

FPGA-Based Hardware Accelerator of Homomorphic Encryption for Efficient Federated Learning

Zhaoxiong Yang¹, Shuihai Hu², Kai Chen^{1,3}

¹SING Lab, Hong Kong University of Science and Technology

²Clustar, ³Peng Cheng Lab

zyangas@connect.ust.hk, shuihai@clustar.ai, kaichen@cse.ust.hk

Abstract

With the increasing awareness of privacy protection and data fragmentation problem, federated learning has been emerging as a new paradigm of machine learning. Federated learning tends to utilize various privacy preserving mechanisms to protect the transferred intermediate data, among which homomorphic encryption strikes a balance between security and ease of utilization. However, the complicated operations and large operands impose significant overhead on federated learning. Maintaining accuracy and security more efficiently has been a key problem of federated learning. In this work, we investigate a hardware solution, and design an FPGA-based homomorphic encryption framework, aiming to accelerate the training phase in federated learning. The root complexity lies in searching for a compact architecture for the core operation of homomorphic encryption, to suit the requirement of federated learning about high encryption throughput and flexibility of configuration. Our framework implements the representative Paillier homomorphic cryptosystem with high level synthesis for flexibility and portability, with careful optimization on the modular multiplication operation in terms of processing clock cycle, resource usage and clock frequency. Our accelerator achieves a near-optimal execution clock cycle, with a better DSP-efficiency than existing designs, and reduces the encryption time by up to 71% during training process of various federated learning models.

1 Introduction

In recent years, deep learning has made an unprecedented leap in the ability of human discovering knowledge and comprehending the world. Nevertheless, the adoption of deep learning is now faced with two barriers, namely data fragmentation and privacy preservation[Yang *et al.*, 2019]. Federated learning has come up as a new machine learning paradigm to tackle the issues, learning models from decentralized datasets in a secure way.

To preserve data privacy, federated learning usually employs various mechanisms like differential privacy (DP), ho-

momorphic encryption (HE), secure multiparty computation (SMC), etc. Whereas DP does not prevent data leakage completely, and the intricate protocols that SMC introduces to the system renders it virtually impractical, HE achieves a balance between security and operability. Moreover, one of the HE scheme named Paillier encryption scheme[Paillier, 1999] has been adopted to protect the data privacy in neural networks[Ma *et al.*, 2017], logistic regression[Hardy *et al.*, 2017], Bayesian network[Wright and Yang, 2004], clustering[Bunn and Ostrovsky, 2007], showcasing its great generality as a privacy preserving mechanism in machine learning.

However, the complicated operations and large operands of HE still impose overhead on federated learning that cannot be ignored. Research community and industry have been haunted by the question of how to provide *secure*, *accurate*, and yet *efficient* federated learning. Previous effort such as FATE[FAT, 2019], a cutting edge federated learning system, has provided convenient interface to implement learning algorithms secured by Paillier HE, but the learning throughput is limited due to encryption by software. In this work, we seek for a hardware solution to improve the training throughput of federated learning, designing a homomorphic encryption framework based on FPGA, since FPGA acceleration card has been commonly available in datacenters[Putnam, 2014] and usually achieve a lower power consumption than GPU. The framework devises a customized FPGA implementation of the Paillier homomorphic encryption, and provides support for federated learning models with secure information exchange.

We demonstrate in this work that homomorphic encryption is usually composed of iterative operations that are hard to parallelize. Therefore, it is more reasonable to consider parallelism across data items to be encrypted, and make each encryption core compact and resource efficient, so as to maximize the overall throughput to handle the massive data in federated learning. The existing works fail to do that, as they either try to exhaust the resource on a single FPGA chip to produce one encryption unit to minimize the processing latency, or they mainly utilize the common circuit units (usually termed CLB or LUT) without making use of the digital signal processing (DSP) units, which are the powerful units for high performance arithmetic operation on modern FPGA. Moreover, most of them rely on the traditional register-level

transfer (RTL) approach, lacking the flexibility of fast development and reconfiguration. In this work, we base our design and implementation on high level synthesis (HLS) that describe the FPGA circuit with high-level programming language for flexibility, allowing the algorithm and operations to be parametric and portable, and we try to derive an analytical model that determines the encryption performance, carry out optimization from multiple dimension.

Since the bulk of computation of Paillier cryptosystem boils down to modular multiplication(ModMult), we focus on designing compact architecture for ModMult operation. We adopt the Montgomery algorithm[Montgomery, 1985] to carry out the operation, which is FPGA-friendly as it eliminates integer division operations. We figure out the key factors that determines the total en/decryption throughput on an FPGA chip, conduct overall optimization on Paillier processors in terms of clock cycle, resource consumption, clock frequency and memory usage respectively to attain the best throughput.

The hardware module are built as OpenCL kernels and incorporate into FATE as an encryption library. Each kernel performs en/decryption for a batch of data to relieve the kernel invocation overhead, and kernels are queued in the OpenCL command queue to help overlap data transfer with computation and hide latency. The proposed encryption framework is general and does not require any change of the model, while preserving the security and accuracy.

We perform extensive evaluation on the proposed framework, demonstrating that it reduces the iteration time for training linear models by up to 26%, and the encryption time in each iteration by 71%. Our hardware framework delivers an acceleration ratio of 10.6 for encryption and 2.8 for decryption compared with software solutions. Our circuit for ModMult operation achieves a better DSP-efficiency than existing FPGA solutions, with a comparable execution latency but a lower usage of DSP blocks.

We summarize our contributions as follows.

- Introducing a hardware-based encryption framework for federated learning, achieving high efficiency without sacrifice of security and utility, supporting accelerated computation in cloud datacenters.
- Presenting architectures for Paillier homomorphic cryptosystem taking a scalable approach making efficient utilization of the FPGA resources, especially DSP blocks.
- Incorporating the encryption framework into cutting-edge federated learning framework, and showing an improvement on training throughput for federated learning models.

The rest of the article will be organized as follows. In Section 2 we will provide the background about federated learning and existing privacy preserving machine learning systems, and introduce the Paillier cryptosystem we work on. Section 3 will present the design and implementation of the framework in detail. Section 4 shows the methodology and results of evaluation. Finally in Section 5 we conclude the article.

2 Background

2.1 Federated learning with HE

Federated learning is a privacy-preserving, decentralized distributed machine learning paradigm. One effective method of preserving privacy and securing computation is homomorphic encryption (HE), i.e. encryption schemes that allows encrypted values to be involved in computation. For the applications of HE in federated learning, we refer the readers to [Hardy *et al.*, 2017], [Gilad-Bachrach *et al.*, 2016], [Aono *et al.*, 2017], [Liu *et al.*, 2019], [Liu *et al.*, 2018], [Chai *et al.*, 2019], which broadly cover machine learning models including linear model, neural network and deep learning, boosting tree, transfer learning and matrix factorization. Typically, HE is employed to encrypt the intermediate data during computation, which will then be transferred and aggregated by homomorphic operation. For nonlinear operations composing the model, such as activation function in a neural network, these works usually rely on approximation to make the model agree with HE computation.

2.2 Privacy-preserving ML systems

There has also been machine learning systems that take privacy preservation into account, such as SecureML[Mohassel and Zhang, 2017] that proposes a system for two non-colluding party collectively training a model, and Sage[L'ecuyer *et al.*, 2019] that presents a differentially private machine learning platform. Among them, FATE[FAT, 2019] introduces a federated learning framework that provides the abstraction and utilities for implementing algorithm and models, along with an architecture to enable distributed, multiparty machine learning. It mainly utilizes Paillier homomorphic encryption to guarantee data security. However, it purely relies on a software solution of encryption that greatly harms the execution efficiency of federated training. Our goal in the work is to find a hardware solution as a rescue to this issue.

2.3 Paillier Homomorphic Encryption

Paillier HE is an additive homomorphic encryption scheme allowing to perform addition and multiplication with scalar on encrypted values without decrypting them. In federated learning, usually multiple parties are involved, each one having a private dataset and wanting to maintain a local model learned from the aggregated dataset, and there may be a coordinator to manage the computation and data exchange among parties (Figure 1). The role of Paillier homomorphic encryption is to encrypt the intermediate data to transfer, so that in each training iteration the coordinator receives the encrypted local updates from parties, aggregates them with the homomorphic property, and sends back the result to each party for decryption and updating local model. In this way, each party obtain a model extracting information from the aggregate dataset, without leaking its private information.

The Paillier HE scheme associates each party with a public key (n, g) and a private key (λ, μ) , where n, g, λ, μ are large integers, typically 1024-bit in FATE. Messages and ciphertexts are also represented as long integers. A message m can be encrypted into ciphertext c by $c = g^m r^n \pmod{n^2}$ with

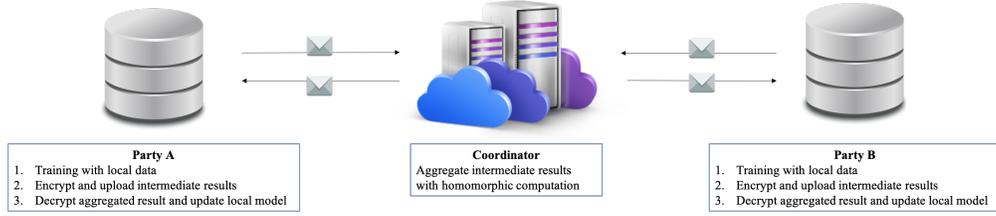


Figure 1: General workflow of homomorphic encryption-based federated learning

random number r , and decryption is performed by $m = ((c^\lambda \bmod n^2) - 1)/n * \mu \bmod n^2$.

We can see from the formulation that the majority of the computation of the Paillier en/decryption is related to modular exponentiation (ModExp), which can be further decomposed to a series of ModMult operations. Hence, the execution of ModMult has a decisive effect on the overall performance. We choose the Montgomery ModMult algorithm [Montgomery, 1985] to perform this operation because it is FPGA-friendly, in that it disposes of the costly integer division. The Montgomery algorithm, shown in Algorithm 1, computes $XY \cdot 2^{-l} \bmod M$ for l -bit integers X , Y and M . It divides integers into k -bit words. The body of the algorithm is a two-level loop, where each outer iteration (line 2-8) aims to compute an intermediate result $S_i = X \cdot Y^i \cdot 2^k \bmod M$ for the i th word of Y , and it further decomposes the computation by each word of X and forms the inner loop (line 4-6).

Algorithm 1: Montgomery Algorithm for Modular Multiplication with Radix 2^k

Input: $X = \sum_{j=0}^{l/k-1} X^j \cdot 2^{jk}$, $Y = \sum_{j=0}^{l/k-1} Y^j \cdot 2^{jk}$,
 $M = \sum_{j=0}^{l/k-1} M^j \cdot 2^{jk}$, $r = 2^k$

Output: $S = X \cdot Y/2^l \bmod M$

```

1  $S_0 \leftarrow 0$ ;
2 for  $i = 0 \dots l/k - 1$  do
3    $q \leftarrow ((S_i + X * Y^i) \cdot (-M^{-1})) \bmod r$ ;
4   for  $j = 0 \dots l/k$  do
5      $\tilde{S}_{i+1}^j \leftarrow S_i^j + X^j * Y^i + q * M^j$ ;
6   end
7    $S_{i+1} \leftarrow \tilde{S}_{i+1}/2^k$ 
8 end
9 if  $S_{l/k} > M$  then
10   $S_{l/k} \leftarrow S_{l/k} - M$ ;
11 end
12 return  $S_{l/k}$ 

```

3 Design and Implementation

3.1 System Overview

The overall architecture of our encryption framework is shown in Figure 2. The framework is envisioned to be hosted on cloud servers belonging to geo-distributed parties of federated learning. It includes components residing on both the

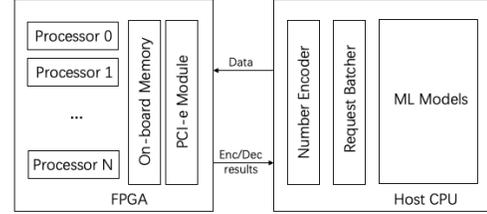


Figure 2: Overview of Our Encryption Framework

host CPU and the FPGA, where a PCI-e bus provides communication between them. The host CPU is responsible for the normal training workload of a machine learning model, while it batches the requests of encryption to send to the FPGA, and encodes the floating point number used by machine learning to integers agree with HE schemes. Apart from these necessities, our main contribution is designing high performance processors for Paillier computation on FPGA and encapsulating the hardware module as OpenCL kernel for invocation, which we will detail in Section 3.2 and 3.3 respectively.

3.2 Micro-architecture for Montgomery ModMult

A Paillier processor encapsulates units for operations involved, i.e. modular multiplication, random number generator and integer divisor, along with its local storage. We replicate Paillier processors in HLS to deploy multiple copies, and the top level function is responsible for dispatching input data and collecting results. Since the Paillier processors are independent and work in parallel, the overall throughput of an FPGA chip can be determined by

$$\text{Throughput} = \frac{\text{Total amount of resource}}{\text{Latency} \times \text{Resource consumption per core}},$$

where resource broadly refers to multipliers, adders, memory, etc., and latency can be further decomposed to clock cycle of execution \times clock frequency. Therefore, our design guideline is to optimize the Montgomery ModMult operation lying at the heart of Paillier cryptosystem, with respect to clock cycle, resource allocation, clock frequency, in addition to memory usage. We elaborate on the optimization on these dimensions as below.

Clock Cycle

Generally, the clock cycle required by an algorithm is intrinsically lower bounded by the number of operations and the

critical path in the dependency graph. As shown in Algorithm 1, the body of the Montgomery algorithm is a two-level loop, consisting of $2(l/k)(l/k + 1)$ multiplications. Thus, the ideal clock cycle will be $2(l/k)(l/k + 1)$ divided by the number of multipliers, even if we ignore the rest of the operations. On the other hand, as the execution of each inner iteration depends on the iteration before, it is hard to force a parallel execution of the inner iterations. Our goal is to deploy two multipliers for an inner iteration, and obtain a clock cycle number as close to $(l/k)(l/k + 1)$ as possible.

Another dependency issue that deserves attention is the computation of q in each outer iteration. In the i th iteration, q depends on the value of S_{i-1} , while it is necessary in the computation of inner loops. However, if q is computed before the start of inner loop, the latency will be magnified by the number of outer iterations.

To enforce a tight scheduling, we make the following optimizations in HLS:

- Unrolling the inner loop. This can be done through an UNROLL directive in HLS, or manually repeat part of the loop. Unrolling the loop does not lead to parallel execution of iterations. However, this is the only way to disassemble all the operations composing the loop, achieving the flexibility of scheduling to overlap operations as much as possible. Also, without unrolling we are not able to insert the computation of q into the middle of an inner loop.
- Interleaving the q computation with the inner loop. As discussed before, the q value used in each inner loop must be computed before. Since the q value for computing S_i only relies on first few words of S_{i-1} , it is possible to start generating q in the last inner loop when those words are ready. In this way we can obtain q in advance and hide its latency.
- Pipelining the outer loop. We achieve this by inserting a PIPELINE directive in HLS, with the initiation interval set to the number of iterations contained in an inner loop. The final step of schedule enforcement is to pipeline all the iterations. We aim to pipeline both the outer loop and the inner loop by unrolling the inner loop, and pipelining the outer loop, so that the inner loop is naturally pipelined by scheduling the disassembled operations, and the outer loop try to start an interaction each time when a whole inner loop is initiated.

The resulted scheduling is shown in Figure 3. We illustrate with an example with operands 4 words in length (i.e. $l/k = 4$), and the computation of each inner iteration takes 4 clock cycles to complete. Initially, the schedule computes q for the first inner loop (not shown in the figure), and then initiates the inner iterations sequentially. In the meantime, as soon as S^0 is ready, it can be used to compute q for the next inner loop. Hence, when the last inner iteration ends, the first iteration of the next inner loop can start immediately with the precomputed q . Therefore, we enable a tight schedule that initiates an inner iteration each clock cycle. The resulted execution clock cycle is $(l/k)(l/k + 1)$, plus the number of pipeline stages, and a few cycles for data read-in and write-out.

Resource Allocation

In this work, we utilize the embedded DSP blocks on the FPGA chip to construct pipelined multipliers. For the remaining logic, including adder, multiplexer, integer comparison, finite state machine, etc., we leave them to look-up-table (LUT). As DSP on FPGA is scarce and expensive, we use them to carry out the heavy multiplication only. Further, we will show purely relying on LUT to implement ModMult operation is not economic (Section 4). Therefore, we will focus on the usage of LUT and DSP, and reduce the area and DSP usage without sacrificing the performance.

We encapsulate the operations comprising an inner iteration into a processing element (PE), as shown in Figure 4. Each PE contains two multipliers to perform the two independent multiplications $x * y$ and $q * m$. Then it accepts S_{i-1}^j of the last outer iteration, and a carry word (not shown in the figure), adds them with the multiplication results, and then outputs S_i^j and a carry word. Then we limit the number of PE to 1, with an ALLOCATION directive in HLS. This is to avoid the resource bloating owing to loop unrolling, so that only resource for computing one inner iteration is actually allocated, and to reduce the overall area of the micro-architecture.

We also employ the Karatsuba algorithm to construct DSP-conservative multipliers. As shown in Algorithm 2, Karatsuba algorithm performs an integer multiplication by recursively breaking it into three of half size. Its efficiency is attributed to one multiplication less than the schoolbook algorithm, and we take advantage of it to allocate DSPs according to the actual number of operations. For instance, a DSP48E1 block is able to carry out 18×25 -bit multiplication, and a 32×32 one can be divided into 16×16 ones, and takes up 3 DSP blocks.

Algorithm 2: Karatsuba algorithm

Input: Operands X and Y , the length of operand k

Output: $S = X * Y$

- 1 Let $X = \overline{X_h X_l}$, $Y = \overline{Y_h Y_l}$, where X_h, X_l, Y_h, Y_l are $k/2$ -bit integers;
 - 2 $HH \leftarrow \text{Karatsuba}(X_h, Y_h)$;
 - 3 $LL \leftarrow \text{Karatsuba}(X_l, Y_l)$;
 - 4 $HL \leftarrow \text{Karatsuba}(X_h + X_l, Y_h + Y_l)$;
 - 5 $S \leftarrow HH * 2^k + (HL - HH - LL) * 2^{k/2} + LL$;
 - 6 **return** S
-

Clock Frequency

The DSP units on the Xilinx FPGA run at a maximum frequency of 400-500MHz. To approach the frequency limit, we need to pay attention to the following measures:

- Declare the multipliers as pipelined multipliers. A pipelined multiplier takes multiple cycles to accomplish a multiplication, distributing its workload and relieving the burden of each cycle. It does no harm to the multiplication throughput since we have resolved the dependency between its input and output.
- Restrict bitwidth of operands. The clock frequency is constraint by the critical path of the circuit, i.e. the

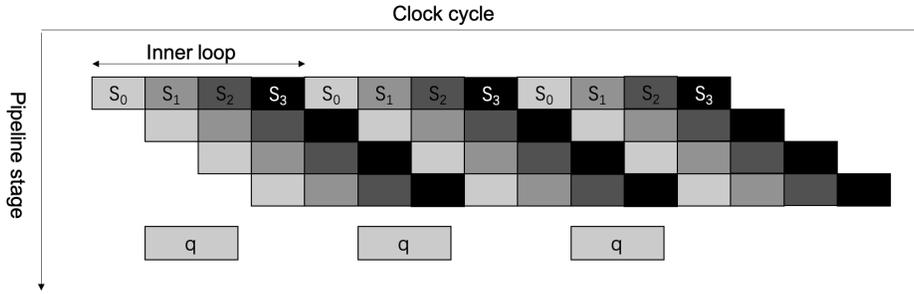


Figure 3: Pipeline Execution of the Montgomery ModMult Operation

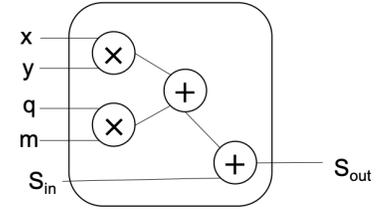


Figure 4: Processing Element Implementing the Inner Loop of Algorithm 1

longest path of gates a signal needs to pass through during one cycle. Arithmetic on integers such as addition or comparison usually results in a long carry chain, and thus we need to avoid computation on very long integers directly. In this work, we use 32-bit as the operand size, and the maximum bitwidth involved is 64 bits.

- Simplify the control logic. For the finite state machine in charge of controlling the compute units, we use one-hot encoding scheme to represent the states for a fast lookup and match. The number of states is related to the number of iterations of each loop and thus one-hot encoding will be acceptable.

Memory Usage

Our design allocate each Paillier processors its own block RAM (BRAM) as local buffer, to hold the input/output data and the intermediate large integers involved in the computation. We do not share storage among processors to prevent data access contention.

Large integers are normally stored as arrays of words in the BRAM. However, we notice that the input data for encryption, which are encoded from floating point numbers used in machine learning, have few effective digits compared with the length of large integers. Therefore, we are able to store the input data as a sparse vector, i.e. only recording the non-zero elements and their indices, reducing the memory footprint.

3.3 Implementation

We develop our encryption framework with the AWS F1 instance and Xilinx SDAccel development suite. The basic logic of the encryption and decryption function is implemented with Xilinx high level synthesis (HLS), allowing to transform an algorithm described in C/C++ into tailor-made implementation on FPGA. Directives like loop pipelining and instance allocation are inserted into the HLS code to fine tune the performance of the resulted architecture.

On the host side, we use the OpenCL API to access the acceleration hardware. The OpenCL API provides an abstraction of the computing device like CPU, GPU and FPGA. An invocation to the device function is named a kernel. OpenCL is used to manage the data transfer between host and device, queue and invoke kernels, and monitor the execution events. We adopt the PyOpenCL APIs to implement a module that

makes use of the FPGA device for cryptographic processes and incorporate it into the FATE framework.

The requests from the host side are divided into fixed size batches, and each batch invokes a kernel on device. Multiple kernels will be queued in the OpenCL command queue. This helps overlap data transfer with computation and hide latency. We also preallocate buffers on the device, arranging them as a ring buffer, in order to reuse buffers among kernels and avoid frequent memory allocation.



Figure 5: Queuing kernels for execution

4 Evaluation

We conduct experiments aiming to perform an extensive evaluation on the proposed encryption framework. We first perform a microscopic examination, comparing the implementation of Paillier algorithm and ModMult operation with software solutions and existing FPGA designs. Then we study its improvement on the overall performance of training process of federated learning. The training tasks are carried out on the open-sourced version of the FATE machine learning framework. We choose two linear models, and adopt Kaggle datasets on breast cancer¹ and motor temperature² and partition the datasets vertically.

We attempt to answer the following questions empirically with the evaluation experiments:

- How do the Paillier processors perform, especially for the ModMult operation, in terms of throughput and resource-efficiency?
- How does the hardware framework compare with software solutions of Paillier cryptosystem in terms of en/decryption throughput?
- How much does the framework affect the training throughput of federated learning with respect to different models or algorithms?

¹<https://www.kaggle.com/uciml/breast-cancer-wisconsin-data>

²<https://www.kaggle.com/wkirkgsn/electric-motor-temperature>

Implementation	Area(slice)	DSP	Clock frequency(MHz)	Execution time(us)	Throughput per DSP(op/s)
This work	483	9	500	8.81	12626
[San and At, 2014]	567	13	490	8.64	8903
[Song <i>et al.</i> , 2010]	180	1	447	135.4	7385
[Huang <i>et al.</i> , 2011]	9268	NA	129	18.70	NA

Table 1: Comparison of ModMult operation

Given the broad adoption of the ModMult operation, many implementations have been proposed by researchers, and we compare ours with them in Table 1. Since we are targeting datacenter acceleration chips and applications, the DSP efficiency is a key factor evaluating an implementation. Comparing with the state of the art solution [San and At, 2014], our ModMult module delivers a close latency but uses fewer DSPs due to our precise limit on resource usage. The authors of [Song *et al.*, 2010] propose an implementation using only one DSP and one block RAM. However, without employing the Karatsuba algorithm, their version turns out to be less efficient than ours. [Huang *et al.*, 2011] gives an implementation using circuit elements entirely without DSP, and it shows that an such a ModMult module consumes much area and limits the clock frequency, and hence not recommendable. Moreover, most of existing solutions are based on register-transfer level (RTL) that describes the circuit directly, but lacks the flexibility of parametrizing and reusing the ModMult module as our HLS version does.

To evaluate the effectiveness of the scheduling of ModMult operation, we compare the number of execution clock cycles with the theoretically ideal clock cycle, given as $T = (l/k)(l/k + 1)$ (Section 3). As shown in Figure 6, for different sizes of operands, our implementation keeps no more than 10% higher than the ideal. The gap is mainly due to pipeline stages, time for initialization and data transfer.

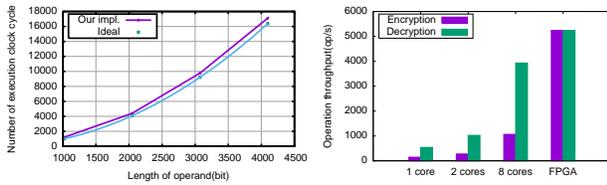


Figure 6: Number of execution clock cycles of ModMult operation
Figure 7: Throughput of FPGA and multicore processor

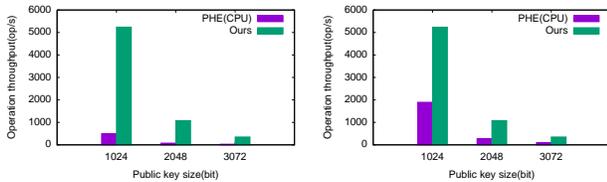


Figure 8: Encryption Throughput Compared with Software
Figure 9: Decryption Throughput Compared with Software

To investigate the performance of FPGA and software solution, we compare the framework with PHE, a popular Paillier library, as shown in Figure 8 and 9. We can see that for a 1024-bit public key, our framework delivers an acceleration ratio of $10.62\times$ and $2.76\times$ for encryption and decryption, respectively. We also compare FPGA with a multicore processor using `libpaillier` library, as shown in Figure 7. It shows that an FPGA effectively outperforms a multicore CPU and is advisable to be used in accelerating computational intensive applications.

Additionally, we test the modified FATE with linear models and the breast and motor datasets. We train a logistic regression and a linear regression model on the two datasets respectively for 10 iterations, and record the timing. Figure 10 and Figure 11 show the training iteration time and the encryption time in each iteration respectively. It demonstrates that for linear models, our framework reduce the training iteration time by up to 26%, and the encryption time during one iteration by 71.2%.

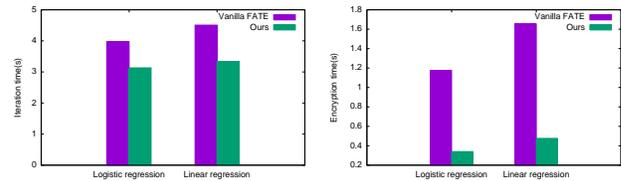


Figure 10: Improvement on Iteration Time
Figure 11: Improvement on Encryption Time Per Iteration

5 Conclusion

In this paper, we have demonstrated the significance of accelerating homomorphic encryption and modular operations. We explored a compact architecture for Paillier cryptosystem with an HLS-based approach, investigating how to optimize the performance, and incorporated the FPGA framework into a federated learning system. We conducted extensive experiments to present the effectiveness and efficiency of our encryption framework.

References

- [Aono *et al.*, 2017] Yoshinori Aono, Takuya Hayashi, Lihua Wang, Shiho Moriai, et al. Privacy-preserving deep learning via additively homomorphic encryption. *IEEE Transactions on Information Forensics and Security*, 13(5):1333–1345, 2017.
- [Bunn and Ostrovsky, 2007] Paul Bunn and Rafail Ostrovsky. Secure two-party k-means clustering. pages 486–497, 2007.
- [Chai *et al.*, 2019] Di Chai, Leye Wang, Kai Chen, and Qiang Yang. Secure federated matrix factorization. *arXiv preprint arXiv:1906.05108*, 2019.
- [FAT, 2019] Federated ai ecosystem. <https://fate.fedai.org/>, 2019.
- [Gilad-Bachrach *et al.*, 2016] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International Conference on Machine Learning*, pages 201–210, 2016.
- [Hardy *et al.*, 2017] Stephen Hardy, Wilko Henecka, Hamish Ivey-Law, Richard Nock, Giorgio Patrini, Guillaume Smith, and Brian Thorne. Private federated learning on vertically partitioned data via entity resolution and additively homomorphic encryption. *arXiv preprint arXiv:1711.10677*, 2017.
- [Huang *et al.*, 2011] Miaoqing Huang, Kris Gaj, and Tarek A. El-Ghazawi. New hardware architectures for montgomery modular multiplication algorithm. *IEEE Trans. Computers*, 60(7):923–936, 2011.
- [L’ecuyer *et al.*, 2019] Mathias L’ecuyer, Riley Spahn, Kiran Vodrahalli, Roxana Geambasu, and Daniel Hsu. Privacy accounting and quality control in the sage differentially private ml platform. *SIGOPS Oper. Syst. Rev.*, 53(1):75–84, July 2019.
- [Liu *et al.*, 2018] Yang Liu, Tianjian Chen, and Qiang Yang. Secure federated transfer learning. *arXiv preprint arXiv:1812.03337*, 2018.
- [Liu *et al.*, 2019] Ximeng Liu, Robert Deng, Kim-Kwang Raymond Choo, and Yang Yang. Privacy-preserving reinforcement learning design for patient-centric dynamic treatment regimes. *IEEE Transactions on Emerging Topics in Computing*, 2019.
- [Ma *et al.*, 2017] Yukun Ma, Lifang Wu, Xiaofeng Gu, Jiaoyu He, and Zhou Yang. A secure face-verification scheme based on homomorphic encryption and deep neural networks. *IEEE Access*, 5:16532–16538, 2017.
- [Mohassel and Zhang, 2017] Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 19–38. IEEE, 2017.
- [Montgomery, 1985] Peter L Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985.
- [Paillier, 1999] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. *theory and application of cryptographic techniques*, pages 223–238, 1999.
- [Putnam, 2014] Andrew Putnam. Large-scale reconfigurable computing in a microsoft datacenter. In *2014 IEEE Hot Chips 26 Symposium (HCS)*, pages 1–38. IEEE, 2014.
- [San and At, 2014] Ismail San and Nuray At. Improving the computational efficiency of modular operations for embedded systems. *Journal of Systems Architecture - Embedded Systems Design*, 60(5):440–451, 2014.
- [Song *et al.*, 2010] Bo Song, Kensuke Kawakami, Koji Nakano, and Yasuaki Ito. An RSA encryption hardware algorithm using a single DSP block and a single block RAM on the FPGA. In *First International Conference on Networking and Computing, ICNC 2010, Higashi Hiroshima, Japan, November 17-19, 2010. Proceedings*, pages 140–147, 2010.
- [Wright and Yang, 2004] Rebecca N Wright and Zhiqiang Yang. Privacy-preserving bayesian network structure computation on distributed heterogeneous data. pages 713–718, 2004.
- [Yang *et al.*, 2019] Qiang Yang, Yang Liu, Tianjian Chen, and Yongxin Tong. Federated machine learning: Concept and applications. *arXiv: Artificial Intelligence*, 2019.