This work explores efficient implementation of Paillier scheme on GPUs and discusses difficulties and limitations during the process. The code is available at: https://github.com/GuoXiongwen/py-cuda-paillier.**

I. Introduction

Among three types of Homomorphic Encryption scheme, PHE is computationally efficient and supports infinite additions or multiplications. What's more, there exist some specific scenarios in practical applications that require the use of only one homomorphic encryption operation. Due to the above reasons, PHE becomes an important component of privacy computation, which can assist in accomplishing a variety of privacy computation functions. Paillier[2], a PHE scheme which supports homomorphic addition, stands out due to its higher efficiency and complete security proof and becomes one of the most commonly used PHE instantiation schemes in privacy computing scenarios.

Since Paillier was proposed, several attempts have been made to optimize the Paillier scheme. D. Catalano et al.[3] proposed that Paillier can take $g = n+1$ in the key generation phase while using the Binomial Theorem to reduce the cost of calculating g^m. I. Damgård et al. proposed Paillier-DJN[4], a simplified version of Paillier scheme, which is the optimal version of Paillier so far. The two aforementioned schemes improve the performance from a theoretical perspective. Unlike them, this work focuses on improving the efficiency of Paillier by implementing on GPUs instead of CPUs, leveraging the powerful parallel computing capabilities of GPUs.

II. Background

In order to reach the goal of realizing high-performance Paillier scheme on GPUs, we first review the Paillier scheme, which includes key generation phase and application phase. The application includes encryption, decryption, homomorphic addition. The scheme works as follows:

A. Key generation

1) Choose two large prime numbers p and q of equal length randomly such that $\gcd(pq, (p-1)(q-1)) = 1$.
2) Compute $n = pq$ and $\lambda = \operatorname{lcm}(p-1, q-1)$.
3) Select random integer g where $g \in \mathbf{Z}^*_{n^2}$.
4) Define function L as $L(x) = \frac{x-1}{n}$, i.e., $L(x)$ is the quotient of $(x-1)$ divided by n and compute $\mu = (L(g^\lambda \bmod n^2))^{-1} \bmod n$.
5) The public key is (n, g) and private key is (λ, μ).

B. Encryption

1) Let m be the plaintext where $0 \leq m < n$.
2) Select random r where $0 < r < n$ and $\gcd(r, n) = 1$
3) The ciphertext is: $c = (g^m \cdot r^n) \bmod n^2$.

C. Decryption

1) Let c be the ciphertext where $c \in \mathbf{Z}^*_{n^2}$.
2) The plaintext is: $m = (L(c^\lambda \bmod n^2) \cdot \mu) \bmod n$.

D. Homomorphic addition

1) Let c_1 and c_2 be two ciphertexts where $c_1, c_2 \in \mathbf{Z}^*_{n^2}$.
2) Compute encrypted addition result of $c_1 + c_2$ as $c = (c_1 \cdot c_2) \bmod n^2$.

III. Optimization and implementation

A. Performance Bottleneck Analysis

After algorithm analysis and performance testing, we found that the Paillier scheme is a computationally intensive program, and the power operation is the most time-consuming performance bottleneck in the whole process. In order to improve the performance of the Paillier program, efficiency of calculating a^b or $(a^b \bmod m)$ must be improved.

B. Optimization Methods for Bottleneck

Naive implementation of a^b is to multiply a by b times, of which time complexity is $O(b)$. Furthermore, suppose b is a power exponent of 2, we can calculate value of a^b as $a^b = (a^{\frac{b}{2}})^2$ and calculate $a^{\frac{b}{2}}$ recursively. In this way, the time complexity can be optimized to $O(\log(b))$. When generalizing the operation a^b to $(a^b \bmod m)$, we can calculate value of $(a^b \bmod m)$ as $(a^b \bmod m) = (a^{\frac{b}{2}} \bmod m)^2 \bmod m$ and calculate $(a^{\frac{b}{2}} \bmod m)$ recursively. Generally, even if b is not a power of 2, we can use a similar approach to get the calculation result in $O(\log(b))$ complexity.

C. Implementation in Python

Python's built-in functions `pow(a,b)` and `pow(a,b,m)` are implemented roughly following the same logic as above. Since only `pow(a,b)` is supported in CUDA, we implement `pow(a,b,m)` in CUDA by ourselves.

IV. Experiments

A. Experiment Setup

We implemented the Paillier scheme on GPUs with Python based on open-source code repository [5][6] and tested the correctness and performance of our implementation. The type of GPUs used is GeForce RTX 2080 Ti. We only focus on the performance of encryption and decryption because others are not performance bottlenecks.

In each round of the performance test, we encrypt a plaintext message (denoted as a list of integers of length N) with a pre-generated public key and decrypt a ciphertext message (also denoted as a list of integers of length N) with a pre-generated private key. To minimize the impact of fluctuations on the results, we repeat K rounds to take the average value.

B. Results of Different Methods

We chose $p = 61$ and $q = 43$ to generate private key ($\lambda = 420, \mu = 2389$) and public key ($n = 2623, g = 4154869$). We implemented the Paillier scheme using 4 methods. Specifically, method 1 and 2 are programs on CPUs and method 3 and 4 are programs on GPUs. Method 1 and 3 implement calculating ($a^b \bmod m$) in $O(b)$ complexity. Method 2 and 4 implement in $O(\log(b))$ complexity (in fact, method 2 calls the built-in Python function `pow(a,b,m)` directly). We recorded not only the total time the program ran, but also the time the program consumed on computation, so that we could observe the time the program consumed on other parts of the program (e.g., moving data between CPUs and GPUs). In the experiments, both the plaintext and ciphertext list lengths N were set to 16384.The time consumed by the four methods on encryption is shown in Table I, and the time consumed on decryption is shown in Table II.

TABLE I
Encryption Time

Method	Computing Time (s)	Total Time (s)
1	12.759118	12.760379
2	2.688422	2.689688
3	0.121923	0.524141
4	0.264637	0.713463

TABLE II
Decryption Time

Method	Computing Time (s)	Total Time (s)
1	1.049174	1.049417
2	0.031346	0.031470
3	0.121339	0.133358
4	0.194023	0.206022

C. Analysis and Discussion

It can be seen that during the encryption process, no matter time complexity is $O(b)$ or $O(\log(b))$, implementation on GPUs is faster than that on CPUs. During the decryption process, method 3 on GPUs is faster than method 1 on CPUs, which is consistent with our expectation.

Surprisingly, method 3 with $O(b)$ time complexity on GPUs is faster than method 4 with $O(\log(b))$ time complexity on GPUs, both in terms of computing time and total time. We analyze that the reason may be that method 4 requires frequent if-else branch selection which reduces its speed. Besides, the value of the exponent b is not large, the performance gain of reducing the time complexity from $O(b)$ to $O(\log(b))$ is not large. What's more, the most surprising thing is that the best performance during the decryption process is achieved by method 2, which is implemented on CPUs. We analyze the possible reasons may be that the number of cores on a GPU limits the theoretical peak of parallel processing or Python's built-in function `pow(a,b,m)` use other algorithms to boost its performance of calculating ($a^b \bmod m$).

V. Conclusion

This work investigates efficient implementation of Paillier scheme on GPUs. By implementing and evaluating Paillier scheme on GPUs, we find the bottleneck of the program and discuss the difficulties and limitations.

Acknowledgment

We sincerely appreciate all the teachers and TAs for their dedication to the course *Parallel and Distributed Computing*.

References

[1] A. Acar, H. Aksu, A. S. Uluagac, and M. Conti, "A survey on homomorphic encryption schemes: Theory and implementation," *ACM Computing Surveys (Csur)*, vol. 51, no. 4, pp. 1–35, 2018.

[2] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Proceedings of the 17th International Conference on Theory and Application of Cryptographic Techniques*, ser. EUROCRYPT'99. Berlin, Heidelberg: Springer-Verlag, 1999, p. 223–238.

[3] D. Catalano, R. Gennaro, N. Howgrave-Graham, and P. Q. Nguyen, "Paillier's cryptosystem revisited," in *Proceedings of the 8th ACM Conference on Computer and Communications Security*, 2001, pp. 206–214.

[4] I. Damgård, M. Jurik, and J. B. Nielsen, "A generalization of paillier's public-key system with applications to electronic voting," *International Journal of Information Security*, vol. 9, pp. 371–385, 2010.

[5] [Online]. Available: https://github.com/homo-paillier-cryprosystem/py-paillier/

[6] [Online]. Available: https://github.com/homo-paillier-cryprosystem/py-cuda-paillier/