High-Performance Hardware Acceleration Architecture for Cross-Silo Federated Learning

Junxue Zhang, Xiaodian Cheng, Liu Yang, Jinbin Hu, Han Tian, and Kai Chen

***Abstract*—Cross-silo federated learning (FL) adopts various cryptographic operations to preserve data privacy, which introduces significant performance overhead. In this paper, we identify nine widely-used cryptographic operations and design an efficient hardware architecture to accelerate them. However, directly offloading them on hardware statically leads to (1) inadequate hardware acceleration due to the limited resources allocated to each operation; (2) insufficient resource utilization, since different operations are used at different times. To address these challenges, we propose FLASH, a high-performance hardware acceleration architecture for cross-silo FL systems. At its heart, FLASH extracts two basic operators—modular exponentiation and multiplication—behind the nine cryptographic operations and implements them as highly-performant engines to achieve adequate acceleration. Furthermore, it leverages a dataflow scheduling scheme to dynamically compose different cryptographic operations based on these basic engines to obtain sufficient resource utilization. We have implemented a fully-functional FLASH prototype with Xilinx VU13P FPGA and integrated it with FATE, the most widely-adopted cross-silo FL framework. Experimental results show that, for the nine cryptographic operations, FLASH achieves up to $14.0\times$ and $3.4\times$ acceleration over CPU and GPU, translating to up to $6.8\times$ and $2.0\times$ speedup for realistic FL applications, respectively. We finally evaluate the FLASH design as an ASIC, and it achieves $23.6\times$ performance improvement upon the FPGA prototype.**

***Index Terms*—FPGA, federated learning, hardware accelerator.**

I. Introduction

TRAINING a high-quality machine learning model requires massive data, which is likely to be distributed across different institutions or companies in the real world. However, the increasing concern about data privacy and emerging regulations and lawsuits restrict these data from being collected together in one place for centralized training. To solve this problem, federated learning (FL) has been proposed to enable distributed learning among these *data silos* by performing local computation within a data silo and securely aggregating the intermediate results (e.g, gradients/parameters) to generate a global model without revealing any original data to the outside world [1], [2], [3].

To ensure the security of cross-silo FL, various cryptographic techniques have been used. For example, partially homomorphic encryptions (PHE), e.g, Paillier, have been used to enable parameter computation/aggregation directly on ciphertexts [4]. RSA is used to build the blind signature-based Private Set Intersections (PSI) for sample alignment [5]. In this paper, we perform a comprehensive analysis of existing cross-silo FL applications and identify nine widely-used cryptographic operations, such as encryption/decryption, computation over ciphertexts, etc. (more details in Section III-A). While preserving privacy, these cryptographic operations significantly degrade the performance (Section III-B). For example, our experiments show that these operations cause up to 60.74× performance degradation. The reasons are two-fold: (1) they are of high computational complexity, e.g., Paillier encryption has a $\mathcal{O}(2^N)$[1] time complexity; (2) they introduce large number calculations, e.g., additively HE and RSA encryption generate 2048-bit ciphertexts which need to be broken down to multiple 64-bit numbers and executed with limited parallelism on current CPU architecture.

In this paper, we ask: *can we offload these cryptographic operations to dedicated hardware to accelerate cross-silo FL?* Towards answering this question, our first attempt went with GPU. We have designed HAFLO [6], a GPU-based joint optimization of storage, IO, and computation for federated logistic regression. While there is still some room to optimize GPU-based solutions, in this paper, to achieve superb performance and low power consumption, we choose to use FPGA as a prototype and further explore an Application-specific Integrated Circuit (ASIC) to improve the acceleration of cross-silo FL. We believe such customized hardware architecture will exhibit several desired properties for our purpose. First, it is possible to tailor a hardware architecture for efficient cross-silo FL by customizing the hardware circuits from scratch, so that we can design an optimized fine-grained pipelining with flexible v bit-width support for accelerating cryptographic operations.

Manuscript received 3 May 2023; revised 9 June 2024; accepted 10 June 2024. Date of publication 13 June 2024; date of current version 1 July 2024. This work was supported in part by the Key-Area Research and Development Program of Guangdong Province under Grant 2021B0101400001, in part by the Hong Kong RGC TRS under Grant T41-603/20-R, in part by the GRF under Grant 16213621, in part by the ITF ACCESS, in part by the NSFC under Grant 62062005 and Grant 62102046, and in part by the Natural Science Foundation of Hunan Province under Grant 2022JJ30618. Recommended for acceptance by S. Song. *(Corresponding author: Kai Chen.)*

Junxue Zhang, Xiaodian Cheng, and Liu Yang are with the iSING Lab, Hong Kong University of Science and Technology, Hong Kong, and also with Clustar Technology Ltd., Hong Kong, China (e-mail: jzhangcs@connect.ust.hk; xchengaq@connect.ust.hk; lyangau@cse.ust.hk).

Jinbin Hu and Kai Chen are with iSING Lab, Hong Kong University of Science and Technology, Hong Kong (e-mail: jinbinhu@ust.hk; kaichen@cse.ust.hk).

Han Tian is with the University of Science and Technology of China, Hefei 230026, China (e-mail: henrytian@ustc.edu.cn).

This article has supplementary downloadable material available at https://doi.org/10.1109/TPDS.2024.3413718, provided by the authors.

Digital Object Identifier 10.1109/TPDS.2024.3413718

[1] N is the bit-width of the exponent n, and n is the public key in Paillier encryption.

Second, the customized hardware architecture allows us to provide sufficient on-chip memory for storing large numbers used in the processing pipeline for superior performance. However, while promising, we identify that directly offloading the nine cryptographic operations to the hardware statically will pose two key challenges (Section III-C):

- *Inadequate hardware acceleration due to limited resources:* To achieve high performance, one operation may need multiple hardware instances for high parallelism. However, as the hardware resource of a chip is limited, directly offloading all these nine operations to the hardware causes inadequate resources to speed up each operation, leading to suboptimal performance. Our implementation with this approach on a Xilinx VU13P FPGA [7] chip shows that each operation only achieves 50% acceleration on average.
- *Insufficient resource utilization due to static offloading:* Different FL applications use different cryptographic operations, and within each application, different operations are used at different times. Consequently, statically offloading *all* the operations as a whole results in resource underutilization because not all the operations are used at all times simultaneously.

To address the challenges, we take a closer look at these nine cryptographic operations and observe that almost all of them build upon two basic operators: *modular exponentiation and modular multiplication*. Based on this observation, we propose FLASH, a high-performance hardware acceleration architecture for cross-silo federated learning (Section IV). At its core, FLASH uses the majority of hardware resources to implement the two basic operators as high-performance engines to achieve adequate hardware acceleration. We also design fine-grained pipelines with sufficient on-chip memory to improve both the intra- and inter-engine execution efficiency for superior performance. Furthermore, based on these basic engines, FLASH adopts a dataflow scheduling module to dynamically compose these engines into different cryptographic operations on-demand to achieve high resource utilization.

We have provided a down-scale but full functional implementation of FLASH with Xilinx VU13P FPGA[2] [7] for prototyping purpose, integrated it with FATE [8]—the most widely-adopted cross-silo FL framework—and evaluated it extensively with real-world FL applications. We compare the performance of FLASH with (1) the vanilla FATE, which uses GMP [9] to implement these cryptographic operations with CPU. GMP provides a highly-optimized implementation for modular multiplication and exponentiation operations, which uses many optimization algorithms, including but not limited to the two mentioned in our paper. We choose Intel Xeon Silver 4114 CPU similar to prior works [10]; (2) the FATE where the cryptographic operations are accelerated by NVIDIA P4 GPU[3] [6], [13]. Here we use P4 GPU because it has the closest INT8 TOPS (although 2× better) as FLASH (we typically use INT8 TOPS to denote the general computation power of a chip). P4 has approximately 20 INT8 TOPS [13]. VU13P has 38.3 INT8 DSP TOPS while reaching peak 891 MHz operation frequency [7], [14]. As FLASH uses 300 MHz, it achieves 12.9 DSP INT8 TOPS. Moreover, both of them are built with 16 nm technology. Finally, with the standard Synopsys software tools (e.g., Design Compiler [15], VCS [16] and Prime Time [17]), we further evaluate the performance of FLASH if implemented as an ASIC.

Moreover, to evaluate how FLASH performs on the public cloud, we also migrate FLASH's design to Alibaba Cloud f3 instance [18], which has two VU9P FPGA chips (Section VI).

Overall, some of our key results are as follows:

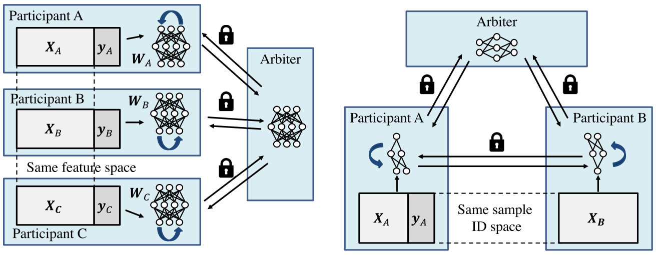
- Across the nine concrete cryptographic operations (Section VII-B), FLASH (with FPGA implementation) outperforms CPU and GPU by 10.4×–14.0× and 1.4×–3.4×, respectively.
- Over the nine realistic FL applications (Section VII-C), FLASH (with FPGA implementation) can consistently outperform CPU and GPU by up to 6.8× and 2.0×, respectively.
- FLASH on the public cloud (Section VII-B) can achieve comparable performance as its FPGA prototype with less than 5% performance loss.
- Our evaluation of FLASH as an ASIC with 12 nm and 28 nm fabrication techniques (Section VII-E) shows that it can achieve 23.6× and 7.1× *additional* performance improvement upon the FPGA implementation, respectively.

As a final note, we are fully aware that there exist various other privacy-preserving techniques [19], [20], [21], [22]. However, in current industry-level deployments, Paillier and RSA schemes built with the nine cryptographic operations we investigated in this work are, to date, the most widely adopted approach in cross-silo FL systems [8], [23], [24], primarily due to the reason that they can achieve relatively better performance and are easier to use compared to other privacy-preserving schemes. Our goal is to provide plug-and-play acceleration capability for these industry-level cross-silo FL systems.

This journal version is an extension to our previous work on FLASH [25] and offers additional technical details regarding its architecture and implementation. We emphasize several key enhancements: (1) We have conducted comparisons among various privacy-preserving technologies to elucidate why PHE stands out as the most commonly utilized privacy-preserving technology in cross-silo FL (Section II-A). (2) A comprehensive overview of FLASH's FPGA implementation has been provided, focusing on the physical arrangement of FLASH modules (Section V). (3) New content has been added to assess FLASH's compatibility with leading cloud providers, with the migration of FLASH's design to Alibaba Cloud highlighted (Section VI). (4) More in-depth experiments on FLASH have been included (Section VII-D).

[2] We use FPGA for prototyping purposes, so we do not consider the price advantages/disadvantages of the VU13P FPGA chip.

[3] We note that latest NVIDIA A100 [11]/H100 [12] may have a better performance than P4/VU13P. However, they utilize a much more advanced technology node. Readers can consult the results in Section VII-E to see how FLASH performs if also implemented as an ASIC with advanced technology node, such as 12 nm.



(a) Horizontal FL (b) Vertical FL

Fig. 1. Two paradigms of cross-silo FL.

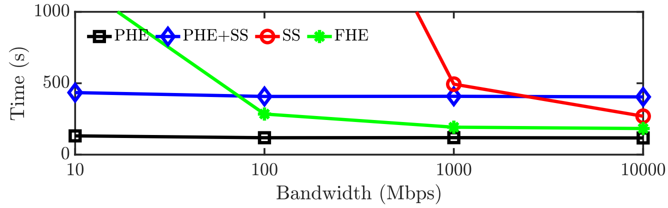


Fig. 2. VLR of 4 different privacy-preserving technologies under varying networking bandwidth (both PHE and FHE are in non-batched mode).

II. Cross-Silo Federated Learning

FL was first proposed by Google to train a language model for keyboard input prediction from massive Android devices without leaking privacy-sensitive data [26], [27]. Recently, FL has evolved from the above cross-device scenarios to collaboratively train machine learning models across different data silos, i.e., cross-silo FL [1], [2], [3]. A data silo is a repository or collection of data under the control of a single entity (e.g., institution, company, etc.), and is isolated from other entities due to the ever-improving management regulations or laws [28]. Cross-silo FL enables machine learning among these data silos and supports both vertical and horizontal FL.

Horizontal FL: As shown in Fig. 1(a), participants in Horizontal FL have different sample spaces but the same feature space, and each participant owns the labels of its samples. In most cases, there is an arbiter for parameter aggregation (the arbiter is a third-party server to assist the FL computations). To train a model, each participant trains local model $\boldsymbol{w}$ with its own samples and encrypts its model weights via PHE (e.g., Paillier [29]). Then all participants send their encrypted weights to the arbiter, and the arbiter directly performs an aggregation over the received ciphertexts to obtain the global model. Eventually, the arbiter sends the aggregated global model to all participants for next-round computation.

Vertical FL: As shown in Fig. 1(b), participants in Vertical FL have the same sample space but different feature spaces. Only one participant holds the label of the FL task. Before training a model, all participants have to align the samples among different data silos based on the common IDs (similar to joining two tables in a database based on the common IDs). One of the most commonly used algorithms is RSA blind signature-based PSI (RSA-PSI) [5]. After sample alignment, all participants can follow a pre-defined protocol for model training, such as Vertical Linear Regression (Please see Table 1 in [1]) and SecureBoost [30]. During the process, participants use PHE to encrypt their intermediate results and exchange them with other participants or the arbiter.

TABLE I
The Ratio of Networking Communication Time to the Total Computation Time

Bandwidth	10Mbps	100Mbps	1000Mbps	10000Mbps
PHE	3.56%	0.39%	0.04%	<0.01%
PHE+SS	1.41%	0.15%	0.01%	<0.01%
FHE	85.11%	36.37%	5.41%	0.57%

For interested readers, we have provided a detailed explanation on how cross-silo FL works and its security analysis in Appendix 1 of the supplementary materials.

A. Privacy-Preserving Technologies in Cross-Silo Federated Learning

Various privacy-preserving technologies can be used in cross-silo FL/machine learning, e.g., PHE [1], [30], secret sharing (SS) [20], fully homomorphic encryption (FHE) [21][4] or combination of PHE and SS [23]. In this section, we explain why PHE, or PHE-related privacy-preserving technology is still the widely-adopted one in real-world applications.

Please note, differential privacy (DP) [19], which is widely adopted in cross-device FL [26], is rarely used in cross-silo FL. The reason is that DP preserves privacy by adding noises to the data, thus it causes inevitable accuracy loss. Such accuracy loss is not acceptable in most cross-silo FL applications, e.g., financial risk controls.

Moreover, cross-silo FL in real-world scenarios usually suffers from limited networking bandwidth, e.g., 10 Mbps to 100 Mbps. Not all of the aforementioned privacy-preserving technologies deliver ideal performance under such an environment. To help readers to better understand it, we will use testbed experiments to illustrate their end-to-end performance. We employ 2 participants cooperatively train a linear regression model. During the training, we use `netem` [31] to vary the available bandwidth. The experiment results are shown in Fig. 2. From the results, we observe that SS and FHE suffer from dramatic performance degradation when the networking bandwidth is limited, especially within 10 Mbps and 100 Mbps. In contrast, the performance of PHE and PHE+SS is less affected by the varying bandwidth.

To investigate the root cause of the above experiments, we conducted further analysis on three privacy-preserving technologies. In particular, we focused on comparing three technologies, excluding SS, which is already known to be networking-intensive. The results of this analysis are presented in Table I. Based on the findings, the networking communication time for the technologies related to PHE is relatively small, accounting for less than 5% of the total computation time in all test cases. In contrast, the networking communication time for the FHE-driven linear regression model is significantly higher, reaching

[4] Only the model parameters, not the original data, are encrypted with FHE in FL/machine learning.

85.1% and 36.7% in scenarios with 10 Mbps and 100 Mbps networking bandwidth, respectively. These bandwidth settings represent typical scenarios in real-world cross-silo FL applications. Consequently, the high sensitivity of FHE to networking bandwidth renders its impractical for real-world cross-silo FL applications.

Therefore, among these privacy-preserving technologies, PHE and traditional (mainly used in RSA-PSI) cryptographic technics are widely used in cross-silo FL and deployed in real-world scenarios due to their practicality, which is the target of this paper. These cryptographic technics are composed of various operations, e.g., data encryption/decryption via PHE, computation over ciphertext, etc., and we call these operations as *cryptographic operations* in this paper.

III. Analysis of Cryptographic Operations

A. Cryptographic Operations

In this section, we present nine cryptographic operations that are widely used in cross-silo FL. Our study is based on the implementation of FATE [8], the most widely adopted open-source framework for cross-silo FL. However, our analysis can also be applied to other cross-silo FL frameworks, e.g., FedLearner [33], TF Encrypted [34], etc. Specifically, these nine cryptographic operations are as follows:

O1. Paillier Encryption: This operation uses Paillier [4], [35], an additively homomorphic cryptographic algorithm, to encrypt cleartexts into ciphertexts. The operation is mainly used for protecting the intermediate data during model training.

O2. Paillier Decryption: This operation decrypts Paillier ciphertexts into cleartexts. It is used when participants need to decrypt the intermediate results for local model updates in the training phase.

O3. Ciphertext Matrix Addition: This operation is used to add two matrices (or vectors/values) of ciphertexts. As Paillier is used, ciphertexts can be summed up.

O4. Ciphertext & Cleartext Matrix Element-wise Multiplication: This operation performs Hadamard product [36] between two matrics of ciphertexts and cleartexts.

O5. Ciphertext & Cleartext Matrix Multiplication:[5] This operation performs matrix multiplication between two matrices of ciphertexts and cleartexts, respectively.

O6. Ciphertext Histogram Building: This operation performs addition operations over encrypted gradient statistics to build decision trees [30].

O7. RSA Encryption/Decryption: This operation conducts encryption or decryption with the public or private key of the RSA algorithm correspondingly. This operation is used when multiple participants try to perform PSI for sample alignment [5].

O8. RSA Blind: This operation blinds the cleartexts with encrypted random numbers.

O9. RSA Unblind: This operation unblinds RSA ciphertexts to remove the random numbers from the ciphertexts.

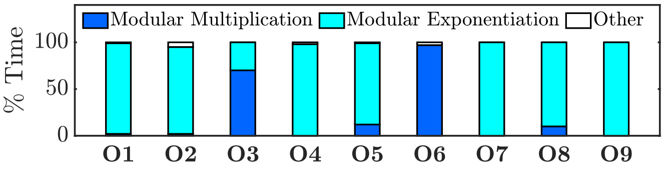


Fig. 3. Cryptographic operation computation time analysis.

As shown subsequently (Section III-B), these cryptographic operations have a large impact on the performance of cross-silo FL applications due to the following two reasons:

- *High time complexity:* These operations are of high computation complexity, e.g., Paillier encryption has a time complexity of $\mathcal{O}(2^N)$. Thus these algorithms are expensive to compute.
- *Large number computation:* Cryptographic operations significantly inflate data, yielding large numbers, e.g., 2048-bit integer. The large number will need to be divided into multiple small numbers and executed on the current CPU architecture with limited parallelism.

B. Quantifying the Performance Impact

We now quantify the performance impact of these cryptographic operations with realistic cross-silo FL applications through testbed experiments.

Testbed Setup: We use two X86 servers in our setup. Each server is equipped with a Mellanox CX-4 NIC [37] and connected to a Mellanox SN2100 [38] switch via 40 Gbps DAC cables. To reflect realistic networking situations in real-world cross-silo FL deployments, we use `netem` [31] to limit the networking bandwidth to 50 Mbps. As to other hardware configurations, each server is equipped with one Intel Xeon Silver 4114 CPU [39], 192 GB memory. We deploy FATE v1.5 as the cross-silo FL framework.

We choose three most widely-adopted vertical FL applications and one horizontal FL application for evaluation: RSA blind signature-based PSI (RSA-PSI), Vertical Logistic Regression (VLR) [32], SecureBoost Decision Tree (SBT) [30] and Horizontal Logistic Regression (HLR). The dataset we use is a commercial dataset from a bank with around 100,000 samples and 80 features. For vertical FL applications, the dataset is vertically partitioned into two parts: one part contains 80 features while the other contains one feature. We first perform RSA-PSI to obtain the data intersection. Then, we run VLR and SBT over the data intersection, respectively. For horizontal FL, the dataset is horizontally partitioned into two parts, each with 50,000 samples. The four applications are executed both with cryptographic operations implemented using GMP (w/ CO) and without cryptographic operations (w/o CO). To implement model training w/o CO, we modify the code of FATE to skip these cryptographic operations. To perform a fine-grained analysis, we also break down these four applications into sub-tasks, and for each sub-task, we show the adopted cryptographic operations. All the applications are executed with ten CPU cores in

[5] To efficiently process a large matrix, we will use optimization algorithms such as blocking the matrix and performing multiplications of the blocked matrices. Thus this operation is not a simple combination of matrix element-wise multiplication (**O4**) and addition (**O3**).

TABLE II
PERFORMANCE PENALTY CAUSED BY CRYPTOGRAPHIC OPERATIONS (CO) WITH DIFFERENT CROSS-SILO FL APPLICATIONS

Applications & Their Sub-tasks		Involved Operations	w/o CO (s)	w CO (s)	Degradation
RSA-PSI	Computing intersection	**O7, O8, O9**	7.91	20.62	$2.60\times\downarrow$
VLR (One Epoch) Total: $12.05\times\downarrow$	Encrypting logits	**O1**	0	32.05	-
	Aggregating logits	**O3**	0.63	2.04	$3.23\times\downarrow$
	Computing fore gradients[a]	**O3, O4**	0.74	2.92	$3.93\times\downarrow$
	Computing gradients	**O3, O4, O5**	3.77	135.96	$36.08\times\downarrow$
	Decrypting gradients	**O2**	0	0.04	-
	Computing loss	**O1, O3, O4**	4.08	8.22	$2.02\times\downarrow$
SBT (One Epoch) Total: $3.49\times\downarrow$[b]	Encrypting gradients	**O1**	0	54.71	-
	Aggregating gradients	**O3, O6**	12.02	223.91	$18.62\times\downarrow$
	Split information synchronization	**O2**	3.62	13.03	$3.60\times\downarrow$
HLR (One Epoch) Total: $60.74\times\downarrow$[b]	Computing gradients	**O3, O4, O5**	1.73	177.94	$102.80\times\downarrow$
	Model update	**O3, O4**	0.0002	0.10	$526.10\times\downarrow$
	Model re-encryption	**O1, O2**	0	0.69	-

[a] According to Federated Logistic Regression [32], the gradient computation takes two steps: fore gradients computation and gradients computation.
[b] The overall performance degradation of SBT/HLR is smaller than the sum of those sub-tasks because we do not include pure cleartext computation or networking communication sub-tasks in the table.

parallel. Table II shows the results, and we make the following observations:

- *Cryptographic operations considerably degrade the performance:* In general, cryptographic operations significantly degrade the performance of cross-silo FL applications. In our experiment, the cryptographic operations cause RSA-PSI, VLR, SBT and HLR to suffer $2.60\times$, $12.05\times$, $3.49\times$ and $60.74\times$ performance degradation, respectively. Moreover, the combinations of these cryptographic operations can degrade the performance from $2.02\times$ to $526.10\times$.
- *Not all the cryptographic operations are used at all times simultaneously:* Different FL applications use different cryptographic operations, and even within a single application, different sub-tasks use different operations.

C. Challenges of Offloading Cryptographic Operations

There are several available methodologies to accelerate these cryptographic operations. We have explored leveraging GPUs as our first attempt. Interested readers could consult our previous research paper, HAFLO [6], where we designed a GPU-based joint optimization of storage, IO, and computation for federated logistic regression. While we acknowledge that there are still some more spaces to explore with GPU to accelerate cryptographic operations, in this paper, we mainly focus on designing an ASIC to achieve superb performance and low power consumption (more discussions are in Section VII-E).

Therefore, we follow the rule-of-thumb approach to use FPGA as a prototype and evaluate the potential of ASIC via software tools [40], [41], [42]. However, we confront the following two challenges in our design:

1) Inadequate hardware acceleration due to limited resources: As identified in Section III-B, all the cryptographic operations cause a performance penalty, so we should offload all of them to hardware. Furthermore, to realize sufficient acceleration, each operation requires multiple hardware instances of accelerating modules/circuits for high parallelism. However, in practice, the hardware chip has limited resources, and if we naïvely offload all cryptographic operations to the chip, each operation has inadequate resources to be fully accelerated. Taking the DSP resources as an example, our preliminary implementation on VU13P FPGA [7] chip shows that to accelerate Paillier encryption (**O1**) by $2\times$, we need to use 2630 DSPs. Yet, a high-end FPGA chip, such as VU13P [7], only has 12288 DSPs, leaving < 1365 DSPs for one operation on average[6] (some DSPs are reserved for PCIe, memory controller, etc.). Thus, directly offloading *all* operations on VU13P FPGA chip leads to less than 50% acceleration on average. A similar problem also applies to the ASIC design.

2) Insufficient resource utilization due to static offloading. Different from software, hardware function is static after being configured/programmed/taped out, thus it cannot change its function dynamically. Nevertheless, as shown in Section III-B, not all the cryptographic operations are used at all times simultaneously. Consequently, if we statically offload all cryptographic operations on the hardware chip, only part of these cryptographic operations is used at a time. Therefore, such static offloading causes low resource utilization and further leads to suboptimal performance.

D. Opportunities With Accelerating Basic Operators

To overcome the above challenges, we further take a look at the internal of these nine cryptographic operations. We discover that *all* these operations are composed of two basic operators: modular multiplication and exponentiation. Then we further find that the performance of these operations is mainly decided by the two basic operators.

Paillier Encryption: Given the public key (n, g) and data m $(0 \leq m < n)$, the Paillier encryption algorithm takes two steps: (1) selecting a random number r where $0 < r < n$ and $r \in \mathbb{Z}_n^*$; (2) computing ciphertext $c = g^m \cdot r^n \mod n^2$. The formula can also be simplified to $c = (1 + mn) \cdot r^n \mod n^2$ by setting $g = (1 + n)$. We use $[[\cdot]]$ to denote the Paillier encryption, e.g., $c = [[m]]$.

Homomorphic Addition: Given two plaintext a and b, homomorphic addition guarantees that $[[a]] \bigoplus [[b]] = [[a + b]]$.

[6] We will show later that all these nine operations share similar building blocks, thus they require similar resources to implement.

In Paillier cryptosystem, $[[a]]\bigoplus[[b]]$ is defined as $[[a]]*[[b]] \mod n^2$. The homomorphic addition is used by operations **O3**, **O5** and **O6**.

Homomorphic Multiplication: Given a plaintext a and k, the homomorphic multiplication is denoted by $k\cdot[[a]]$. It can be actually regarded as a homomorphic addition: $\Sigma^k[[a]]$. Thus, $k\cdot[[a]]=[[a]]^k \mod n^2$. The homomorphic multiplication is used by operations **O4** and **O5**.

Paillier Decryption: Given the public key (n,g), private key (p,q) and ciphertext c, the Paillier Decryption algorithm can be optimized via Chinese Remainder Theorem (CRT) to reduce the original workload to only about one-quarter of the original decryption algorithm. To use CRT, we define L_p and L_q to be $L_p(x)=\frac{x-1}{p}$ and $L_q(x)=\frac{x-1}{q}$. The decryption algorithm takes the following three steps: (1) computing $m_p=L_p(c^{p-1} \mod p^2)L_p(g^{p-1} \mod p^2)^{-1} \mod p$; (2) computing $m_q=L_q(c^{q-1} \mod q^2)L_q(g^{q-1} \mod q^2)^{-1} \mod q$; and (3) computing plaintext $m=\mathrm{CRT}(m_p,m_q) \mod n$.

RSA-related Operations: The RSA-related operations are used in RSA blind signature-based PSI [5]. It is commonly known that the core of these RSA-related algorithms is either modular multiplication or modular exponentiation.

Through the above mathematical analysis, we find that *all* cryptographic operations used in cross-silo FL are built upon the two basic operators: modular multiplication and exponentiation. Then, we further perform testbed experiments to investigate how these two basic operators impact the performance of the nine original cryptographic operations. The results are shown in Fig. 3. Clearly, we find that across *all* nine original operations, the two basic operators occupy $>95\%$ of the total execution time.

Observation: We can compose all the nine cryptographic operations with these two basic operators, and by accelerating these two basic operators, all the nine original operations used in cross-silo FL applications can be effectively accelerated.

IV. The FLASH Design

Inspired by the above observation, we present FLASH, a high-performance hardware acceleration architecture for cross-silo FL. This section describes how we design FLASH in detail. Please note that our design has been fully implemented in our FLASH prototype with FPGAs as well as rigorously evaluated with the Synopsys tools for the ASIC.

A. Architecture Overview

Fig. 4 shows the overall architecture of FLASH. It contains a hardware acceleration card and an integrated software package. The accelerator card can be plugged into a server via PCIe Gen3×16 interface. The server is installed with cross-silo FL software, e.g., FATE. The software can invoke FLASH's software package to *offload* the cryptographic operations on the card for efficient acceleration.

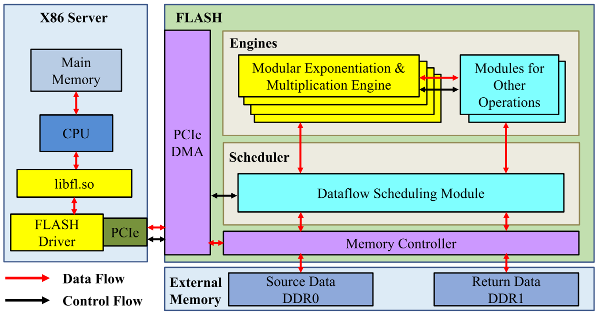


Fig. 4. FLASH architecture.

The idea of our FLASH design is that it does not directly offload all cryptographic operations on the hardware, but leverages the insights of our observation to (1) utilize the limited programmable resource for most performance-critical basic operators: modular exponentiation and multiplication to achieve *adequate acceleration* (Section IV-B), and (2) design an on-chip dataflow scheduling module to dynamically compose different cryptographic operations on-demand based on these engines, achieving *high resource utilization* (Section IV-C). In addition, to make FLASH a general solution to support a wide range of cross-silo FL frameworks, our software package provides standard APIs. In this way, different cross-silo FL software can utilize FLASH by harnessing its APIs (Section IV-D).

B. Modular Exponentiation and Multiplication Engines

To implement modular exponentiation and multiplication operators as high-performance engines on hardware, FLASH makes the following design decisions. First, instead of directly offloading the modular exponentiation and multiplication operators, we optimize the algorithms of the two operators to make them suitable for the hardware implementation (Section IV-B1). Second, based on the optimized algorithms, FLASH further leverages pipelining technologies to efficiently execute them with high parallelism (Section IV-B2). Third, we provide sufficient on-chip memory for pipeline execution and make the data transfer as part of the pipeline to efficiently exchange data between off-chip memory and engines (Section IV-B3).

1) Algorithm Optimization: The mathematical formulas of the two basic operators: modular exponentiation (1) and multiplication (2) are as follows:

$$P=m^e \mod n \qquad m,e,n\in\mathbb{Z}^+ \tag{1}$$

$$P=ab \mod n \qquad a,b,n\in\mathbb{Z}^+ \tag{2}$$

When used in cryptographic operations, all the numbers a,b,m,n are large numbers, leading to high computation complexity. Therefore, before designing FLASH's engines, we first apply some commonly-used optimization strategies in the cryptographic research community to optimize the two basic operators, including Binary Exponentiation [43] and Montgomery Modular Multiplication [44], etc. The advantages of using these optimization strategies are: (1) lowering the number of multiplications used in modular exponentiation from $\mathcal{O}(2^N)$ to $\mathcal{O}(N)$ (N is the bit-width of e), and (2) replacing the modulo operation with the hardware-friendly bit-shifting operation.

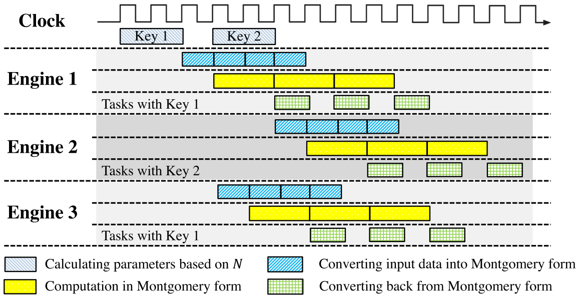

Clock
Key 1
Key 2
Engine 1
Tasks with Key 1
Engine 2
Tasks with Key 2
Engine 3
Tasks with Key 1
Calculating parameters based on N
Converting input data into Montgomery form
Computation in Montgomery form
Converting back from Montgomery form

Fig. 5. Pipeline executions of both inter- and intra-engines.

After applying these optimization methods, FLASH's modular exponentiation and multiplication operators require the following four stages for the computation:

S1) Preparing common data needed in Montgomery space based on the input data n. Since, in both Paillier and RSA cryptosystems, n is decided by the public key, we can re-use these prepared data for all computations with the same public key. This is common in cross-silo FL applications as they use one key for all cryptographic operations within one application.

S2) Performing input data pre-precessing and converting them into Montgomery space.

S3) Performing computation in Montgomery form. Major operations in this stage include large-number multiplication, addition, and bit-shifting. No modulo operation is needed.

S4) Converting all output data of the operators out of Montgomery space.

2) Pipelining: We next introduce how FLASH efficiently performs the above four computation stages via inter- and intra-engine pipelining.

Inter-engine pipelining: To enable inter-engine pipelining, FLASH employs an engine pipeline stage manager to control the execution strategies for different stages. Fig. 5 gives an overview of how these stages are pipelined. First, FLASH reserves S1 as a standalone stage, which can be executed in advance once it obtains the public key. Second, for all computation tasks with the same public key (Engine 1 and 3 in Fig. 5), they can be executed in parallel once their data preparation is completed (S1). The start time of these engines may have a small gap of several clock cycles because FLASH adopts a round-robin strategy to dispatch stage executions. Third, for tasks with different keys (Engine 2 in Fig. 5), they can be executed independently.

Intra-engine pipelining: FLASH further performs intra-engine pipelining within the most computation-intensive stage S3 to accelerate the stage's internal execution. The key design goals are: (1) FLASH should support variable bit-widths thus the application can choose the key length based on their security requirements; (2) the hardware resources should be fully utilized to achieve superb performance.

Algorithm 1: Montgomery Modular Multiplication.

```
▷ Given 3 input numbers $X$, $Y$ and $N$, the Montgomery Modular
Multiplication outputs $Z = X \cdot Y \cdot R^{-1} \mod N$, where $R$ is a power
of 2 and $\lfloor \log_2 R \rfloor = \lfloor \log_2 N \rfloor$.
▷ We use only two multipliers in our implementation: Mul 1 and Mul
2. Operations assigned to Mul 1 and Mul 2 are marked with red and
green respectively.
Input: $X = (X_{d-1}, ..., X_0)$, $Y = (Y_{d-1}, ..., Y_0)$, $N = (N_{d-1}, ..., N_0)$,
    $N'$, where $N' = (-N)^{-1} \mod r$,        ▷ $N'$ is pre-computed in S1
    $r = 2^w$, $d = \lfloor \log_r N \rfloor + 1$,        ▷ $r$ and $d$ is used to split data
    $\gcd(N, r) = 1$, with $N \times N' \equiv -1 \mod r$
Output: $Z = \text{ModMult}(X, Y, N) = X \times Y \times R^{-1} \mod N$
$Z = (Z_{d-1}, ..., Z_0) = 0$                          ▷ Initialization
for all $i = 0, 1, ..., d-1$ do                         ▷ Loop on $Y$
    $\alpha = [X_0 \times Y_i]_{\text{low}}$
    $\beta = \alpha + Z_0$
    $q = [\beta \times N']_{\text{low}}$
    $\delta_1 = \alpha + [q \times N_0]_{\text{low}}$
    $\delta_2 = \delta_1 + Z_0$
    $Z_0 = [\delta_2]_{\text{low}}$
    $C = [\delta_2]_{\text{high}}$
    for all $j = 1, 2, ..., d-1$ do                     ▷ Loop on $X$
        $\delta_0 = [X_{j-1} \times Y_i]_{\text{high}} + [q \times N_{j-1}]_{\text{high}}$
        $\delta_1 = \delta_0 + [X_j \times Y_i]_{\text{low}} + [q \times N_j]_{\text{low}}$
        $\delta_2 = \delta_1 + Z_j + C$
        $Z_{j-1} = [\delta_2]_{\text{low}}$
        $C = [\delta_2]_{\text{high}}$                  ▷ Carry higher bits
    end for
    $Z_{d-1} = C$
end for
if $Z \geq N$ then
    $Z = Z - N$
end if
return $Z$
```

To achieve the first goal, FLASH builds an efficient pipeline that processes data based on radix-2^w arithmetic [45]. We use $w = 64$ in FLASH's implementation for the following three reasons: 1) The Xilinx Multiplier IP [46] for FPGA supports inputs ranging from 1 to 64 bits wide. 2) When our multiplication is implemented with the 64-bit basic multipliers, it reaches the optimal performance per resource (DSP) compared to other choices. 3) The key length used in cross-silo FL is larger than 64 b-width, thus $w = 64$ should support all required cases. Given any input data, we split it into d w-bit integers. For example, $d = 16$ when bit-width of X is 1024, and $d = 32$ when bit-width of X is 2048. Theoretically, the pipeline can be adapted for input data with any bit-width as long as the bit-width is or can be extended by complementary zeros to the integer multiple of d. After data splitting, the complete algorithm of S3 (Montgomery Modular Multiplication) is shown in Algorithm 1. Note, compared to the original Montgomery Modular Multiplication (shown in our supplementary materials), we make the following optimizations to make the algorithm more hardware-friendly: (1) the computation of S (i.e., line 6 of Algorithm 4.2 in our supplementary materials) is separated into computations of lower w bits and higher w bits so that the bit-width required in operations (e.g., addition) is halved; (2) the first iteration of the inner loop where $j = 0$ is unrolled to remove the conditionals in the original algorithm (i.e., line 7 to 9 of Algorithm 4.2 in the supplementary materials) and keep the consistency of computation logic (equivalence are discussed in Section IV.C of supplementary materials).

In Algorithm 1, the most computation-intensive operations are the multiplications of $X_j \times Y_i$ and $q \times N_j$ respectively (both operations require d^2 w-bit multiplications). Moreover, the data

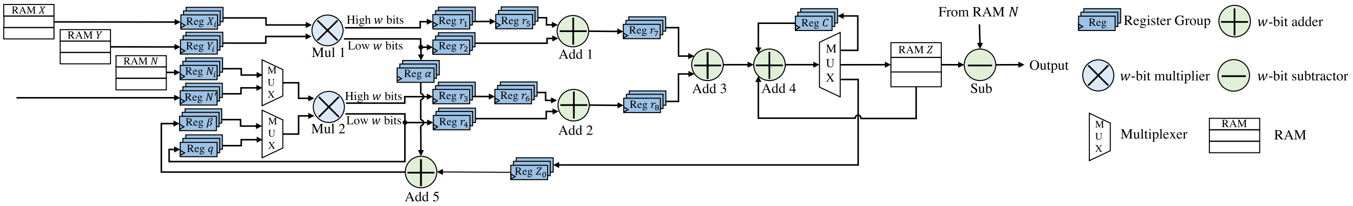


Fig. 6. The Montgomery Modular Multiplication engine circuit design. Our circuit uses two multipliers and four on-chip RAMs for efficient pipelining.

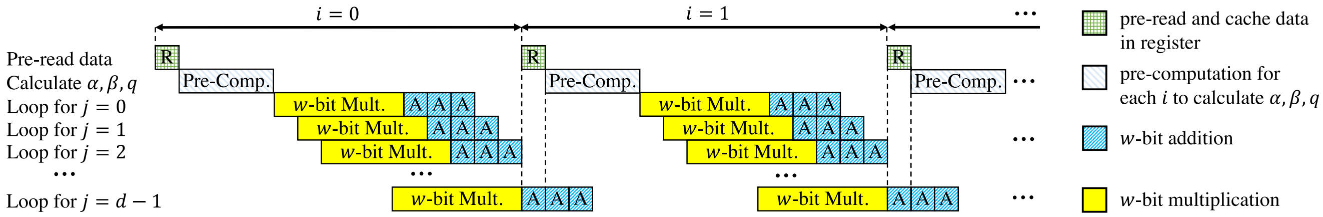


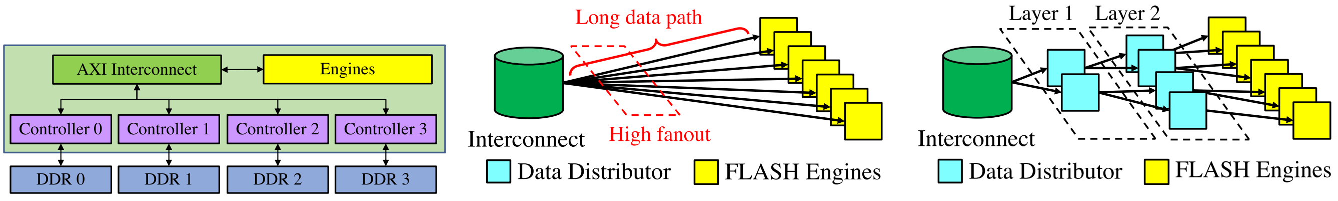
Fig. 7. Efficient pipelining of Montgomery Modular Multiplication.

required in these two multiplications are totally independent. Therefore, we use two multipliers (one 64-bit multiplier consists of 32 DSP48E2 slices [47] on our FPGA prototype), Mul 1 and Mul 2, for these two multiplication operations and reuse them to execute the rest of the multiplication operations as well (operations assigned to Mul 1 and Mul 2 are marked with red and green respectively in Algorithm 1). Since the multiplier can continuously process data, to fully utilize the multiplier, we have the following design decisions. First, we design a circuit to fully pipeline the inner loop (line 10 to 16 of Algorithm 1). The circuit is shown in Fig. 6 and we will introduce it later. Second, when the multiplications of i-th iteration finish and some other operations are still under execution, e.g., addition operations in the right part of the Fig. 6, FLASH allows direct starting the multiplications in $i+1$-th iteration to minimize the delay between different iterations. We also use Fig. 7 to visualize how the operations in S3 are efficiently pipelined.

Below we show how the circuit in Fig. 6 works. Since the outer loop iterates over operand Y, the circuit sequentially reads different Y_i from RAM Y and performs execution over them. The workflow of our Montgomery Modular Multiplication circuit contains four steps. Steps 1 and 2 in the following introduction focus on the workflow for a fixed Y_i while steps 3 and 4 show how we bridge the operations between iterations with different i and obtain the final result. We use Reg to represent the register group.

1) X_0 and Y_i are sent to Mul 1 to get a $2w$-bit multiplication result. The higher w bits of the result are cached in Reg r_1. The lower w bits, denoted as α in Algorithm 1 in the main text, are cached in both Reg α and r_7. (The Add 1 is bypassed at step 1 as we are using the circuit to calculate α, and the inner loop of Algorithm 1 hasn't started. Therefore, we temporarily use r_7 to store α without cache conflicting through r_1 to r_7). The numbers stored in Reg α are used for the calculation of β via Add 5. With β and N' available, their multiplication result q can be obtained from the output of Mul 2. After that, q is sent back to the input of Mul 2 and multiplied with N_0. Similarly, the higher w bits of the multiplication result are cached in Reg r_3 and the lower w bits are sent to Reg r_8. Combining data cached in r_7 and r_8, it is straightforward to get δ_1 and δ_2 with several addition units. The results of addition, Z_0 and C, are cached in Reg Z_0 and Reg C for subsequent operations.
2) Following step 1, the inner loop for j begins. Different iterations of the inner loop are fully pipelined, which implies the jth iteration is executed by the circuit just one cycle after the $(j-1)$th iteration. At the beginning of the pipeline, Mul 1 and Mul 2 simultaneously calculate $X_j \times Y_i$ and $q \times N_j$. The higher w bits and lower w bits of the results are separately cached in different registers. Please note that we use Reg r_5 and r_6 to register the higher w bits in the circuit for one more cycle compared to the lower w bits so that δ_1 can be calculated through the addition between higher w bits from the $(j-1)$th iteration and lower w bits from the jth iteration (i.e., line 12 in Algorithm 1 in the main text). After the calculation of δ_1, the subsequent calculation of δ_2 can also be simply executed by the adders in the pipeline. The intermediate results, Z_{j-1} and C, are cached in RAM Z and Reg C.
3) We begin the execution of the $(i+1)$th iteration of the outer loop when all the multiplications in the ith iteration accomplish. Although some operations like addition are still in progress, the multipliers are free to start the multiplications in Step 1 for the $(i+1)$th iteration.
4) After accomplishing the outer loop, the result Z should be stored in RAM Z. If $Z < N$, we directly output Z. Otherwise, we calculate $Z - N$ as the final output.

3) On-Chip & Off-Chip Memory: To provide sufficient on-chip memory for efficient pipeline execution, FLASH allocates four memory units for each modular multiplication and exponentiation engine (shown in Fig. 6). For our FPGA prototype implementation, we use 4× 36Kbit BRAM (Block RAM) units.



(a) AXI Interconnect.

(b) Naïve data distribution.

(c) Hierarchical data distribution.

Fig. 8. FLASH adopts hierarchical data distribution to enable efficient data exchange between on-chip and off-chip memory.

We use the 36Kbit BRAM as it is the smallest true dual-port BRAM module available in UltraScale+ FPGA that supports simultaneous read and write operations [48]. This size is sufficient for our needs as we only store one single ciphertext in each BRAM at any given time. While the on-chip memory is mainly used for pipeline execution, FLASH further exploits external memory (shown in Fig. 4) for input, output and intermediate data storage. To achieve high performance, data exchange between on-chip and off-chip memory is part of the pipeline itself. Specifically, FLASH primarily utilizes a sequential access pattern to retrieve batches of input data. During the computation of a particular batch, FLASH can proactively fetch data from the subsequent batch, so that the data fetching time can overlap with the computation time. Since FLASH can determine the addresses of all batches in advance, it eliminates the need for additional hardware design, such as a pre-fetcher, thereby avoiding any associated costs or overhead. As the data fetch time is typically shorter than the computation time, it effectively hides the off-chip memory access latency, leading to perfect pipelining.

As shown in Fig. 8(a), FLASH adopts AXI interconnect to manipulate the external memory [49], [50]. However, as the on-chip memory units are placed near each engine for low latency, naïvely distributing data from the AXI interconnect to these memory units, as shown in Fig. 8(b), leads to high fan-out near interconnect and long data paths. These two issues will cause (1) large design difficulties for circuits placement because there are too many long paths to be placed near interconnect; (2) degraded performance because long paths cause large delay for the circuits.

To solve the problem, we design a hierarchical data distribution mechanism as shown in Fig. 8(c). Instead of directly sending data to all engines, FLASH distributes data at multiple layers. At each layer, the data distributors receive data from the previous layer and further distribute data to the data distributors/engines in the next layer. Suppose we have m engines and n layers, the fan-out of each data distributor is approximately $\log_n m$, which is much smaller than m. As a result, FLASH achieves a much smaller fan-out and shortened logical data path. These two advantages first reduce the design complexity because a small fan-out with short logical paths will make the circuits' placement much easier. Furthermore, they also improve performance because they allow a high operating frequency by restricting the delay of all logic paths. In our FPGA implementation, the delay of all logic data paths is within 3.3 ns, thus we can achieve a high FPGA operation frequency of 300 MHz [51].

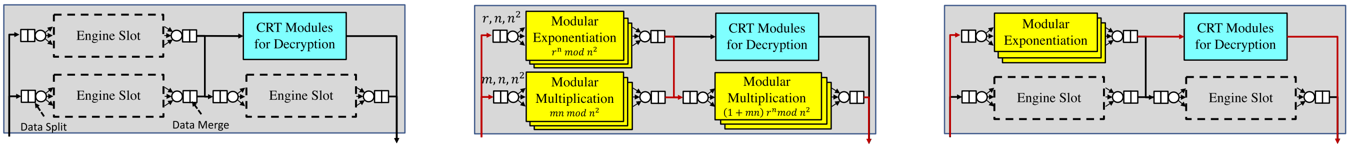
Moreover, the hierarchical data distribution module is designed to be fully pipelined, ensuring no additional overhead in terms of throughput. Although the hierarchical data distribution mechanism introduces some latency due to the processing required by each module, it is important to note that in throughput-sensitive scenarios such as data exchange, this mechanism does not introduce notable overhead to the overall end-to-end performance.

C. Dataflow Scheduling

We now introduce how FLASH composes various cryptographic operations over basic engines through dataflow scheduling. First, we show how our engines can work at different modes (Section IV-C1). Then, we present how different cryptographic operations are constructed by combining particular engines (Section IV-C2).

1) Dynamic Engine Switching: To build various cryptographic operations over basic engines, FLASH needs to enable dynamic engine switching between modular exponentiation and multiplication. Mathematically, modular exponentiation can be realized by performing modular multiplication multiple times. Thus, FLASH leverages a hardware control module to achieve it without reconfiguring hardware (it is almost impossible to reconfigure the ASIC). Specifically, to accelerate modular exponentiation, FLASH constructs a dataflow loop over the multiplication engine multiple times. In contrast, when the engine needs to execute modular multiplication, FLASH directs the dataflow through the modular multiplication engine once. While the design works well for most cryptographic operations that use either modular exponentiation or multiplication, it cannot directly support operations that simultaneously require both modular exponentiation and multiplication, e.g., Paillier encryption (**O1**), matrix multiplication (**O5**), in which FLASH has to decide the ratio of engines in different modes.

How to decide the ratio? We use domain knowledge in cross-silo FL applications to decide the ratio. Taking matrix multiplication operation (**O5**) as an example, it first performs ciphertexts and cleartexts multiplication (requires modular exponentiation) and then ciphertexts addition (requires modular multiplication). Considering the modular exponentiation, the exponent e is a cleartext, which has a common bit-width of 64. As mentioned in Section IV-B1, since we use Binary Exponentiation to optimize the modular exponentiation, the number of modular multiplication required may vary from 64 to 127 depending on the specific value of cleartext. On average, 96 modular multiplications are



(a) All dataflow scheduling paths. (b) Dataflow for encryption. (c) Dataflow for decryption.

Fig. 9. Dataflow scheduling. Black arrow indicates all available paths for dataflow scheduling while red arrow indicates the active paths for a particular cryptographic operation. Each engine slot can have multiple engines.

required. Thus, the throughput of the modular exponentiation should be around $1/96$ of modular multiplication. Based on this, we can adjust the ratio of the engines working in different modes to make the throughput of both modular exponentiation and modular multiplication balanced. In this way, the hardware resources can be efficiently utilized and no engines will sit idle.

2) Building Cryptographic Operations: As shown in Fig. 9(a), the core idea of dataflow scheduling is to use an on-chip controller to dynamically determine: (1) which data paths (they are logical paths that do not reflect the physical wiring) should be active, and (2) what to put in the engine slots, based on which operation is offloaded. Each engine slot contains one data splitting module and one data merging module to distribute data to different engines and aggregate results from these engines, respectively. The data splitting and merging modules are equipped with physical wires connecting them to all engines, establishing a comprehensive network where each engine is linked to all splitting and merging modules. Therefore, by selectively activating specific wires, we can precisely allocate engines to their respective engine slots, thereby determining the exact number of engines accommodated in each slot. We also design a hierarchical data distribution mechanism, as mentioned in Section IV-B3, for better performance. Specifically, we can construct a Paillier encryption operator by following the dataflow scheduling strategy shown in Fig. 9(b). As mentioned in Section III-D, the Paillier encryption follows equation: $c = (1 + mn) \cdot r^n \mod n^2$. So we can distribute the data m, n, n^2 to modular multiplication engines (these engines are denoted E_1) to calculate $r_1 = mn \mod n^2$ and distribute the data r, n, n^2 to modular exponentiation engines (these engines are denoted E_2) to calculate $r_2 = r^n \mod n^2$. Then the results can be further sent to modular multiplication engine (these engines are denoted E_3) to calculate $(1 + r_1) \times r_2 \mod n^2$. Please note that the $1 + r_1$ is completed in the input data pre-precessing stage (S2 in Section IV-B1) with a lightweight dedicated hardware module. The ratios of E_1, E_2 and E_3 are determined through the strategies discussed above, thus we can assign particular numbers of engines to these engine slots. Similarly, Fig. 9(c) shows the dataflow used in Paillier decryption. In this case, FLASH uses other modules besides modular exponentiation and multiplication engines to realize the decryption operation. In particular, we use CRT to optimize the decryption algorithm as discussed in Section III-D, thus FLASH implements several CRT modules to accelerate this operation. We put all engines, working in modular exponentiation mode, in the top left corner engine slot to achieve high resource utilization.

Engine Interconnection: Every single engine of FLASH can support modular multiplication and exponentiation individually. Since most cryptographic operations require element-wise operations, they do not require engine interconnection. Thus, FLASH uses data splitting and merging modules mentioned above to achieve these operations. For some particular operations, such as **O5**, they require data aggregation after some operations, such as sum operation, we use tree-like aggregator rather than complex interconnection to achieve it. The aggregator can be part of the inter-engine pipeline to achieve high efficiency.

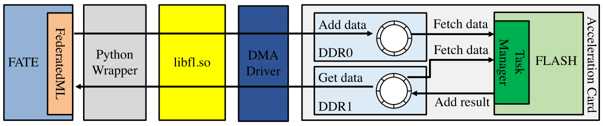


Fig. 10. FLASH integrates with cross-silo FL frameworks by providing an integrated software package.

```
import flash_np as np
# Generating two Paillier-encrypted arrays accelerated by
    FLASH
x1 = np.array([1, 2, 3], encryption="paillier")
x2 = np.array([4, 5, 6], encryption="paillier")

x3 = x1 + x2 # Homomorphic addition
x4 = np.array([1, 2, 3], encryption=None)
x5 = x4 * x1 # Ciphertext & cleartext multiplication

x3.decrypt() # Decrypting the ciphertext
x5.decrypt()

x3.get() # Transferring the data from accelerator to host
x5.get()
```

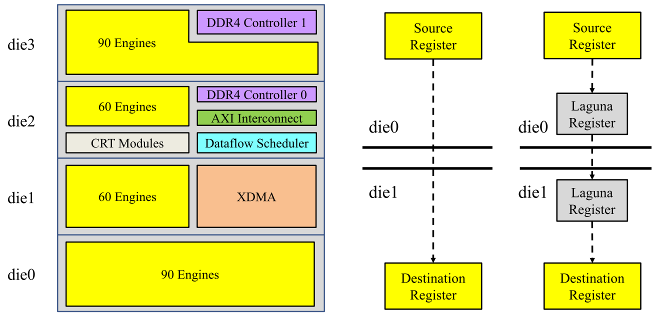
Listing 1: FLASH's NumPy-like APIs.

D. Software Integration

Fig. 10 illustrates how FLASH integrates with the cross-silo FL software. While our current implementation integrates FLASH with FATE, the design of FLASH is generic and works with other cross-silo FL systems/frameworks. They can harness FLASH by using the standard APIs.

For easy integration, FLASH provides Python NumPy-similar APIs as shown in Listing 1. The Python APIs are also wrappers of the C/C++ library: `libfl.so`. Besides providing standard APIs, the `libfl.so` library manipulates the status of all installed FLASH accelerators, such as temperature, workload, etc.

By using these APIs, users can easily create encrypted scalar, vector, or matrix via Paillier or RSA encryption method. Users can further perform homomorphic addition and multiplication operations over these data. To reduce the data exchange between the FLASH accelerator and the host, we put the computation



(a) Placement of FLASH modules.

(b) Using Laguna register to optimize latency.

Fig. 11. Implementation details of FLASH.

results on the off-chip memory unless the `get` API is used. As shown in Section IV-B3, the data exchange between on-chip and off-chip memory is efficiently pipelined, leading to better end-to-end performance. Moreover, since `libfl.so` works in a stateless way, it can be easily scaled out to support different tasks from various FL applications.

Multi-accelerator Support: The server-side software also enables multi-accelerator support. If there are multiple FLASH accelerators on the server, when applications invoke the APIs, `libfl.so` will break the task into multiple sub-tasks and dispatch them to multiple accelerators. The dispatching strategy is least workload first, and can be configured to use different strategies, such as round-robin.

V. Implementation

Prototype Implementation with FPGA: We implement FLASH with FPGA using approximately 30,000 lines of Verilog [52] code. We use Xilinx Virtex UltraScale+ VU13P chip [7] in our implementation. FLASH implements 300 modular exponentiation and multiplication engines with the chip.

We now introduce how we place these engines and other modules of FLASH on VU13P chip in detail. The programmable resources of VU13P chip are distributed into 4 vertically-stacked dies (also named SLR – Super Logic Region [53]) with similar sizes. These dies are named `die0`, `die1`, `die2` and `die3` from bottom to top, as Fig. 11(a) indicates. The wiring resources reside only between neighboring dies, such as `die0`-to-`die1`, `die1`-to-`die2`, etc. Wiring between non-neighboring dies involves multiple wirings between neighboring dies. Since FLASH contains many different modules, e.g., 300 modular exponentiation and multiplication engines, data scheduling control module, etc, properly arranging these modules on the four dies is crucial to FLASH's overall performance. The goal is two-fold: 1) the resource utilization of every single die is high; 2) the wirings among dies should be as fewer as possible.

To achieve the goal, the core idea is to first place single modules which require large resource consumption to a specific die, then distribute these 300 engines to fulfill the leftover resources of each die, as shown in Fig. 11(a). The detailed procedure has the following steps. (1) Determine the placement of the Xilinx DMA (XDMA) module [54]. Since the PCIe interface of VU13P resides on the `die1`, we place the XDMA module on the same die to reduce the wiring complexity. (2) Determine the placement of the DDR4 controllers. The VU13P chip has 4 DDR4 controllers on the 4 dies in total, which could be further connected to 4 16 GB DDR4 RAMs. In FLASH's design, we use 2 16 GB DDR4 RAMs, thus we place the two DDR4 controllers on `die2` and `die3` respectively. Since DDR4 controller and XDMA are both large modules, such placement could better balance resource utilization. (3) Determine the placement of the AXI interconnect module. The AXI interconnect is connected with the XDMA module, the 2 DDR controllers and the engines, which plays an important role in caching data on the external memory (Section IV-D) and further distributing data from external memory to the engines. The AXI interconnect module is placed on the `die2`. The advantage of such placement is that the communication between XDMA and AXI interconnect module, or between DDR4 controller and AXI interconnect module, is occurred in the same die or between neighboring dies, thus the wiring complexity is reduced. (4) Determine the placement of other control modules. For example, the dataflow scheduler is placed on the `die2`. We place it on the same die of AXI interconnect since data will flow from interconnect to the scheduler first, then to the engines. The CRT modules used for decryption (Section IV-C2) are also placed on the `die2` since the decrypted data will be sent back to the scheduler as the final step. (5) Determine the placement of engines. After placing these large modules, we will distribute the 300 modular exponentiation and multiplication engines into 4 dies. As the Fig. 11(a) indicates, there are 90, 60, 60, 90 engines on `die0`, `die1`, `die2`, `die3`, respectively.

As mentioned in Section IV-B3, FLASH adopts a hierarchical data distribution mechanism to distribute from external memory to massive engines. Considering specific implementation with VU13P chip, we take the following approaches. First, we establish 4 wires from the AXI interconnect to each die (since AXI interconnect is on `die2`, 3 wires are across dies) as the first data distribution layer. Second, we further adopt hierarchical data distribution mechanisms within each die. For `die0` and `die3`, we further construct two data distribution layers, with 1-to-5 and 5-to-18 distributors (please see Section IV-B3 for a more detailed description of distributors) on each layer, respectively. Thus, it could support $5 \times 18 = 90$ engines. For `die1` and `die2`, we also construct two data distribution layers, with 1-to-6 and 6-to-10 distributors, respectively.

After placing all modules, we finally perform optimizations over cross-die wires. The idea is leveraging Laguna registers [55]. Laguna registers are special registers designed for cross-die connections and reside at the edge of each die. As shown in Fig. 11(b), we add two registers in the cross-die wire, and set constraints to map the two registers to Laguna registers during placement. The advantage is: (1) the cross-die wire is shortened (the wire between the two registers), and (2) the Laguna registers could further optimize the communication latency [55].

Table III demonstrates the resource utilization of FLASH in detail. As a final note, the operation frequency of our FPGA implementation is 300 MHz while we achieved 88% DSP resource

TABLE III
RESOURCE CONSUMPTION OF THE FPGA PROTOTYPE

Name	Used	Total	Ratio
LUT	1094K	1728K	63%
Registers	1766K	3456K	51%
DSP	10756	12288	88%
RAM	324Mb	455Mb	71%
IO	282	676	42%
Serdes	16	76	21%

TABLE IV
FPGA TYPES PROVIDED BY THE MAJOR CLOUD VENDORS

Cloud Vendor	Instance	FPGA Type
Amazon AWS	F1 [58]	Virtex UltraScale+ VU9P
Microsoft Azure	NP [59]	Virtex UltraScale+ VU13P
Alibaba Cloud	f3 [18]	Virtex UltraScale+ VU9P
Huawei Cloud	fp1 [60]	Virtex UltraScale+ VU9P

utilization, which, to the best of our knowledge, is relatively high in FPGA's industry [56].

Server-side Software Stack Implementation: Our implementation of FLASH's server-side software contains around 10,000 lines of C/C++ and Python code. This includes modifications of FATE to harness FLASH's acceleration capacity. We mainly modify the `federatedml` module [57] in FATE by replacing normal collection operations with FLASH's NumPy-like APIs. We further use XDMA IP Reference driver [54] for high-performance direct memory access through the PCIe interface.

Evaluating FLASH as ASIC: We leverage multiple standard softwares to assess the FLASH design as an ASIC. Specifically, we first use Synopsys Design Compiler [15] to convert FLASH's design logics into physical implementations, i.e., netlist, over both 12 nm and 28 nm technology libraries. Then, we use Synopsys VCS [16] to verify that the generated netlist functions correctly and use Synopsys Prime Time [17] for static timing analysis to validate that the netlist satisfies all timing constraints. More evaluation results of the ASIC performance will be discussed in Section VII-E.

VI. FLASH ON CLOUD

Recently, people are moving cross-silo federated learning to the cloud. Thus, in this section, we are discussing whether FLASH's design can be directly used on the cloud or what changes we should make to make our design feasible on the cloud.

FPGA types provided by major cloud vendors: First, to investigate whether FLASH's design can be directly used on the cloud, we have surveyed the FPGA types provided by four cloud vendors: Amazon AWS, Microsoft Azure, Alibaba Cloud and Huawei Cloud. These vendors have provided the virtual machine instances that provide FPGA as a service. Table IV summarizes the FPGA types provided by these cloud vendors and we have the following observations:

- Microsoft Azure NP instance [59] provides Xilinx U250 acceleration card [61] with Xilinx Virtex UltraScale+ VU13P FPGA chip [7], which is identical to our in-house FPGA implementation (Section V). Therefore, we can directly deploy FLASH in the Microsoft Azure Cloud.
- Other cloud vendors, including Amazon AWS F1 instance, Alibaba Cloud f3 instance and Huawei Cloud fp1 instance, provide several VU9P FPGAs [18], [58], [60]. Since VU9P has around half programmable resources of VU13P chip, we cannot directly apply FLASH's design to VU9P. However, considering the popularity of VU9P chip on the public cloud, we decide to migrate FLASH's FPGA design to match that of a VU9P FPGA chip. This migration is feasible since VU9P and VU13P share similar fundamental components.

In the following sections, we will introduce how we migrate FLASH's design on Alibaba Cloud f3 instances which have two VU9P FPGA chips. We will evaluate the performance of FLASH on cloud with its FPGA prototype in Section VII-B.

FLASH on Alibaba Cloud: For management purpose, cloud vendors reserve a particular area on the FPGA chip for fixed functions, which users cannot modify. Specifically, Alibaba Cloud adopts a shell + role architecture, which is shown in the online document [18]. The XDMA modules of the two VU9P chips are placed in the static shell region. Moreover, one DDR controller of each VU9P is also in the shell region due to constrains from the vendor. The other modules will be put in the role region, which can be configured by the user. Since the static shell region occupies a certain chip area that cannot be modified, the timing constraints of FPGA on the cloud are harder to meet. Therefore, we put 148 engines on each VU13P provided by Alibaba Cloud f3 instance, reducing the total number of FLASH engines from 300 to 296.

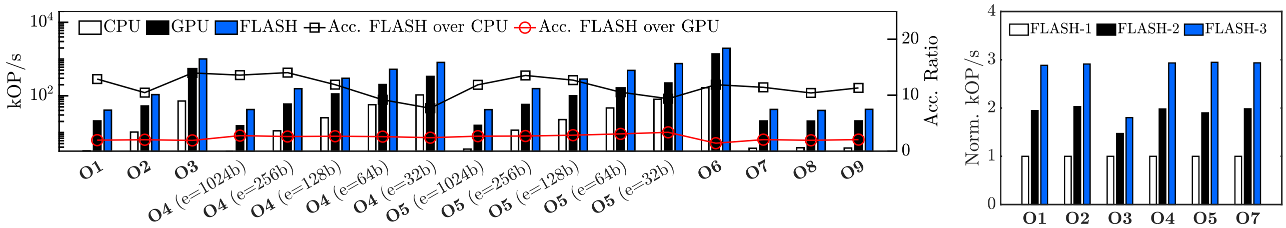
VII. EVALUATION

In this section, we first present our evaluation methodology (Section VII-A). Then we show that for the nine cryptographic operations, FLASH achieves up to 14.0× and 3.4× acceleration over CPU and GPU (Section VII-B), translating up to 6.8× and 2.0× speedup for realistic FL applications (Section VII-C), respectively. Finally, we evaluate the performance of FLASH as an ASIC (Section VII-E).

A. Methodology

Environment Setup: We utilize the identical testbed mentioned in Section III-B for consistency. Additionally, each server is equipped with a single FLASH acceleration card. For our multi-accelerator experiment, multiple acceleration cards are installed on each server. Furthermore, in the multi-participant experiment, we will employ up to five of these servers.

Schemes Compared: We mainly compare the performance achieved by FLASH with that achieved by: (1) Original FATE that uses a highly-optimized GMP library to execute cryptographic operations with CPU (denoted as CPU in the following charts). We choose Intel Xeon Silver 4114 CPU similar to prior works [10]. All CPU experiments are executed with all ten physical cores in parallel. (2) GPU-based accelerator (denoted as GPU). We extend the GPU implementation of HAFLO [6] which only implements logistic regression. Note that only the



(a) Cryptographic operation performance of all compared schemes. The left Y-axis is associated with the bar chart and is in the log scale. The right Y-axis is associated with the line chart.

(b) Performance with multiple accelerators.

Fig. 12. Performance of cryptographic operations.

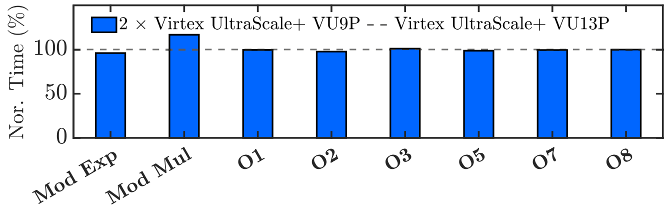


Fig. 13. Evaluation results of FLASH on Alibaba Cloud with two Virtex UltraScale+ VU9P FPGAs.

cryptographic operations are accelerated by GPU in our experiments. We use NVIDIA P4 GPU because it has the same technology of 16 nm and achieves the closest INT8 TOPS as FLASH (although 2× better. P4 reaches around 20 INT8 TOPS while FLASH achieves 12.9 INT8 TOPS).

Performance Metrics: We use *the number of operations performed per second (OP/s)* as the metric when evaluating the performance of cryptographic operations, and *acceleration ratio over CPU/GPU* as the metric when evaluating FL applications.

B. *Cryptographic Operations*

To demonstrate that FLASH can efficiently accelerate the nine cryptographic operations, we compare the performance achieved by CPU, GPU, and FLASH, respectively. For operations **O4** and **O5**, we also evaluate different exponent bitwidths (32bit – 1024bit). The experiment results are shown in Fig. 12(a). In general, FLASH can consistently outperform CPU and GPU for all cryptographic operations. Specifically, FLASH outperforms CPU by 7.7× – 14.0× and GPU by 1.4× – 3.4×, showing that FLASH's hardware architecture fits the computational requirements of these cryptographic operations. Furthermore, we observe that when handling a larger exponent, FLASH tends to achieve a better acceleration ratio. For example, FLASH achieves 13.6× acceleration than CPU when evaluating O4 with $e = 1024$bit, but drops to 7.7× with $e = 32$bit. The results show that when the computation is more intensive, i.e., with a large exponent, FLASH can achieve even better performance.

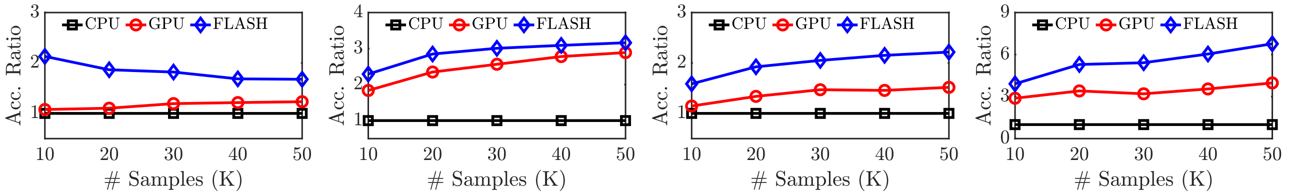
Multi-accelerator Support: We inspect how FLASH performs when we use multiple FLASH acceleration cards to speed up cryptographic operations. We evaluate one, two and three accelerators, denoted as FLASH-1, FLASH-2 and FLASH-3 respectively. For space limitation, we only pick some operations for demonstration: **O1**, **O2**, **O3**, **O4** with $e = 1024$bit, **O5** with $e = 1024$bit, and **O7**. The results are shown in Fig. 12(b). We observe that for most cryptographic operations, e.g., **O1**, **O2**, **O4**, **O5** and **O7**, the overall performance of FLASH is almost linear to the number of accelerators: FLASH-2 achieves 1.90× – 1.98× while FLASH-3 achieves 2.89× – 2.95× speedup for these operations. However, for **O3**, FLASH-2 and FLASH-3 only achieve 1.47× and 1.80× acceleration, respectively. The reason is as follows: the computation workload of **O3** is relatively low, thus the control overhead, e.g., multi-accelerator synchronization, takes a considerable portion, leading to non-linear speedup. However, in the real-world use case, we envision that FLASH with multiple accelerators would still be an efficient solution to accelerate large-scale cross-silo FL applications.

Comparison with Other Paillier Accelerators: To give readers a better understanding of how efficient the FLASH's hardware design is, we further compare FLASH with some state-of-the-art hardware-based solutions, e.g., Paillier Cryptoprocessor (PCP) [62], HLS [64], and SoC [65] based solutions. Moreover, due to the limited hardware resources, some of these works only implement a subset of cryptographic operations supported by FLASH. The comparison results are shown in Table V. PCP and HLS report their data with public key $N = 1024$ bit, while SoC uses $N = 2048$ bit, thus we report the performance of FLASH with both $N = 1024$ and 2048 bit. The results show that, compared to PCP, HLS and SoC, FLASH consumes 10.97×, 1.80×, 4.88× DSP resources, but delivers 28.95×, 7.77×, and 10.75× encryption acceleration and 93.15×, 20.56×, and 34.38× decryption acceleration, respectively. The results demonstrate that by using inter- & intra-engine pipelining and dataflow scheduling, FLASH can (1) deliver much better performance if utilizing comparable resources, and (2) support more complete functions.

Performance of FLASH on Alibaba Cloud: We use f3 instance with 2 Xilinx Virtex UltraScale+ VU9P FPGAs. The server is equipped with 32 vCPU (Intel Xeon Platinum 8163 Skylake), 128 GB RAM and 200G ESSD. Fig. 14 demonstrates the results and we have normalized the performance of FLASH on Alibaba Cloud to the performance of FLASH's FPGA prototype for better visualization. We have observed that for most operations, FLASH on Alibaba Cloud achieves comparable performance as its FPGA prototype with a performance degradation of less than 5%. An interesting finding is that the modular multiplication with FLASH on cloud is even faster than its local FPGA (16% better). The observed 16% impact on performance can be attributed to the following factors: 1) individual modular multiplication operation is highly efficient, resulting in I/O time

TABLE V
RESOURCE CONSUMPTION & PERFORMANCE COMPARISON AMONG FLASH AND OTHER PAILLIER ACCELERATORS

	FPGA	Logic Cells	DSP	Public Key $N = 1024$bit		Public Key $N = 2048$bit	
				Encryption (kOP/s)	**Decryption** (kOP/s)	**Encryption** (kOP/s)	**Decryption** (kOP/s)
FLASH	VU13P	3,780,000	12,288	40.706	107.707	6.033	19.373
PCP [62]	7VX330T [63]	326,400	1,120	1.40625	1.15625	-	-
HLS [64]	VU9P [7]	2,586,000	6,840	5.238	5.238	-	-
SoC [65]	ZU9EG [66]	600,000	2,520	-	-	0.561	0.563



(a) RSA-PSI (b) VLR (c) SBT (d) HLR

Fig. 14. Performance of RSA-PSI, VLR, SBT, and HLR with changing data volumes.

TABLE VI
MODELS & DATASETS USED IN EVALUATION OF FLASH

	Models	Datasets
Vertical FL	RSA-PSI [5] VLR [32] SBT [30]	CreditCard [67]
Horizontal FL	HLR	CreditCard [67]
	MLP	FMNIST [68]
	LSTM [69]	Shakespeare [70]
	DenseNet169 [71]	
	ResNet50 [73]	Cifar-10 [72]
	VGG16 [74]	

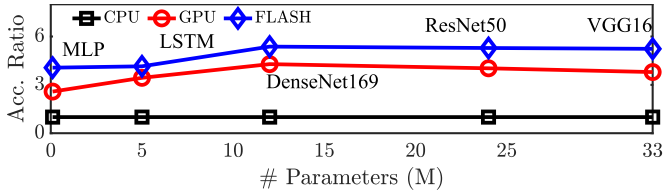


Fig. 15. Performance of five deep learning applications.

accounting for approximately 70% of the overall execution time; 2) the Alibaba Cloud utilizes a superior SSD, leading to a significant reduction in I/O time.

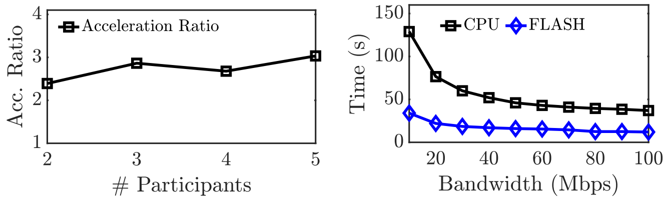
C. Cross-Silo FL Applications

We then present how FLASH can accelerate real-world cross-silo FL applications, including both vertical and horizontal. The models and datasets used are shown in Table VI. For vertical FL, before performing the model training algorithms, we first run a commonly used sample alignment algorithm: RSA blind signature-based PSI (RSA-PSI). Then, we perform Vertical Logistic Regression (VLR) [32] and Secure Boosting Tree (SBT) [30] algorithms over the data intersection (generated from PSI), respectively. For horizontal FL, we mainly evaluate Horizontal Logistic Regression (HLR) and five deep learning applications with different parameters. Each application runs a fixed number of epochs.

RSA-PSI, VLR, SBT, and HLR: The performance of RSA-PSI, VLR, SBT, and HLR is related to the data volumes. Thus we evaluate FLASH with different data volumes. The results are shown in Fig. 15. In general, FLASH consistently outperforms CPU and GPU by achieving $1.6\times - 6.8\times$ and $1.1\times - 2.0\times$ acceleration ratio respectively. The results have demonstrated that by designing a tailored hardware acceleration architecture for cross-silo FL, we can effectively speed up FL applications and outperform the existing CPU/GPU architectures. Furthermore, we also notice that for RSA-PSI and VLR, GPU tends to reach a similar acceleration ratio as FLASH while processing more data. The reason is that for RSA-PSI and VLR, the cleartext computation, which is purely executed on the CPU, takes a significant portion of the total computation time. For example, in VLR, when handling 50K data samples in one epoch, after sufficient acceleration, the ciphertext computation takes $< 10\%$ of the total computation time. Therefore, the performance is mainly decided by the time of cleartext computation when the cryptographic operations are sufficiently accelerated, which leads to the results that FLASH and GPU achieve similar acceleration ratios over CPU. In contrast, for HLR and SBT, FLASH can achieve a higher acceleration ratio than GPU because the cryptographic operations of these two applications consume a significant portion of the total computation time.

Deep Learning Applications: We have further evaluated five deep learning models of different numbers of parameters with horizontal FL. The results are shown in Fig. 15. We find that FLASH can outperform CPU and GPU by achieving $4.1\times - 5.4\times$ and $1.2\times - 1.6\times$ acceleration ratio respectively due to a similar reason discussed above. Furthermore, we note that for models with more parameters, e.g., DenseNet169, ResNet50, VGG16, FLASH can achieve a higher speedup than models with fewer parameters, e.g., MLP, LSTM. This experiment implies that for more computation-intensive tasks, FLASH can deliver more notable results.

Performance on Alibaba Cloud: We also evaluate the performance of cross-silo FL applications with FLASH on Alibaba



(a) Performance of VLR with varying participants.

(b) Performance of VLR with varying bandwidth.

Fig. 16. FLASH deep-dive.

Cloud. The results obtained were nearly identical to our local evaluations, with a variance of less than 5%.

Correctness: In addition to evaluating the performance of the above nine cross-silo FL applications, we also validate the final results of all compared schemes (we avoid the randomness by setting an identical random seed). Results have shown that all schemes yield identical results, showing that FLASH does not affect the correctness of model training.

Summary: Implemented as an FPGA prototype, FLASH has already largely outperformed CPU and achieved moderately better performance than GPU with comparable price. We also understand that high-end GPUs, e.g., A100 [11], H100 [12], may outperform FLASH's FPGA prototype due to more advanced foundry technology, which are also of much higher price. However, they still share the drawbacks as mentioned in Section III-C. The goal of our paper is to design a more efficient hardware acceleration architecture for cross-silo FL beyond existing CPU/GPU architectures. As we will demonstrate in Section VII-E, if implemented as an ASIC, the performance of FLASH can be significantly improved, which should boost the acceleration ratio for these applications to a much higher level.

D. FLASH Deep-Dive

In this part, we mainly investigate the followings:

1) How does the number of participants affect the performance of FLASH?
2) How does the varying network bandwidth affect the performance of FLASH?
3) What's the performance breakdown of FLASH?
4) How can FLASH cooperate with other cross-silo FL optimization mechanisms?

Number of Participants: We evaluate VLR with two to five participants and measure the acceleration ratio of FLASH over CPU. The experiment result is shown in Fig. 16(a) and we observe that in general, the number of participants does not largely impact the acceleration of FLASH.

Varying Bandwidth Setting: In this part, we use `netem` [31] to limit the available bandwidth between the two participants from 10 Mbps to 100 Mbps. We run VLR and measure the execution time of one iteration with both CPU and FLASH. Fig. 16(b) shows the results and we can observe that when the bandwidth is over 50 Mbps, the running times of both CPU and FLASH are stable, where FLASH outperforms CPU by around 3×. The results show that the varying network bandwidth does not have a noticeable impact on FLASH.

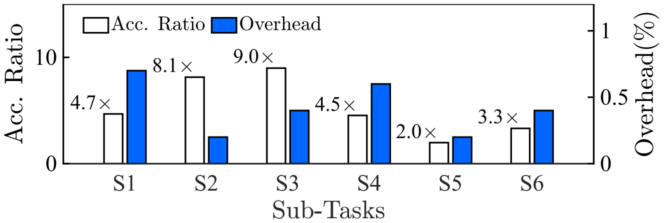
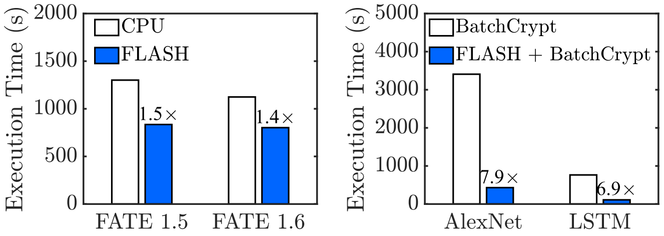


Fig. 17. FLASH can consistently accelerate VLR with small dataflow scheduling overhead.



(a) SecureBoost.

(b) BatchCrypt.

Fig. 18. FLASH can further accelerate other cross-silo FL optimization strategies.

Performance Breakdown: We now dig deep to investigate how FLASH can accelerate different sub-tasks within one application. We also investigate the overhead of the dataflow scheduling mechanism when offloading different cryptographic operations in different sub-tasks. We mainly show the percentage of overhead over the total execution time. We use VLR as an example here. We also follow Table II to break the application into 6 sub-tasks, i.e., S1: encrypting logits to S6: computing loss. The results are shown in Fig. 17 and we have the following two results.

First, we observe that FLASH can achieve 2.0× – 9.0× acceleration regarding different sub-tasks. Towards the sub-task with the lowest acceleration ratio: S5, we find that the execution time of this task is relatively short, thus the software overhead takes a significant portion, leading to a relatively low acceleration ratio. Second, the overhead of switching among different cryptographic operations via dataflow scheduling is within 1% of the total execution time, which can be ignored. It confirms that by designing the dataflow scheduling mechanism on FPGA, FLASH can efficiently react to applications' diverse cryptographic operations usage patterns with small overhead.

Cooperation with Other Optimization Mechanisms: We evaluate how FLASH works when cooperating with other cross-silo FL optimization mechanisms. First, we will show FLASH's performance towards evolving algorithm. To show it, we evaluate FLASH with SecureBoost from both FATE v1.5 and FATE v1.6. The SecureBoost in FATE v1.6 has a more efficient decision tree building strategy [75], thus it yields better performance than SecureBoost in FATE v1.5. The evaluation results are shown in Fig. 18(a). We can observe that the SecureBoost in FATE v1.6 achieves 14% better performance than the one in FATE v1.5. Nevertheless, FLASH can still deliver a comparable acceleration ratio for the improved version of SecureBoost (1.4× v.s. 1.5×) although SecureBoost in FATE v1.6 has a shorter

TABLE VII
ASIC Resource Evaluation for Both 28 nm and 12 nm Technology Libraries

	28nm Technology Library (Actual Op. Frequency: 800MHz)			**12nm Technology Library** (Actual Op. Frequency: 1120MHz)		
	Area/Unit (mm^2)	**# Unit**	**Total Area (mm^2)**	**Area/Unit (mm^2)**	**# Unit**	**Total Area (mm^2)**
PCIe Gen3×16	8.46	1	8.460 (6.56%)	5.25	1	5.250 (4.04%)
DDR4 Controller	7.25	2	14.500 (11.24%)	4.43	2	8.860 (6.81%)
Engine Logic	0.093	800	74.480 (57.72%)	0.046	1900	87.499 (67.26%)
Engine Memory	0.033	800	26.200 (20.30%)	0.014	1900	25.927 (19.93%)
Dataflow Scheduling & Others	5.399	1	5.399 (4.18%)	2.561	1	2.561 (1.97%)
Total	-	-	129.04 (99.26%)	-	-	130.10 (100.08%)

time of cleartext computation, causing the computation time of cryptographic operations over total execution time to decrease. Second, we will evaluate whether FLASH can further accelerate one state-of-the-art cross-silo FL optimization mechanism: BatchCrypt [76]. BatchCrypt accelerates FL by packing a batch of integers into one integer for Paillier encryption to reduce the cost of Paillier encryption. The experiment results are in Fig. 18(b). We show that FLASH can further accelerate BatchCrypt by up to 7.9×, which can achieve more than 20× acceleration ratio compared to pure CPU execution.

In summary, the above experiments show that FLASH is a general and practical solution. FLASH can effectively accelerate cross-silo FL applications by accelerating these common cryptographic operations used in FL applications.

E. ASIC Performance Assessment

Given that our FPGA-based prototype implementation of FLASH has performance limitations due to the intrinsic drawback of FPGA (e.g., low operation frequency), in this section we intend to demonstrate some preliminary results of how FLASH performs as an ASIC. As introduced in Section V, we use standard software tools to assess the performance of FLASH if implemented as an ASIC. We evaluate FLASH's ASIC implementation with two technology libraries: 28 nm and 12 nm. Based on the industry experience, we set the operating frequency to be 1000 MHz and 1400 MHz, respectively, for these two technology libraries. Furthermore, we set the die area to be around 130 mm^2. We believe this setting could balance the performance and power consumption for FLASH.

The detailed evaluation includes the following steps: First, we perform logic synthesis using Synopsys Design Compiler [15] to convert FLASH's design into netlist under the frequency and die area constraints. Table VII illustrates the results. With 28 nm technology library, we can allocate 800 modular multiplication and exponentiation engines successfully, while with 12 nm technology library, we can allocate 1900 such engines. Second, we use Synopsys VCS [16] and Synopsys Prime Time [17] to confirm that both netlists are valid and function correctly. The third step is to estimate the performance gain of FLASH as an ASIC. Since the actual operating frequency after physical design should be lower than logic synthesis, we reduce the actual operation frequency by multiplying 80% by the design target for a conservative purpose.

Then, our final performance estimation is as follows. With 28 nm technology library, we can allocate 2.67× engines compared to our FPGA implementation (800 v.s. 300), and the operation frequency of these engines is 2.67× that of the FPGA implementation (800 MHz v.s. 300 MHz), leading to an overall 7.11× performance gain on modular exponentiation operator (we use modular exponentiation operator as the metrics since it can fulfill the computation capacity of an engine). With 12 nm technology library, we can allocate 6.33× engines (1900 v.s. 300) with 3.73× operation frequency (1120 MHz v.s. 300 MHz), and achieve 23.64× overall performance improvement. To give our readers a better understanding of FLASH's performance as an ASIC, we also evaluate the modular exponentiation operator with a state-of-the-art GPU – NVIDIA A100 [11], our results show that A100 can only achieve 5.78× performance gain than our downscale FPGA prototype. Finally, we estimate the power consumption for a single engine, which is 16.6 mWatt with 28 nm technology library. Thus, the total power consumption for all engines is 13.28 Watt. Although we do not have the power consumption data of other parts, e.g., PCIe controller, we believe the total power consumption of FLASH as an ASIC should be significantly lower than the 120 Watt of our FPGA implementation.

Peak Memory Analysis: Existing hardware accelerators for FHE argue that their major design challenge is how to allocate adequate memory bandwidth for data movement [77], [78], [79]. However, FLASH does not require such high memory bandwidth because PHE and traditional cryptosystems do not inflate plaintext as much as FHE [80]. For example, when the public key $N = 1024$bit, as shown in Fig. 12(a), FLASH can reach a maximum performance of 1000 kOP/s for most cryptographic operations. While the input size has a maximum length of 4096 b, the peak memory bandwidth required in FLASH's FPGA prototype is 488 MB/s. While the ASIC has a 23.6× performance improvement, it does not exceed the usual memory bandwidth of DDR4, around 16 GB/s.

VIII. Related Works

Accelerating FL: Recently, due to the increasing deployment of FL, various research works have emerged to accelerate FL. MAGE proposes to optimize the secure computation from a memory perspective [81]. BatchCrypt tries to optimize the Paillier encryption by encoding a batch of quantized gradients into a long integer and encrypting it in one batch [76]. VF ^{2} Boost proposes a novel training protocol to reduce the idle time of each participant [24]. Relative to them, we design FLASH from a different angle: accelerating the cryptographic operations used in FL, and our FLASH could be easily combined with these prior works.

Domain Specific Accelerator (DSA): DSA has recently been an emerging research topic that adopts hardware, e.g., FPGA, ASIC, etc, to accelerate particular applications [10], [78], [79], [82], [83], [83], [84], [85], [86]. For example, Tiara [82] uses FPGA and a programmable switch to accelerate layer-4 load balancing. FlowBlaze [83] offloads complex networking functions to a NetFPGA SmartNIC. hXDP [84] proposes to use FPGA to accelerate eBPF programs for fast XDP execution. MicroRec [85] offloads neural networks to FPGA to implement efficient recommendation systems. POCLib provides a high-performance framework for enabling near orthogonal processing on compression [87]. Various DSAs have been proposed to accelerate fully homomorphic encryption (FHE) [88], such as HEAX [10], F1 [77], BTS [78] and CraterLake [79]. Similar to them, FLASH follows the principle of DSA to design a hardware-based solution to efficiently accelerate cross-silo FL.

IX. Conclusion

This paper presented FLASH, a hardware acceleration architecture for cross-silo FL. We have provided a fully functional FPGA prototype and evaluated our design as an ASIC. Extensive experiments with realistic applications and cryptographic operations have shown that FLASH is a viable solution.

References

[1] Q. Yang, Y. Liu, T. Chen, and Y. Tong, “Federated machine learning: Concept and applications,” *ACM Trans. Intell. Syst. Technol.*, vol. 10, no. 2, pp. 12:1–12:19, 2019.

[2] O. Fink, T. H. Netland, and S. Feuerriegel, “Artificial intelligence across company borders,” *Commun. ACM*, vol. 65, no. 1, pp. 34–36, 2022.

[3] Y. Cheng, Y. Liu, T. Chen, and Q. Yang, “Federated learning for privacy-preserving AI,” *Commun. ACM*, vol. 63, no. 12, pp. 33–36, 2020.

[4] P. Paillier, “Public-key cryptosystems based on composite degree residuosity classes,” in *Proc. Int. Conf. Theory Appl. Cryptographic Techn.*, Prague, Czech Republic, 1999, pp. 223–238.

[5] E. D. Cristofaro and G. Tsudik, “Practical private set intersection protocols with linear complexity,” in *Proc. Int. Conf. Financial Cryptography Data Secur.*, 2010, pp. 143–159.

[6] X. Cheng, W. Lu, X. Huang, S. Hu, and K. Chen, “HAFLO: GPU-based acceleration for federated logistic regression,” 2021, *arXiv:2107.13797*.

[7] “Xilinx UltraScale FPGA production table and production selection guide,” 2020. [Online]. Available: https://www.xilinx.com/content/dam/xilinx/support/documents/selection-guides/ultrascale-plus-fpga-product-selection-guide.pdf,

[8] “Federated AI Technology Enabler,” 2019. [Online]. Available: https://fate.fedai.org

[9] “The GNU multiple precision arithmetic library,” 2020. [Online]. Available: https://gmplib.org

[10] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, “HEAX: An architecture for computing on encrypted data,” in *Proc. Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Lausanne, Switzerland, 2020, 2020.

[11] “NVIDIA A100,” 2020. [Online]. Available: https://www.nvidia.com/en-us/data-center/a100/

[12] “NVIDIA H100,” 2022. [Online]. Available: https://www.nvidia.com/en-us/data-center/h100/

[13] “NVIDIA P4,” 2016. [Online]. Available: https://images.nvidia.com/content/pdf/tesla/184457-Tesla-P4-Datasheet-NV-Final-Letter-Web.pdf

[14] “Xilinx virtex UltraScale FPGA data sheet: DC and AC switching characteristics,” 2020. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ds923-virtex-ultrascale-plus

[15] “Synopsys design compiler,” 2022. [Online]. Available: https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html

[16] “Synopsys VCS,” 2022. [Online]. Available: https://www.synopsys.com/verification/simulation/vcs.html

[17] “Synopsys prime time,” 2022. [Online]. Available: https://www.synopsys.com/implementation-and-signoff/signoff/primetime.html

[18] “Alibaba cloud F3,” 2018. [Online]. Available: https://www.alibabacloud.com/blog/deep-dive-into-alibaba-cloud-f3-fpga-as-a-service-instances_594057

[19] M. Abadi et al., “Deep learning with differential privacy,” in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Vienna, Austria, 2016, pp. 308–318.

[20] P. Mohassel and Y. Zhang, “SecureML: A system for scalable privacy-preserving machine learning,” in *Proc. IEEE Symp. Secur. Privacy*, 2017, pp. 19–38, San Jose, CA, USA, 2017.

[21] S. Sav et al., “POSEIDON: Privacy-preserving federated neural network learning,” in *Proc. 28th Annu. Netw. Distrib. System Secur. Symp.*, 2021, pp. 1–18.

[22] H. Tian et al., “Sphinx: Enabling privacy-preserving online learning over the cloud,” in *Proc. 43rd IEEE Symp. Secur. Privacy*, San Francisco, CA, USA, 2022, pp. 2487–2501.

[23] C. Chen et al., “When homomorphic encryption marries secret sharing: Secure large-scale sparse logistic regression and applications in risk control,” in *Proc. 27th ACM SIGKDD Conf. Knowl. Discov. Data Mining*, Singapore, 2021, pp. 2652–2662.

[24] F. Fu et al., “VF mbox2Boost: Very fast vertical federated gradient boosting for cross-enterprise learning,” in *Proc. Int. Conf. Manage. Data*, China, 2021, pp. 563–576.

[25] J. Zhang, X. Cheng, W. Wang, L. Yang, J. Hu, and K. Chen, “FLASH: Towards a high-performance hardware acceleration architecture for cross-silo federated learning,” in *Proc. 20th USENIX Symp. Netw. Syst. Des. Implementation*, Boston, MA: USENIX Association, 2023, pp. 1057–1079. [Online]. Available: https://www.usenix.org/conference/nsdi23/presentation/zhang-junxue

[26] T. Yang et al., “Applied federated learning: Improving google keyboard query suggestions,” 2018, *arXiv: 1812.02903*.

[27] K. A. Bonawitz et al., “Towards federated learning at scale: System design,” in *Proc. Mach. Learn. Syst.*, Stanford, CA, USA, 2019, pp. 374–388.

[28] M. Goddard, “The eu general data protection regulation (GDPR): European regulation that has a global impact,” *Int. J. Market Res.*, vol. 59, no. 6, pp. 703–705, 2017.

[29] L. T. Phong, Y. Aono, T. Hayashi, L. Wang, and S. Moriai, “Privacy-preserving deep learning via additively homomorphic encryption,” *IEEE Trans. Inf. Forensics Secur.*, vol. 13, no. 5, pp. 1333–1345, May 2018.

[30] K. Cheng et al., “SecureBoost: A lossless federated learning framework,” *IEEE Intell. Syst.*, vol. 36, no. 6, pp. 87–98, Nov./Dec. 2021.

[31] “netem,” 2019. [Online]. Available: https://man7.org/linux/man-pages/man8/tc-netem.8.html

[32] S. Hardy et al., “Private federated learning on vertically partitioned data via entity resolution and additively homomorphic encryption,” 2017, *arXiv: 1711.10677*.

[33] “FedLearner,” 2021. [Online]. Available: https://github.com/bytedance/fedlearner

[34] “TF Encrypted,” 2022. https://github.com/tf-encrypted/tf-encrypted

[35] A. Acar, H. Aksu, A. S. Uluagac, and M. Conti, “A survey on homomorphic encryption schemes: Theory and implementation,” *ACM Comput. Surv.*, vol. 51, no. 4, pp. 79:1–79: 35, 2018.

[36] R. A. Horn, “The hadamard product,” in *Proc. Symp. Appl. Math*, vol. 40, 1990, pp. 87–169.

[37] “Mellanox ConnectX-4 EN adapter card single/dual-port 100 gigabit ethernet adapter,” 2019. [Online]. Available: https://www.mellanox.com/products/ethernet-adapters/connectx-4-en

[38] “Mellanox SN2100 open ethernet switch,” 2019. [Online]. Available: https://www.mellanox.com/related-docs/prod_eth_switches/PB_SN2100.pdf

[39] “Intel Xeon Silver 4114 Processor,” 2017. [Online]. Available: https://www.intel.com/content/www/us/en/products/sku/123550/intel-xeon-silver-4114-processor-13-75m-cache-2-20-ghz/specifications.html

[40] V. Gogte, A. Kolli, M. J. Cafarella, L. D’Antoni, and T. F. Wenisch, “HARE: Hardware accelerator for regular expressions,” in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Taipei, Taiwan, 2016, pp. 1–12.

[41] S. Xu, T. Bourgeat, T. Huang, H. Kim, S. Lee, and S. Arvind, “AQUOMAN: An analytic-query offloading machine,” in *Proc. 53rd Annu. IEEE/ACM Int. Symp. Microarchitecture*, Athens, Greece, 2020, pp. 386–399.

[42] A. AziziMazreah and L. Chen, “Shortcut mining: Exploiting cross-layer shortcut reuse in DCNN accelerators,” in *Proc. 25th IEEE Int. Symp. High Perform. Comput. Archit.*, Washington, DC, USA, 2019.

[43] D. M. Gordon, “A survey of fast exponentiation methods,” *J. Algorithms*, vol. 27, no. 1, pp. 129–146, 1998.

[44] C. Kaya Koc, T. Acar, and B. S. K. Kaliski, “Analyzing and comparing montgomery multiplication algorithms,” *IEEE Micro*, vol. 16, no. 3, pp. 26–33, Jun. 1996.

[45] M. Ibrahim, “Radix-2^n multiplier structures: A structured design methodology,” *IEE Proc. E. (Comput. Digit. Techn.)*, vol. 140, no. 4, pp. 185–190, 1993.

[46] “Multiplier v12.0 LogiCORE IP product guide,” 2015. [Online]. Available: https://docs.xilinx.com/v/u/en-US/pg108-mult-gen

[47] “Xilinx UltraScale architecture DSP slice user guide,” 2020. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ug579-ultrascale-dsp

[48] “UltraScale architecture memory resources,” 2020. [Online]. Available: https://www.xilinx.com/support/documents/user_guides/ug573-ultrascale-memory-resources.pdf

[49] M. R. Nigam and S. Bande, “AXI interconnect between four master and four slave interfaces,” *Int. J. Eng. Res. Gen. Sci*, vol. 2, no. 4, pp. 2091–2730, 2014.

[50] P. Kumbhare and V. Krishna, “Designing high-performance video systems in 7 series FPGAs with the AXI interconnect,” Xilinx, Inc., San Jose, CA, USA, Rep. XAPP741 (v1.3) , 2012.

[51] “Timing closure user guide,” 2020. [Online]. Available: https://www.xilinx.com/content/dam/xilinx/support/documentation/sw_manuals/xilinx14_7/ug612.pdf,

[52] D. E. Thomas and P. Moorby, *The Verilog Hardware Description Language*. 2. ed., Berlin, Germany: Springer, 1995.

[53] “Super logic region SLR,” 2021. [Online]. Available: https://docs.xilinx.com/r/2021.2-English/ug949-vivado-design-methodology/Super-Logic-Region-SLR

[54] “Xilinx DMA,” 2020. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf

[55] “Laguna Registers,” 2021. [Online]. Available: https://docs.xilinx.com/r/2021.1-English/ug949-vivado-design-methodology/Using-SLR-Crossing-Registers

[56] T. Ye, S. R. Kuppannagari, R. Kannan, and V. K. Prasanna, “Performance modeling and FPGA acceleration of homomorphic encrypted convolution,” in *Proc. 31st Int. Conf. Field- Program. Log. Appl.*, Dresden, Germany, 2021, pp. 115–121.

[57] “Federated machine learning,” 2019. [Online]. Available: https://github.com/FederatedAI/FATE/tree/master/python/federatedml,

[58] “Amazon AWS F1,” 2023. [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/

[59] “Microsoft Azure NP,” 2023. [Online]. Available: https://learn.microsoft.com/en-us/azure/virtual-machines/np-series

[60] “Huawei Cloud FP1,” 2023. [Online]. Available: https://www.huaweicloud.com/product/facs.html

[61] “Xilinx alveo U250 data center accelerator card,” 2020. [Online]. Available: https://www.xilinx.com/products/boards-and-kits/alveo/u250.html

[62] I. San, N. At, I. Yakut, and H. Polat, “Efficient paillier cryptoprocessor for privacy-preserving data mining,” *Secur. Commun. Netw.*, vol. 9, no. 11, pp. 1535–1546, 2016.

[63] “Xilinx Virtex7 FPGAs,” 2019. [Online]. Available: https://www.xilinx.com/support/documentation/selection-guides/virtex7-product-table.pdf

[64] Z. Yang, S. Hu, and K. Chen, “FPGA-based hardware accelerator of homomorphic encryption for efficient federated learning,” 2007, *arXiv: 2007.10560*.

[65] M. Bahadori and K. Järvinen, “A programmable SoC-based accelerator for privacy-enhancing technologies and functional encryption,” *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 28, no. 10, pp. 2182–2195, Oct. 2020.

[66] “Xilinx Zynq UltraScale MPSoCs,” 2019. [Online]. Available: https://docs.xilinx.com/v/u/en-US/zynq-ultrascale-plus-product-selection-guide

[67] “Credit card cheating detection,” 2020. [Online]. Available: https://www.kaggle.com/arslanali4343/credit-card-cheating-detection-cccd

[68] H. Xiao, K. Rasul, and R. Vollgraf, “Fashion-MNIST: A novel image dataset for benchmarking machine learning algorithms,” 2017, *arXiv: 1708.07747*.

[69] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.

[70] “Text generation with an RNN,” 2019. [Online]. Available: https://www.tensorflow.org/text/tutorials/text_generation

[71] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, “Densely connected convolutional networks,” in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Honolulu, HI, USA, 2017, pp. 2261–2269.

[72] A. Krizhevsky et al., “Learning multiple layers of features from tiny images,” Toronto, ON, Canada, 2009.

[73] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Las Vegas, NV, USA, 2016, pp. 770–778.

[74] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” in *Proc. 3rd Int. Conf. Learn. Representations*, San Diego, CA, USA, 2015, pp. 1–14.

[75] “Hetero Fast SecureBoost,” 2022. [Online]. Available: https://fate.readthedocs.io/en/latest/_build_temp/python/federatedml/ensemble/README.html

[76] C. Zhang, S. Li, J. Xia, W. Wang, F. Yan, and Y. Liu, “BatchCrypt: Efficient homomorphic encryption for cross-silo federated learning,” in *Proc. USENIX Annu. Tech. Conf.*, 2020, pp. 493–506.

[77] N. Samardzic et al., “F1: A fast and programmable accelerator for fully homomorphic encryption,” in *Proc. 4th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Greece, 2021, pp. 238–252.

[78] S. Kim et al., “BTS: An accelerator for bootstrappable fully homomorphic encryption,” in *Proc. 49th Annu. Int. Symp. Comput. Archit.*, New York, New York, USA, 2022.

[79] N. Samardzic et al., “Craterlake: A hardware accelerator for efficient unbounded computation on encrypted data,” in *Proc. 49th Annu. Int. Symp. Comput. Archit.*, New York, New York, USA, 2022, pp. 173–187.

[80] Z. Jiang, W. Wang, and Y. Liu, “FLASHE: Additively symmetric homomorphic encryption for cross-silo federated learning,” 2021, *arXiv:2109.00675*.

[81] S. Kumar, D. E. Culler, and R. A. Popa, “MAGE: Nearly zero-cost virtual memory for secure computation,” in *Proc. 15th USENIX Symp. Operating Syst. Des. Implementation*, 2021, pp. 367–385.

[82] C. Zeng et al., “Tiara: A scalable and efficient hardware acceleration architecture for stateful layer-4 load balancing,” in *Proc. 19th USENIX Symp. Networked Syst. Des. Implementation*, Renton, WA, USA, 2022, pp. 1345–1358.

[83] S. Pontarelli et al., “FlowBlaze: Stateful packet processing in hardware,” in *Proc. 16th USENIX Symp. Networked Syst. Des. Implementation*, Boston, MA, 2019, pp. 531–548.

[84] M. S. Brunella et al., “hXDP: Efficient software packet processing on FPGA nics,” in *Proc. 14th USENIX Symp. Operating Syst. Des. Implementation*, 2020, pp. 973–990.

[85] W. Jiang et al., “MicroRec: Efficient recommendation inference by hardware and data structure solutions,” in *Proc. Mach. Learn. Syst.*, 2021, pp. 845–859.

[86] C. Zeng et al., “FAERY: An fpga-accelerated embedding-based retrieval system,” in *Proc. 16th USENIX Symp. Operating Syst. Des. Implementation*, Carlsbad, CA, USA, 2022, pp. 841–856.

[87] F. Zhang, J. Zhai, X. Shen, O. Mutlu, and X. Du, “POCLib: A high-performance framework for enabling near orthogonal processing on compression,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 2, pp. 459–475, Feb. 2022.

[88] J. Zhang, X. Cheng, L. Yang, J. Hu, X. Liu, and K. Chen, “SoK: Fully homomorphic encryption accelerators,” 2022, *arXiv:2212.01713*.