Accelerating Privacy-Preserving Machine Learning With GeniBatch

Xinyang Huang[1], Junxue Zhang[1], Xiaodian Cheng[1,2], Hong Zhang[2], Yilun Jin[1]
Shuihai Hu[1], Han Tian[1], Kai Chen[1,3]
[1]iSING Lab, Hong Kong University of Science and Technology
[2]University of Waterloo [3]USTC

Abstract

Cross-silo privacy-preserving machine learning (PPML) adopts Partial Homomorphic Encryption (PHE) for secure data combination and high-quality model training across multiple organizations (*e.g.*, medical and financial). However, PHE introduces significant computation and communication overheads due to data inflation. Batch optimization is an encouraging direction to mitigate the problem by compressing multiple data into a single ciphertext. While promising, it is impractical for a large number of cross-silo PPML applications due to the limited vector operations support and severe data corruption.

In this paper, we present GeniBatch, a batch compiler that translates a PPML program with PHE into an efficient program with batch optimization. GeniBatch adopts a set of conversion rules to allow PHE programs involving all vector operations required in cross-silo PPML and ensures end-to-end result consistency before/after compiling. By proposing bit-reserving algorithms, GeniBatch avoids bit-overflow for the correctness of compiled programs and maximizes the compression ratio. We have integrated GeniBatch into FATE, a representative cross-silo PPML framework, and provided SIMD APIs to harness hardware acceleration. Experiments across six popular applications show that GeniBatch achieves up to 22.6× speedup and reduces network traffic by 5.4×-23.8× for generic cross-silo PPML applications.

CCS Concepts: • **Security and privacy**; • **Networks**;

Keywords: privacy-preserving machine learning, homomorphic encryption, batch compiler

EuroSys '24, April 22–25, 2024, Athens, Greece

ACM ISBN 979-8-4007-0437-6/24/04…$15.00
https://doi.org/10.1145/3627703.3629563

ACM Reference Format:
Xinyang Huang, Junxue Zhang, Xiaodian Cheng, Hong Zhang, Yilun Jin, Shuihai Hu, Han Tian, and Kai Chen. 2024. Accelerating Privacy-Preserving Machine Learning With GeniBatch. In *European Conference on Computer Systems (EuroSys '24), April 22–25, 2024, Athens, Greece.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3627703.3629563

1 Introduction

Machine learning (ML) has been widely used to increase productivity in industries such as medicine, finance, recommendation services, and threat analysis. Data quality is crucial for training effective ML models, and there is an increasing demand to combine data from different sources. However, gathering data from multiple organizations for centralized model training, *e.g.*, patient medical records from different hospitals [16] and user search histories from different internet companies [72], raises privacy concerns and violates government regulations [6, 26]. To solve this problem, cross-silo privacy-preserving ML (PPML), such as Federated Learning [65], offers an appealing solution to connect "data silos" among organizations. More specifically, a global model is collaboratively learned by aggregating encrypted intermediate results (*e.g.*, gradients/parameters) from multiple data sources without revealing any original data [10, 19, 22, 64].

In practice, large companies or organizations adopt cross-silo PPML in critical businesses that require rigorous security guarantees and high model accuracy. Thus, implementations of PPML based on differential privacy (DP) are rarely used since the noise added by DP degrades model accuracy [9, 13]. Instead, Partial Homomorphic Encryption (PHE), notably Paillier [47], allows direct computation over ciphertexts, thus enabling lossless implementations of various PPML applications [4, 15, 17, 25, 28, 38, 46] (§2.1). Although promising, PHE significantly degrades the performance of PPML. The reason is *data inflation*. For example, a 32-bit floating-point number would expand to an integer with 2048 bits after encryption, causing a 64× data inflation. Such inflation brings significant computation and communication overhead: (1) processing 2048-bit operations is much slower than 32-bit operations in modern CPU architectures [27]; (2) traffic in transferring ciphertexts is 64× greater than in transferring plaintexts. Experiments in §2.2 show that performing operations on large integers (ciphertexts) and transferring them over wide area networks (WANs) are more than 7.13× and

66× slower than plaintext operations and data transfers, respectively. This motivates us to improve PHE performance by mitigating the overhead caused by data inflation.

Batching, *i.e.*, embedding multiple plaintexts into a single ciphertext (denoted as a batch ciphertext), is a promising direction for overcoming the performance degradation caused by data inflation. The reason is that, by batching k plaintexts, we perform k operations simultaneously with one large integer operation, reducing the average computation costs by $k\times$; meanwhile, the total number of ciphertexts decreases by $k\times$, reducing the overhead on data transfer. However, as revealed in §2.3.2, directly applying it suffers from *limited operations support* and *severe data corruption*: (1) Homomorphic operations in PHE only enable vector addition and scalar multiplication over batch ciphertexts. However, such operations fail to support many important PPML scenarios (*e.g.*, Vertical LR [28] and Secure XGBoost [17]), where more complicated operations such as Hadamard product (*i.e.*, element-wise product for two vectors) and inner-sum (*i.e.*, sum all elements in a vector) are required. (2) Performing operations over batch ciphertexts may result in overflow, which leads to corrupted computation results and further degrades model accuracy. The same problems exist in previous batch techniques [12, 39, 63, 68], thus limiting the applicability of batching optimization in cross-silo PPML.

To address the above challenges, we propose GeniBatch, a batch compiler that translates a PHE program with general vector operations into an efficient program with batching (§3). The core idea of GeniBatch is as follows. First, we observe that the fragments of desired results for a single Hadamard product or inner-sum operation are packed in replicated batch ciphertexts. Based on such fragmented information, GeniBatch designs a set of conversion rules for original PHE programs and translates them to dataflow graphs over batch ciphertexts. To further optimize dataflow execution, we design graph rewrite rules that defer relatively inefficient operations to the end so that they only occur once. Second, GeniBatch reserves necessary zero-padding bits and encodes data with necessary sign bits, to prevent overflow as well as maximize compression ratio by scrutinizing the bits expansion in dataflow execution. As a result, GeniBatch enables lossless implementations of general cross-silo PPML applications with batching and mitigates performance degradation caused by PHE in both computation and communication.

To integrate GeniBatch into various secure ML frameworks [1–3], we decouple the implementation of GeniBatch into User Interfaces and a GeniBatch Core (§4). The User Interfaces provides a set of NumPy-like APIs with Python. By using them, users can easily implement vector operations on encrypted data and leverage batching optimization with only minor changes to existing programs. The GeniBatch Core automatically compiles programs with User Interfaces to dataflow over batch ciphertexts and executes it via ML

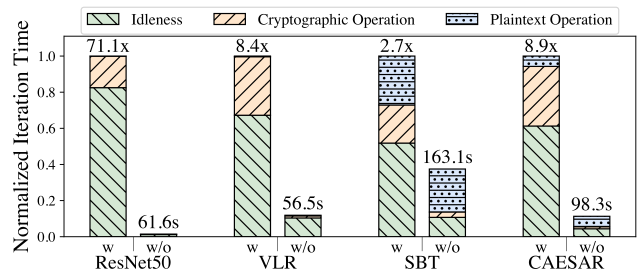


Figure 1. Iteration time breakdowns of Horizontal ResNet50, VLR, SBT, and CAESAR. The two columns in each set of experiments denote training with encryption (w) and without encryption (w/o), respectively.

framework APIs in data storage, communication and secure components. We have integrated GeniBatch into FATE [1], the widely adopted open-source framework for secure computation in the industry. To further harness state-of-the-art hardware acceleration for cryptographic operations in PHE [18], GeniBatch also packs operators and partitions input data to support SIMD (Single Instructions Multiple Data) executions in parallel.

We extensively evaluate GeniBatch in real-world PPML scenarios—two geo-distributed participating servers with a 40-core CPU for parallel execution or a supplemental GPU for cryptographic acceleration. The participating servers collaboratively train six popular PPML models: FedAvg-based [43] ResNet50 [29], DenseNet169 [31], and EfficientNetB0 [52] for horizontal PPML; Vertical Logistic Regression [28], Secure XGBoost[1] [17] and CAESAR [15] for vertical PPML. Compared with the implementations in FATE, GeniBatch achieves promising speedup for all applications: (1) 15.9× to 20.1× and 1.59× to 3.17× improvements for horizontal and vertical PPML applications respectively; (2) 19.5× to 22.6× and 1.66× to 1.95× improvements respectively with GPU acceleration. Note that GeniBatch does not compromise model accuracy, and it is compatible with various optimizations such as relaxed synchronization [30, 37, 41] and model compression [11, 51, 58].

2 Background and Motivation

2.1 Privacy-preserving ML

PHE-based PPML. Many ML applications require massive training data from multiple *participants* in different regions and entities. However, due to the increasingly strict lawsuits and regulations (e.g., GDPR [26]), gathering all the data in one place to perform centralized training is not always possible. Privacy-preserving ML (PPML) has been proposed to train a global model across participants in a decentralized manner, where participants still hold original data and collaboratively aggregate their local intermediate information

[1]For Secure XGBoost, the computation of the histogram is more suitable for the CPU, thus we only evaluate it via 40 cores CPU in parallel.

	Applications	Aggregating Function	Vector operations over ciphertexts	Plaintext Size (MB)	Ciphertext Size (MB)	Supported by Batching PHE
Horizontal PPML	FedAvg	$[[X_G]] = \sum_P [[X_P]]$	VecAdd, ScalarMul	359	23722	◐
Vertical PPML	VLR	$[[X_G]] = ([[X_A]] + [[X_B]]) \cdot Y_A$	VecAdd, HadmrdProd, InnerSum	4	291	○
	SBT	$[[X_G]] = \sum_{P,i} [[X_{P_i}]]$	InnerSum	8	590	○
	CAESAR	$[[X_G]] = X_A \cdot [[X_B]] - \mathbf{c} \cdot Y_A$	VecAdd, ScalarMul, HadmrdProd, InnerSum	8	528	○

Table 1. Aggregating functions and the data size before/after encryption in cross-silo PPML applications. The vector operations over ciphertexts include **Vec**tor **Add**ition, **Scalar Mul**tiplication, **Had**a**m**a**rd Prod**uct, and **Inner-Sum**.

after encryption. PPML has been extensively studied and widely adopted in cross-device and cross-silo settings. In the cross-device setting, the participants are a large number of mobile or IoT terminals with limited computing resources [23, 62, 65]. In contrast, the participants are a few datacenters belonging to different organizations or companies in the cross-silo setting [24, 65, 71]. This paper focuses on the cross-silo setting.

Compared with cross-device PPML, cross-silo PPML rigorously requires privacy guarantees and learning accuracy. Instead of applying differential privacy (DP) [8], most real-world cross-silo PPML implementations [1–3, 15, 17, 25, 28, 38, 46] adopt Partial Homomorphic Encryption (PHE) to encrypt local information and subsequently exchange them between participants. The reason is that DP ensures privacy by adding noise but inevitably degrades model accuracy [9, 13], while PHE allows direct computation over ciphertexts and thus enables lossless implementations of cross-silo PPML applications. In addition, PHE is friendly to existing learning systems as it only requires setting few encryption parameters (*e.g.*, cipher key length) and imposes no constraints for synchronization schemes [68]. Paillier [47] is the most widely used PHE scheme for cross-silo PPML, which supports both homomorphic addition (⊕) and multiplication (⊗) over ciphertexts:

$$[[u]] \oplus [[v]] \coloneqq ([[u]] \times [[v]]) \bmod n^2 = [[u+v]]$$
$$u \otimes [[v]] \coloneqq [[v]]^u \bmod n^2 = [[u \times v]]$$

where $[[\cdot]]$ denotes a ciphertext encrypted with Paillier, and n denotes the cipher key.

Vector operations with PHE. PPML applications can be summarized into two categories according to the data distribution characteristics [65]. As we show in Table 1, they apply different aggregating functions for decentralized training and thus require different vector operations with PHE.

• *Horizontal PPML*: original datasets among participants share the same feature space but different samples. To perform secure aggregation in horizontal PPML, each participant first computes the local gradients with its own dataset and encrypts them via PHE. A central server collects encrypted local gradients from all participants, sums received ciphertexts to obtain the global gradients, and sends them back to each participant for model updating. The aggregating function is a simple sum or average [43], where the central server performs vector addition and scalar multiplication with PHE.

• *Vertical PPML*: original datasets among participants share the same set of samples, but each participant only has a subset of the features. Table 1 shows that the aggregating functions in vertical PPML vary depending on the specific ML models and training protocols, such as Vertical Logistic Regression (VLR) [28], Secure XGBoost (SBT) [17] and CAESAR [15]. In general, four vector operations are involved in vertical PPML: vector addition, scalar multiplication, Hadamard product[2], and inner-sum. Participants leverage PHE to encrypt local intermediate information and perform vector operations over ciphertexts to achieve secure aggregation.

2.2 Performance Overhead of PHE

Although the PHE cryptosystem easily facilitates PPML applications, it brings significant overheads in both communication and computation. To quantify the performance impact of PHE, we decompose the iteration time of four popular PPML applications for deep dive, including FedAvg-based ResNet50 [29], VLR [28], SBT [17] and CAESAR [15]. These applications are profiled both with and without encryption, executed on a 40-core CPU in parallel[3]. We use a commercial dataset from a bank with 1M samples with #100 features, and the bandwidth between participants is 50Mbps. In general, PHE causes 71.1×, 8.4×, 2.7×, and 8.9× performance degradation for four applications, respectively. Figure 1 and Table 1 show the results, and we analyze the reasons below:

Data inflation causes computational and communication overheads. After encrypting with PHE, the size of a single ciphertext expands to 2048 bits (twice the length of the cipher key), and the amount of data transfers in an iteration inflates to 23.7GB, 291MB, 590MB, 528MB, respectively. We observe that performing cryptographic operations over large integers is 99.89×, 27.73×, 7.13×, and 18.99× slower than computing over plaintexts (32-bit numbers), while the data transmission time of inflated ciphertexts is 66.1×, 72.7×, 73.7×, and 66.0× longer than plaintexts, respectively. Such large overheads rooted in data inflations further extend the idle time of participating servers by 74.23×, 6.82×, 5.32×, and 12.94×, respectively, which poses challenges to deploying

[2]Perform Hadamard product between $\mathbf{x}$ and $\mathbf{y}$ is equal to $(x_1 y_1, x_2 y_2, \dots)$
[3]We adopt the implementations of four PPML applications in FATE [4], and the length of cipher key is 1024 bits.

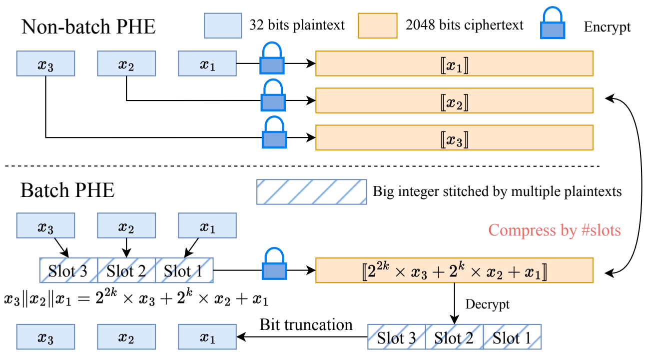


Figure 2. Encrypt a vector (x_1, x_2, x_3) with non-batch PHE and batch PHE respectively.

PPML for real-world applications, *e.g.*, enlarged deployment cost due to high server renting fee, *etc.*

It is not surprising to observe data inflation since the PHE encryption procedure involves big integer operations. For example, in Paillier encryption algorithm, an integer m is encrypted to c by $c = g^m \times r^n \mod n^2$, where g, r, n are all large integers (1024 bits in minimum). As a result, the size of the raw data inflates from 32 bits to 2048 bits. Paillier allows the bit width of plaintext to vary from 1 to 1024; however, only 32 bits are used in realistic implementations of PPML. It indicates an opportunity to mitigate such huge data inflation by fully utilizing the wasted bit width of plaintexts.

2.3 Boost PHE with Batching

2.3.1 Benefit of batch PHE. To address the performance degradation caused by data inflation, we first attempt to directly leverage existing batch strategies [12, 39, 68], *i.e.*, *batching multiple plaintexts* into *a single ciphertext*. As shown in Figure 2, instead of encrypting each plaintext separately (non-batch PHE), plaintexts are firstly stitched into a large integer and subsequently encrypted into a ciphertext at once (batch PHE). Theoretically, when encrypting a list of floating-point numbers with a 1024 bits cipher key, batch PHE can compress ciphertexts by a maximum of 32× compared to the non-batch PHE. Meanwhile, the computation overhead on several cryptographic operations *e.g.,* encryption/decryption and vector addition is also reduced.

To better illustrate how batching could mitigate the problem, we demonstrate the difference in communication/computation overhead before/after applying existing batch strategies through a concrete example. We use the following Horizontal ResNet50 [29] as an example (other vertical applications are hardly implemented with batch PHE and we temporarily ignore the data corruption problem in this example, which we will discuss in §2.3.2).

Suppose a central server uses FedAvg [43] algorithm to aggregate local parameters from two participants. Each participant generates a local ResNet50 model with 25M parameters for each training iteration. To obtain the global parameters, participants first encrypt local parameters and send them to the central server; then, the central server sums all received ciphertexts and sends them back. In this procedure, encryption&decryption and homomorphic additions are performed 25M times in participants and central servers respectively. Meanwhile, each participant must send and receive 25M ciphertexts. However, after applying batch PHE, participants only need to perform 25M/32 encryption&decryption operations and send&receive 25M/32 ciphertexts, and the central server only needs to perform 25M/32 homomorphic additions, which indicates the communication and computation overheads are both reduced by a factor of 32.

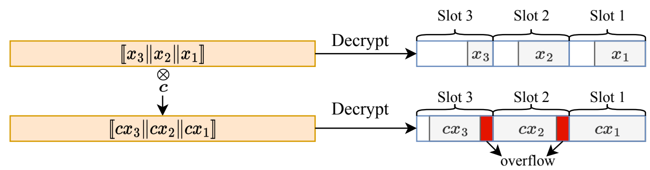


Figure 3. Data corruption due to the slot overflow.

In this paper, we denote a ciphertext encrypted by batch PHE as *batch ciphertext* and refer to the bits that store raw data and results as *slots*. We use the terms $[\![a \| b]\!]$ and $[\![2^k a + b]\!]$ interchangeably in the rest of this paper to denote a batch ciphertext that encapsulates two slots (store a and b).

2.3.2 Challenges of batch PHE We believe that a batch PHE cryptosystem can significantly mitigate the performance penalties caused by non-batch PHE in general. However, implementing all cross-silo PPML applications with batch PHE still needs to address two challenges below:

Challenge 1: limited applications due to partial vector operations support. PHE, *e.g.* Paillier, only supports homomorphic addition and multiplication over ciphertexts. Performing them on batch ciphertexts is shown below:

$$[\![x_1 \| x_2 \| ...]\!] \oplus [\![y_1 \| y_2 \| ...]\!] = [\![x_1 + y_1 \| x_2 + y_2 \| ...]\!] \rightarrow [\![\mathbf{x} + \mathbf{y}]\!]$$

$$c \otimes [\![x_1 \| x_2 \| ...]\!] = [\![c \times x_1 \| c \times x_2 \| ...]\!] \rightarrow [\![c\mathbf{x}]\!]$$

where homomorphic addition and multiplication are equivalent to vector addition and scalar multiplication over ciphertexts, respectively. However, as we mentioned in §2.1, PPML applications require four vector operations with PHE, where Hadamard product and inner-sum are unavailable after naive batching. It indicates that implementing vertical PPML applications with batch PHE is not feasible due to the *partial vector operations support.* Moreover, it is impossible to directly switch ciphertexts from batch modes to non-batch modes to perform unsupported vector operations, as such a switch requires re-decryption and re-encryption, which is considered to breach the privacy guarantee in general. Some previous batch stratagies [12, 39, 63, 68] have explored performing vector operations over batch ciphertexts with homomorphic operations, as we mentioned above, but still do not support Hadamard product and inner-sum.

Challenge 2: data corruption due to the slot overflow.

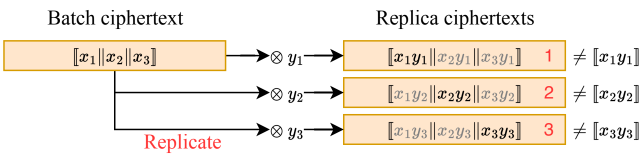


Figure 4. Realize Hadamard product over batch ciphertexts by replication, and the desired results are x_1y_1, x_2y_2 and x_3y_3. Numbers in replica ciphertexts denote the valid slots.

After performing a sequence of operations on a batch ciphertext, data after decryption may be corrupted due to the *slot overflow*. As shown in Figure 3, after multiplying a constant with a batch ciphertext and decrypting it, bits in slots 1 and 2 are not sufficient to store the results cx_1 and cx_2, which causes slot overflows and further leads to data corruption. Once the highest slot overflows, all data in a batch ciphertext would be completely corrupted. In practice, the slot overflow frequently occurs in aggregating functions which contain multiplication or massive additions, *e.g.*, sum operations for ciphertexts in the previous example of Horizontal ResNet50. Existing work [68] uses two sign bits to detect overflow but cannot prevent it for addition, and such an encoding scheme is ineffective for multiplication, *e.g.*, the MSB (most significant bit) overflows to a higher slot when performing addition or multiplication between negative numbers. As a result, participants in cross-silo PPML applications may receive corrupted global information after aggregation.

3 GeniBatch

To boost the performance of general cross-silo PPML applications with batching, we present GeniBatch, a batch compiler that addresses the above two challenges. GeniBatch achieves the following desirable properties: (1) it allows original programs to contain four vector operations required for cross-silo PPML applications, and automatically compiles them to executable programs with batch PHE (§3.1); (2) it prevents slot overflow when performing operations on batch ciphertexts, enabling lossless implementations of PPML applications (§3.2); and (3) it further optimizes the dataflow execution to achieve maximum computational efficiency (§3.3). With GeniBatch, users can implement secure aggregation as with non-batch PHE, while the performance overhead due to the data inflation is significantly reduced.

3.1 End-to-end Dataflow with batch PHE

To compile non-batch PHE programs to batch, the primary target is to handle unsupported vector operations. We observe that simply replicating a batch ciphertext for *#slots* provides opportunities to realize Hadamard product and inner-sum. For the example of Hadamard product in Figure 4, after performing replication and homomorphic multiplication, desired results (x_1y_1, x_2y_2 and x_3y_3) are hidden in three replica ciphertexts respectively. However, such three

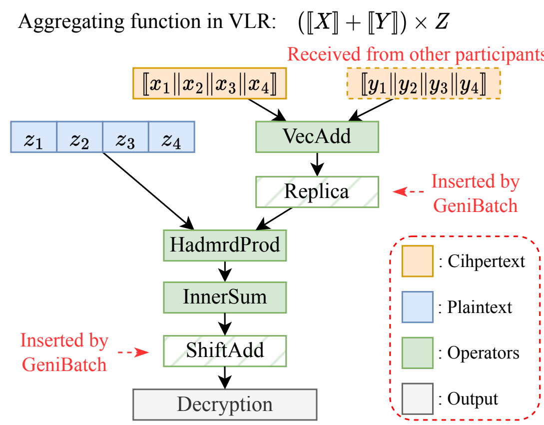


Figure 5. The generated end-to-end dataflow over batch ciphertexts for VLR.

ciphertexts are inconsistent with the original ones, *i.e.*, the outputs of Hadamard product in the non-batch PHE program are $[[x_1y_1]]$, $[[x_2y_2]]$ and $[[x_3y_3]]$). Therefore, directly performing subsequent operations over replica ciphertexts (*e.g.*, vector addition or inner-sum) will result in incorrect outputs.

We believe that it is feasible to satisfy end-to-end consistency between original non-batch PHE programs and compiled batch PHE programs, by leveraging implicit results in replica ciphertexts. To achieve it, GeniBatch applies the following conversion rules to ensure that desired results are hidden in the output for each vector operation after compiling. A compiled batch PHE program is represented via dataflow, as shown in Figure 5, and we summarize corresponding operators in Table 2.

• *Rules for operations in replica mode.* For Hadamard product and inner-sum, GeniBatch inserts Replica to convert their inputs to replica ciphertexts and after performing homomorphic operations, each of them stores a fragment of desired results. For example, in Table 2, the output ciphertexts of HadmrdProd and InnerSum store $x_1y_1\&x_2y_2$ and $x_1 + x_3\&x_2 + x_4$, respectively. Directly merging the replica ciphertexts after InnerSum (*e.g.*, obtain $x_1+x_2+x_3+x_4$) will damage the implicit results in them. For example, merging fragments of InnerSum with $[[x_1+x_3\|\text{n/a}]] \oplus [[\text{n/a}\|x_2+x_4]]$ will damage x_1+x_3 and x_2+x_4. Therefore, GeniBatch inserts ShiftAdd that consists of shift operations and homomorphic additions, enabling addition among replica ciphertexts. The shift operations can be implemented with homomorphic multiplication, *i.e.*, $[[x\|\text{n/a}]] := [[x]] \otimes 2^k$, where k denotes the bit width of a slot.

• *Rules for operations in batch mode.* As vector addition and scalar multiplication are available over batch ciphertexts, no special operations are inserted by GeniBatch. Note that if vector addition occurs after HadmrdProd and InnerSum, GeniBatch substitutes it with Replica and ShiftAdd to ensure the additions over replica ciphertexts are consistent with the original vector addition.

Operator	Ciphertext Mode	Formula Representation	Slot Expansion	Bit Expansion
VecAdd	Batch mode	$[[x_1 \| x_2 \| \ldots]] \oplus [[y_1 \| y_2]] = [[x_1 + y_1 \| x_2 + y_2]]$	Null	1
ScalarMul	Batch mode	$c \otimes [[x_1 \| x_2]] = [[cx_1 \| cx_2]]$	Null	bits of c
HadmrdProd	Replica mode	$[[x_1 \| \text{n/a}]] \otimes y_1, [[\text{n/a} \| x_2]] \otimes y_2 \rightarrow [[x_1 y_1 \| \text{n/a}]], [[\text{n/a} \| x_2 y_2]]$	Null	bits of $max(y_1, y_2)$
InnerSum	Replica mode	$[[x_1 \| \text{n/a}]], [[\text{n/a} \| x_2]], [[x_3 \| \text{n/a}]], [[\text{n/a} \| x_4]] \rightarrow [[x_1 + x_3 \| \text{n/a}]], [[\text{n/a} \| x_2 + x_4]]$	Null	$\lceil \log(\text{num of adds}) \rceil$
Replica	Batch → Replica mode	$[[x_1 \| x_2]] \rightarrow [[x_1 \| \text{n/a}]], [[\text{n/a} \| x_2]]$	Null	Null
ShiftAdd	Replica → Batch mode	$[[x_1 \| \text{n/a}]], [[\text{n/a} \| x_2]] \rightarrow [[\text{n/a} \| x_1 + x_2 \| \text{n/a}]]$	$\#slots - 1$	$\lceil \log(\text{num of adds}) \rceil$

Table 2. GeniBatch operations and corresponding slot & bit expansion, each batch ciphertext is assumed to contain two slots.

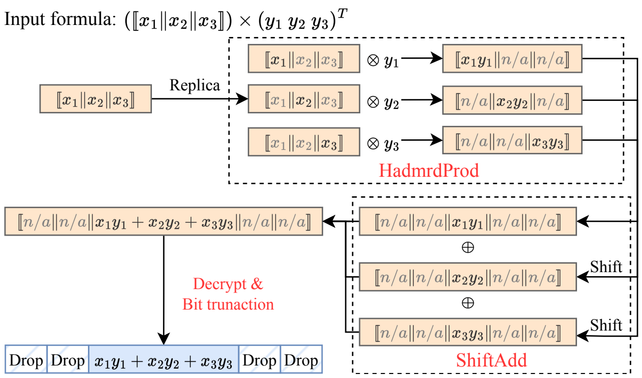


Figure 6. Performing vector dot-production with GeniBatch operations.

To facilitate the understanding of how GeniBatch supports all vector operations over batch ciphertexts and prove the correctness of the compiled program, we show a concrete example that performs a dot-product between a batch ciphertext $[[\mathbf{x}]]$ and an unencrypted vector $\mathbf{y}$ in Figure 6. GeniBatch uses a HadmrdProd to multiply the two vectors' corresponding elements and uses a ShiftAdd to sum up them. After decryption, the result of dot-product can be extracted from the valid slot (the third slot). In this case, we assume that each slot has enough bits to avoid overflow, and defer the discussion on the allocation of bits for slots to §3.2.

For the compiled dataflow, GeniBatch initially assumes each encrypted vector (the inputs of the dataflow) has been batched in one batch ciphertext. Such an assumption may not be feasible (*e.g.*, batching millions of numbers in one ciphertext) and we will discuss how to split it to practice in §3.2. When executing the dataflow, the encryption/decryption, VecAdd and ScalarMul are performed in batch mode, which indicates their computational overhead is negligible. Meanwhile, the communication overhead for exchanging ciphertexts is reduced by $\#slots$. Although the inserted Replica and ShiftAdd operations incur extra computational overhead, we will show that the GeniBatch is still more efficient than non-batch PHE after dataflow optimization in §3.3.

3.2 Batching without Overflow

The reason for slot overflow is *bit expansion*, which commonly occurs in two's complement operations. For example,

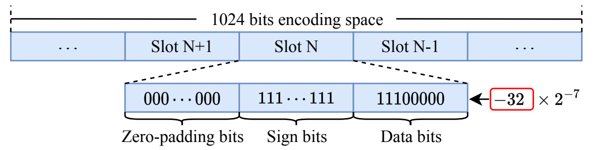


Figure 7. The components of encoded plaintexts and slots, where −0.25 is encoded into Slot N.

the bit width of multiplying two 32-bit floating-point numbers will expand to 64 bits, of which high 32 bits are invalid. Modern CPUs handle such expanded bits by throwing them away. However, it is impossible to do so after encryption via PHE, which means the expanded bits will spread to higher slots and consequently cause data corruption. As a result, the only way to prevent slot overflow is to reserve enough zero-padding bits in advance. GeniBatch achieves it by assigning zero bits for each slot before encryption, and we refer to this process as batch encoding. The design principle of batch encoding is to maximize the number of slots in each batch ciphertext (*i.e.*, maximize the compression ratio), in other words, only to reserve necessary zero bits. In this section, we first describe the components of a batch encoding number (§3.2.1) and then discuss how to assign bits for each component given a GeniBatch dataflow (§3.2.2).

3.2.1 Components of a Batch Encoding Number

Figure 7 shows the components of a batch encoding number that encodes −0.25 in Slot N, and we use the term encoding space to denote its maximum bit width (usually 1024 bits). Such a batch encoding number is divided into several slots by bits, and each slot stores an element of input data. Each slot consists of three parts:

- *Data bits.* As PHE can only encrypt integers, the input data is represented with a fixed-point or quantized representation in practice [1, 68]. Each input data is encoded to an integer associated with an unencrypted scaling factor (*e.g.*, −0.25 is encoded to -32×2^{-7} with 8 bits, where the scaling factor is 2^{-7}). Data bits are used to store the integer part of an encoded plaintext, where the most significant bit (MSB) represents the sign of input data.
- *Sign bits.* GeniBatch encodes the integer part of a fixed-point or quantized number with two's complement to represent both positive and negative numbers. According to arithmetic operations over 2's complements, the MSB of data

Algorithm 1 Bit-reserving

Input:
$(Op_0, ..., Op_n)$, $(D_0, ..., D_n)$: GeniBatch operators and corresponding data inputs.
Acc: Minimum data precision.
Range: The range of data.
Output: Batch scheme.

1: #data_bits= $\log_2(Range/Acc)$
2: // Compute slots and bits expansion for each operator.
3: **for** $i = 0, 1, \ldots, n$ **do**
4: #extra_slots← SlotsExpansion(Op_i, D_i)
5: // #res_bits maintains the current bit width after an operation.
6: #res_bits← BitsExpansion(Op_i, D_i)
7: **end for**
8: #sign_bits=#res_bits−#data_bits
9: // Reserve bits of sign expansion
10: **for** $i = 0, 1, \ldots, n$ **do**
11: #res_bits← BitsExpansion(P_i, D_i)
12: **end for**
13: #pad_bits=#res_bits − #sign_bits − #data_bits
14: **return** #data_bits, #sign_bits, #pad_bits, #extra_slots

bits should be repeated in all sign bits so that the decoder can correctly identify the sign of calculation results.

• *Zero-padding bits*. When performing computations over batch ciphertexts, the sign bits of each slot would expand to higher bits according to the two's complement operations. Zero-padding bits are used to store the expanded sign bits and prevent them from overflowing to higher slots.

3.2.2 Batch Scheme Generation

The next question is how to generate a *batch scheme* that determines the total number of slots in a batch encoding number and assigns bit width of data bits, sign bits, and zero-padding bits for each slot. Such a batch scheme must ensure the sign bits can only expand to zero-padding bits and the MSB of sign bits can correctly represent the sign of calculation results after a sequence of cryptographic operations.

Assigning bits for data bits (#data_bits) is straightforward. Depending on the input data's range and accuracy, #data_bits should be larger than $\log_2(max - min)/accuracy$. For example, suppose the input data varies from -1 to 1 and its minimum accuracy is 2^{-7}, the bit width of data bits should be larger than 8. To ensure no overflow occurs when executing the GeniBatch dataflow, we secondly scrutinize the impact of the GeniBatch operations on the number of slots (#slots) and the bit width of each slot (#bits), which we have summarized in Table 2:

- *Slot shifting*: The operations involving shift will increase #slots, *e.g.*, ShiftAdd will increase #slots by #slots−1.
- *Bit expansion*: Performing any arithmetic operation over batch ciphertexts will expand #bits. For example, #bits would expand by $\lceil \log_2 t \rceil$ for InnerSum, where t denotes the number of homomorphic additions.

Based on the above analysis, a batch scheme can be generated by scanning GeniBatch dataflow and calculating the slots&bits expansion for the eventual results. As shown in Algorithm 1, GeniBatch scans all operators in a given dataflow and sums up the slots&bits expansion by looking up Table 2 (from line 1 to line 7). To ensure that the MSB of results can correctly represent the sign of the results, GeniBatch sets the sign bits by replicating the MSB of data bits for #res_bits−#data_bits times. Note that the sign bits may still expand to higher after a sequence of cryptographic operations. Therefore, GeniBatch scans operators again to determine the width of zero-padding bits (*i.e.*, all bits are set to 0) to prevent slot overflow (from line 9 to line 13). To prevent the highest slot from overflowing, GeniBatch reserves extra slots for slot shifting. Such extra slots are not allowed to store input data (*i.e.*, all bits in extra slots are set to 0). With the batch scheme, GeniBatch encodes input data to batch encoding numbers and subsequently encrypts them to batch ciphertexts. Executing the GeniBatch dataflow on such batch ciphertexts will never cause slot overflow.

Take the dataflow of VLR as an example, which we have mentioned in Figure 5. Suppose the width of data bits is 32 and the encoding space is 1024 bits. After first scanning the dataflow, GeniBatch gets the width of results as 66 bits, thus assigning 34 bits for sign bits, where 1, 32 and 1 bits expansion due to VecAdd, HadmrdProd and ShiftAdd respectively. Next, GeniBatch secondly scans the dataflow to assign zero-padding bits as $68 = 1 + 66 + 1$. Finally, GeniBatch generates a batch scheme with 32 data bits, 34 sign bits, 68 zero-padding bits, and $\#slots - 1$ extra slots, which indicates each batch encoding number can encapsulate 7 slots (compute with $\lfloor 1024/(32 + 34 + 68) \rfloor$) where the lowest four slots are valid to store plaintexts.

Afterward, GeniBatch will batch-encode all input data with the same batch scheme, and split dataflow from Figure 8 (a) to Figure 8 (b). The dataflow splitting procedure is straightforward (*i.e.*, split each encrypted vector to multiple batch ciphertexts, and split operators based on vector arithmetic properties), thus we omit the details. Several *scaling operators* are always inserted before each additive operator (*e.g.*, VecAdd, InnerSum and ShiftAdd) to align the scaling factors of its inputs. In §3.3 we will discuss how to eliminate unnecessary scaling operators before dataflow execution.

3.2.3 Correctness Proof for Batch Encoding

We first establish the losslessness of GeniBatch quantization. Assume we need to encode a 32-bit (with 23-bit mantissa) floating-point number x in $[-a, a]$. GeniBatch will adopt $r = \log_2 a + 24$ as the #data_bits and quantize x to $q_x = \lfloor 2^{r-1} x/a \rceil$, where $\lfloor \cdot \rceil$ denotes rounding. We use $s = 2^{r-1}/a$ to denote the scaling factor. After dequantization with $\hat{x} = q_x/s$, the error is bounded by $|x - \hat{x}|$, smaller than $1/s = 2^{-23}$. Therefore, the quantization error is smaller than the least significant bit

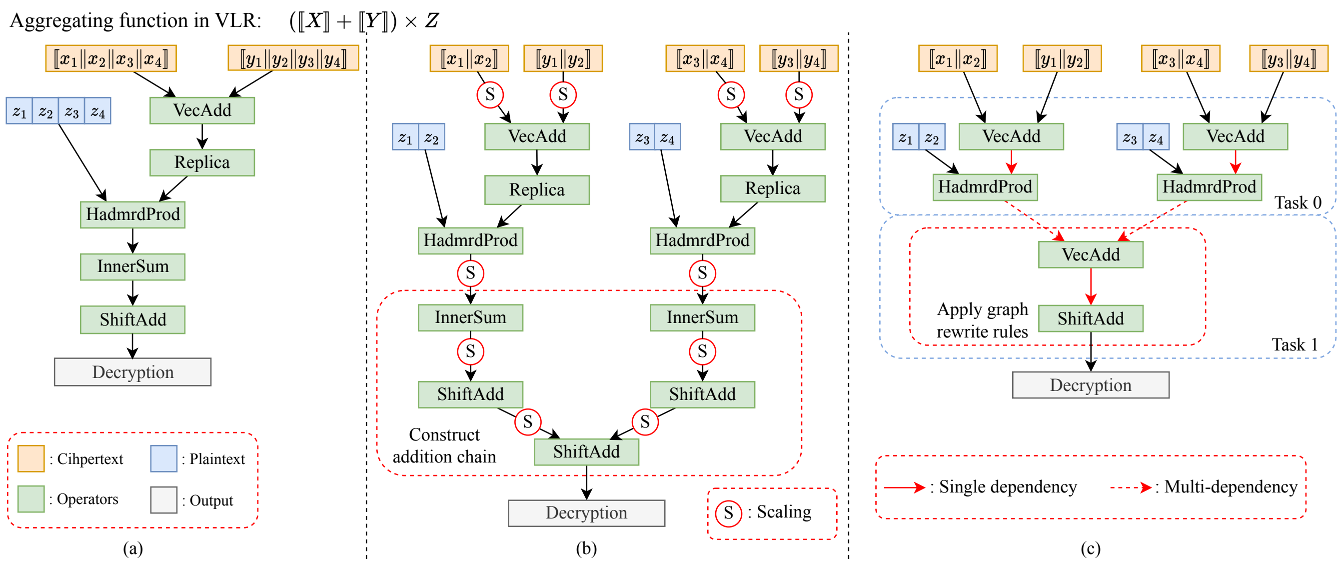


Figure 8. Internal dataflow in GeniBatch.

of the 23-bit mantissa, which proves that no precision loss during quantization and dequantization.

GeniBatch next simply fills the quantization numbers into slots, padding sign bits and zero-padding bits. As we mentioned in §3.2.2, the slots will not overflow during data processing, and the calculation correctness is guaranteed by two's complement computation and homomorphic properties. Thus, GeniBatch is a lossless batch encoding scheme.

3.3 Dataflow Optimization

With the batch scheme, GeniBatch obtains an executable dataflow over batch ciphertexts, which ensures no overflow occurs during the execution. However, the dataflow may not be optimal due to ShiftAdd and unnecessary scaling operators: (i) shift operations included in ShiftAdd are time-consuming and cause slots expansion (reduces the overall compression ratio); (ii) the scaling operators implemented with homomorphic multiplication are more expensive than additive operators implemented with homomorphic addition.

In this section, we design *graph rewrite rules* to optimize the dataflow (§3.3.1) and further analysis the performance of dataflow execution compared with non-batch PHE (§3.3.2).

3.3.1 Graph Rewrite Rules The insight of dataflow optimization is to reduce the number of ShiftAdd via lazy operating, so that shift operations are delayed to the end and only occur once. We observe that a shift operation occurs when adding two replica ciphertexts in which slots are unaligned. Based on it, GeniBatch eliminates shift operations by succinctly applying the two following graph rewrite rules.

Addition chain reordering. GeniBatch scans subsequent additive operators (VecAdd, InnerSum and ShiftAdd) from a replica ciphertext to form an addition chain, which records all input values that need to sum up. The opportunity to reduce shift operations is to perform as many additions in batch mode (VecAdd) as possible by reordering the operators in the addition chain. To achieve it, GeniBatch rewrites the subgraph in two steps: (i) groups input values by slots; and (ii) adds intra-group values with VecAdd and adds inter-group values with ShiftAdd.

Figure 9 (a) shows an example that applies addition chain reordering to a subgraph of dataflow. The ShiftAdd is delayed until the end and the number of ShiftAdd is reduced, and thus executing the subgraph would be faster after addition chain reordering. In practice, such a subgraph frequently occurs in the histogram construction of SBT [17].

Multiplication passing. GeniBatch applies multiplication passing to defer a ShiftAdd operator after multiplicative operators (*e.g.*, ScalarMul). Such the graph rewrite rule is used to extend the addition chain mentioned above, increasing opportunities to reduce the ShiftAdd operations. As shown in Figure 9 (b), each input value of original ShiftAdd ($n_1.value$ and $n_2.value$) should perform ScalarMul with the corresponding scalar ($n_3.value$). Afterward, GeniBatch greedily reduces the ShiftAdd operators by applying addition chain reordering. In practice, such a subgraph occurs when adopting gradient optimizers in VLR (*e.g.,* Adam [33] and RMSprop [54]) during the secure aggregation procedure.

GeniBatch applies the above graph rewrite rules to optimize the performance of the dataflow execution. In the meantime, unnecessary scaling operators are also eliminated by checking the scaling factors of the inputs for each additive operator. As shown in Figure 8 (c), the ShiftAdd is deferred to the end, and all scaling operators are removed since all input data in the dataflow have the same scaling factor after batch encoding.

3.3.2 Performance Analysis As mentioned in §3.1, GeniBatch inserts additional Replica and ShiftAdd in dataflow, incurring extra computational overhead. In this section, we

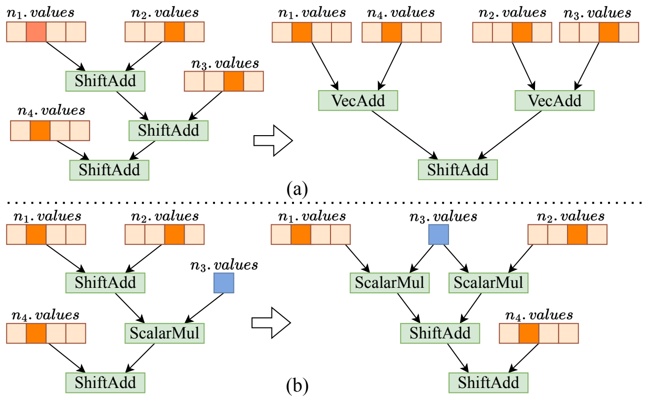


Figure 9. Two graph rewrite rules in GeniBatch: (a) applying addition chain reordering and (b) applying multiplication passing.

show that after applying dataflow optimization, the extra overhead is negligible and the performance of dataflow execution is more efficient than non-batch PHE. Before analysis, we assume the input encrypted/unencrypted vector includes n elements and #slots in a batch ciphertext is m.

• *Operations on batch mode.* Encryption/decryption, VecAdd and ScalarMul are performed over batch ciphertexts, thus only n/m homomorphic operations are involved in GeniBatch while n operations in non-batch PHE. Therefore, such operations in GeniBatch are always more efficient than non-batch PHE.

• *Operations on replica mode.* HadmrdProd and InnerSum are performed over replica ciphertexts. In this case, GeniBatch involves the same number of homomorphic operations as non-batch PHE, *e.g.* performing n homomorphic multiplications when conducting HadmrdProd.

• *Replica and ShiftAdd.* Replica and ShiftAdd both occur once after optimization, which incurs m ciphertext replication and $m-1$ homomorphic additions&multiplications. As m is much smaller than n, *e.g.*, $m = 6, n = 10^6$ in VLR, the overhead of these operations is negligible.

4 Implementation

Integrated with secure ML framework. We have fully integrated GeniBatch into FATE (v1.8) [1] as a boosting engine. Note that GeniBatch can also be applied to other secure ML frameworks, *e.g.,* FedLearner [2], TF Encrypted [3], *etc.* GeniBatch consists of User Interfaces and GeniBatch Core, and the overall architecture and workflow of it are shown in Figure 10. To facilitate GeniBatch usage, User Interfaces provides a set of Numpy-like APIs and original GeniBatch operations with Python (*e.g.*, matrix addition, dot-product, *etc.*) and users can easily leverage them to enable aggregating functions by writing a program with them. The GeniBatch Core consists of Compiler, Batch Encoder, Executor, and Manager. The Compiler is responsible for translating a program with User Interfaces into an executable dataflow

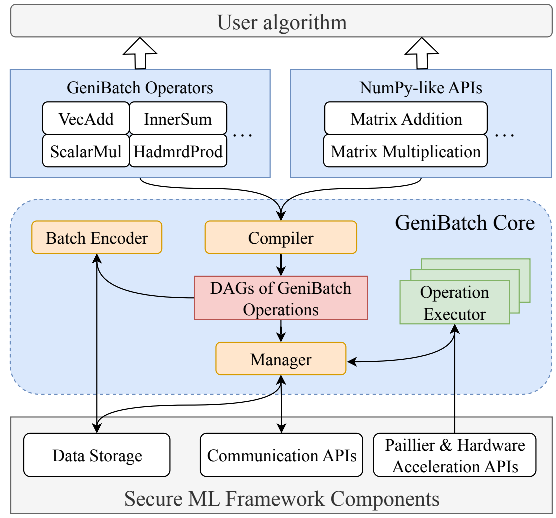


Figure 10. Overall architecture and workflow of GeniBatch.

graph, as mentioned in §3.3. Afterward, the Batch Encoder generates a batch scheme with Algorithm 1 and encodes or batch encodes all input data with the batch scheme. The elements of encrypted vectors are batched into multiple batch encoding numbers, and the elements of unencrypted vectors are encoded to two's complement representation with the #sign_bits and #data_bits in the batch scheme. GeniBatch implements operations over batch ciphertexts based on homomorphic additions and multiplications, which are supported by most secure ML frameworks or third-party libraries. To facilitate usage, all operations are encapsulated into a set of Executors, where each Executor interacts with the underlying secure ML framework or third-party libraries to call the corresponding homomorphic operations. For example, VecAdd can be implemented by calling the PaillierAdd operation in FATE. Finally, the Manager is in charge of executing dataflow by calling Executors in sequence and exchanging results between participants via communication APIs in underlying frameworks (*i.e.*, the federated APIs in FATE).

Parallel execution. GeniBatch utilizes multiprocess module of Python3 to achieve parallelization in CPUs and utilizes the implementation of HAFLO [18] (a set of Paillier operations accelerated with GPUs) to achieve parallelization in GPUs. The Manager classifies the dependencies between two operators into two types: single dependency, where #parent of an operator is one; and multi-dependency, where #parent of an operator is more than one. The Manager packs the operators with single dependencies into a single task and executes it in parallel by partitioning data inputs. For operators with multi-dependencies, the Manager first merges the results in parent nodes, and subsequently re-partitions inputs and starts the next task in parallel. In the graph of Figure 8(c), the operators are packed into two tasks based on dependencies. The Manager partitions the inputs of each task and calls the executors in parallel via multiprocess for CPUs or HAFLO APIs for GPUs.

5 Evaluation

In this section, we evaluate GeniBatch with 6 real-world PPML applications involving geo-distributed datacenters (§5.1). First, we demonstrate that GeniBatch boosts end-to-end performances of various PPML applications by up to 22.6× (§5.2). Second, we deep dive into GeniBatch to investigate the impact of each design component on the performance improvement (§5.3). Finally, we present the scalability of GeniBatch (§5.3.5).

All GeniBatch and benchmarking application codes are available at: https://github.com/Huangxy-Minel/GeniBatch.

5.1 Methodology

Testbed Setup. We consider PPML applications with two participants and a central orchestrating server if required. To evaluate the scalability of the proposed system, we involve more participants, as described in §5.3.5. Each participant is equipped with an Intel Xeon Gold 5218R CPU, 256GB RAM, and an NVIDIA GeForce RTX 3090 GPU. Since participants exchange encrypted information over WANs in practical geo-distributed settings, we use tc qdisc [7] to restrict the sending bandwidth between participants to 50Mbps. We deploy FATE v1.8 as the underlying secure ML framework.

Benchmarking Applications. We evaluate GeniBatch with six benchmarks. For horizontal PPML applications, we choose FedAvg [43] algorithm with three representative models, ResNet50 [29], DenseNet169 [31], and EfficientNetB0 [52], training them with CIFAR10 [34] dataset. For vertical PPML applications, we choose three popular algorithms, Vertical Logistic Regression (VLR) [28], Secure XGBoost (SBT) [17], and CAESAR [15], training them with SUSY [14] dataset (100% dense, 5M samples and 18 features) and a synthetic dataset (100% dense, 1M samples and 100 features). We adopt the Paillier cryptosystem [47] with 1024-bit keys as the PHE scheme for secure aggregation. Participants train models in parallel under two settings: (1) over a 40-core CPU (denoted as the *CPU* setting); and (2) using a GPU to accelerate cryptographic operations (i.e., Paillier encryption/decryption and homomorphic computations), while plaintext computations (e.g., compute local gradients/parameters) are still executed on a 40-core CPU (referred to as the *GPU* setting).

Baselines. For the CPU setting, we use the non-batch Paillier implementation in FATE as the baseline. For the GPU setting, we use the HAFLO [18] implementation (also implemented on FATE) as the baseline. We also compare GeniBatch to the ideal plaintext learning (*i.e.*, plain distributed learning where no encryption is involved, over CPU setting) and the state-of-the-art BatchCrypt [68]. Since BatchCrypt introduces loss quantization while GeniBatch does not, we impose the same quantization precision (*i.e.*, 23 bits accuracy for floating-point numbers) on both to ensure a fair comparison.

Metric	ResNet50			VLR with SUSY		
	Naive	BatchCrypt	GeniBatch	Naive	BatchCrypt	GeniBatch
Time	156.4	218.3	217.9	235.9	284.3	414.4
Acc	0.621	0.626	0.649	0.513	0.524	0.746

Table 3. Iteration **Time**(s) and **Acc**uracy of **Naive** batching, **BatchCrypt**, and **GeniBatch**.

Metrics: In this study, we present the average iteration time and model accuracy for 50 epochs (in horizontal applications) or 20 iterations (in vertical applications). Note that we employ the Area Under the ROC Curve (AUC) for vertical applications as the default accuracy metric.

5.2 End-to-End Evaluation

We first evaluate the end-to-end performance of GeniBatch with 6 PPML applications. Our findings show that GeniBatch significantly enhances the end-to-end performance of all studied PPML applications, resulting in a speedup ranging from 1.59 to 22.6×.

GeniBatch vs. Plain and Non-batch baselines. We report the average iteration time, speedup, and accuracy for all benchmarks, with and without GeniBatch in Table 4. As HAFLO does not support SBT, the performance of SBT on GPU is not available. We make the following observations:

- For horizontal PPML, we observe consistent and significant speedups achieved by GeniBatch (15.9-22.6×) for all three models. For vertical PPML, although less significant, GeniBatch also boosts the end-to-end performance by 1.59-4.25×. These end-to-end speedups demonstrate that GeniBatch is effective and versatile for general PPML applications.
- GeniBatch achieves similar speedups under both CPU and GPU settings, which demonstrates the compatibility of GeniBatch with different hardware supports.
- GeniBatch achieves identical accuracy as the plain learning and non-batch PHE baselines. It indicates that GeniBatch is a lossless technique and will not affect the model convergence, corresponding to the analysis in §3.2.3.
- The speedups vary significantly across different applications. For instance, while GeniBatch yields about 20× speedup in horizontal PPML, it only provides an average of 2.7× in vertical PPML. Additionally, vertical PPML applications exhibit varying degrees of improvement, ranging from 1.59-4.25×. We will delve into the variations in §5.3.

GeniBatch vs. Other batch techniques. We compare GeniBatch to the naive batching [12, 39] (discussed in §2.3) and BatchCrypt with ResNet50 and VLR, as shown in Table 3. For the horizontal cases, GeniBatch achieves the same iteration time as BatchCrypt since they employ an identical batch scheme (*i.e.*, 2 bits for zero-padding bits and 1 bit for sign bits). There is a slight degradation in accuracy because BatchCrypt clips parameters into a fixed range and the naive

Dataset	Application	Plain	CPU Setting				GPU Setting				Accuracy (All methods)
			Non-batch	GeniBatch			Non-batch	GeniBatch			
		Time	Time	Time	vs. Plain	vs. Non-batch	Time	Time	vs. Plain	vs. Non-batch	
CIFAR10	ResNet50	61.6	4378	217.9	↓3.53×	↑20.1×	3133	138.5	↓2.24×	↑22.6×	0.649
	DenseNet169	35.8	2313	131.2	↓3.66×	↑17.6×	1663	75.6	↓2.11×	↑22.0×	0.716
	EfficientNetB0	15.3	798.8	50.2	↓3.29×	↑15.9×	566.2	29.1	↓1.91×	↑19.5×	0.889
SUSY	VLR	132.5	1268	414.4	↓3.13×	↑3.06×	742.8	277.0	↓1.71×	↑3.27×	0.746
	SBT	451.8	1608	677.7	↓1.50×	↑2.37×	-	-	-	-	0.855
	CAESAR	190.1	2548	598.8	↓3.15×	↑4.25×	901.6	297.4	↓1.56×	↑3.03×	0.728
Synthesis	VLR	56.5	475.7	206.8	↓3.66×	↑2.30×	223.9	114.9	↓2.03×	↑1.94×	0.782
	SBT	163.1	435.2	274.2	↓1.68×	↑1.59×	-	-	-	-	0.849
	CAESAR	98.4	870.5	281.8	↓2.86×	↑3.08×	273.0	164.0	↓1.66×	↑1.66×	0.754

Table 4. Iteration time(s), speedup, and accuracy of GeniBatch compared to non-batch and plain distributed baselines. GeniBatch achieves the same accuracy as other baselines.

batching results in overflows. For the vertical cases, forced using naive batching or BatchCrypt results in a significant degradation in accuracy, as predicted in our analysis in §2.3.2.

5.3 Performance Breakdown

In this section, we deep dive into GeniBatch to (1) show the effectiveness of each design component, and (2) analyze how GeniBatch accelerates different PPML applications. We first analyze how the batch encoding scheme (§3.2) affects the compression ratio and hence mitigates the communication overhead. Second, we analyze how the designed operations for batch ciphertexts (§3.1) mitigate the computation overhead for different vector operations under a fixed compression ratio (*i.e.*, the minimal ratio in experiments for six PPML applications). Finally, we analyze how the graph rewrite rules (§3.3) optimize GeniBatch dataflow execution and improve the overall performance. To balance the computation and communication ratio for vertical cases, we adopt the synthetic dataset as default.

5.3.1 Communication Speedup by Batch Scheme

We first analyze how the batch encoding scheme reduces communication traffic under different vector operations. As shown in §3.2.1 and Table 2, different vector operations trigger different bits/slots expansions, which in turn determines the number of slots in each batch ciphertext, *i.e.*, the compression ratio. We show the compression ratio, both in theory and in practice, with different numbers of homomorphic additions and multiplications in Figure 11, from which make the following observations.

• The compression ratio is not highly sensitive to the number of homomorphic additions. For example, with one homomorphic multiplication, the compression ratio decays from 8 to 6 after 10^6 additions. The observation corresponds with Table 2 that t additions expand the number of bits by $\log t$.

• The compression ratio is sensitive to the number of multiplications. For example, with 10^6 additions, the compression ratio drops from about 15 to 6 after performing once homomorphic multiplication. The observation corresponds with Table 2 that a multiplication (ScalarMul and HadmrdProd)

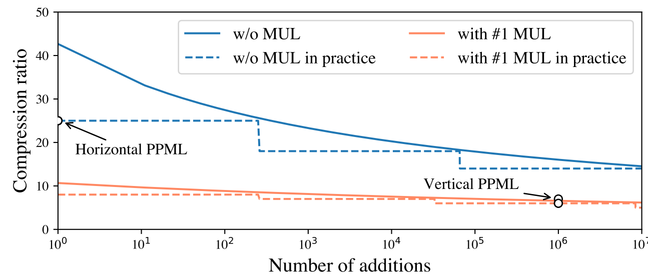


Figure 11. Compression ratio of GeniBatch varies with the number of homomorphic additions. In practice, the bit width of each slot should be multiples of 8.

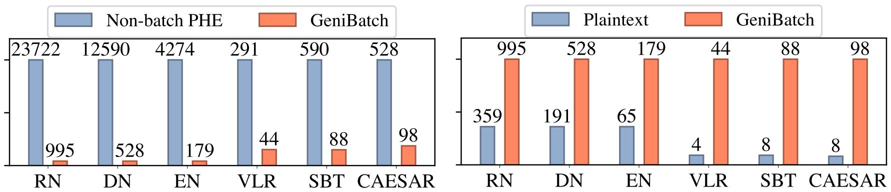
with a n bits number expands result bits by n. However, as up to 1 multiplication is used in the studied PPML applications, GeniBatch can still deliver a promising compression ratio in practice (over 6×).

Next, we proceed to analyze how GeniBatch achieves communication speedups in real-world PPML applications. Figure 12 illustrates the traffic size per iteration for all the PPML applications studied and shows that the traffic size is reduced by about 24× for horizontal PPML applications and 6-7× for vertical PPML applications. With GeniBatch, the total traffic size substantially approaches plain learning. As horizontal PPML does not involve multiplications while vertical PPML requires 1 multiplication (Table 1), the difference in traffic drops well corresponds with Figure 11. The difference in traffic reductions partially explains the performance variations observed in Table 4, as communication cost is a major component in the end-to-end iteration time (Figure 1).

5.3.2 Computation Speedup by GeniBatch Operations

In this section, we analyze the effectiveness of all GeniBatch operations, including encryption/decryption, VecAdd, ScalarMul, HadmrdProd, and InnerSum. We fix the compression ratio to 6, select the CPUs as hardware, and report the elapsed time per 1 million operations with and without (w/o) GeniBatch in Figure 13. We make the following observations.

• For operations in the batch mode, *e.g.*, encryption/decryption, VecAdd, ScalarMul, GeniBatch delivers a speedup



(a) vs. Non-batch PHE **(b)** vs. Plain Distributed Learning

Figure 12. Normalized traffic size (MB) per iteration, where RN, DN, EN denote ResNet50, DenseNet169 and EfficientNetB0, respectively.

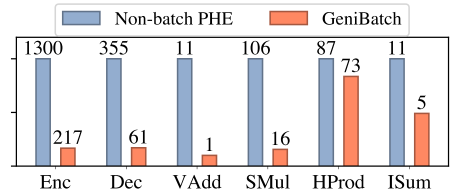


Figure 13. Normalized execution time (s) of 1 million encryption, decryption and vector operations.

similar to the compression ratio, i.e. 6 in this case. The result corresponds with the analysis in §3.3.2 that operations in the batch mode are accelerated by m times if each batch ciphertext contains m slots.

• GeniBatch accelerates operations in the non-batch mode, i.e. HadmrdProd and InnerSum by 1.2× and 2.0× respectively. As discussed in §3.1, operations in the non-batch mode require additional overheads caused by Replica and ShiftAdd, which accounts for the less significant improvements compared to those in the batch mode. However, following the analysis in §3.3.2, GeniBatch is still provably better than non-batch PHE with the dataflow optimization. Therefore, the results on HadmrdProd and InnerSum still complies with our analysis.

5.3.3 Decomposition of End-to-end Iteration Time

As detailed in §5.2, the improvements exhibit variation across different PPML applications. In this section, we probe into the reasons underlying the variation. We select two applications, ResNet50 in horizontal PPML and VLR in vertical PPML, and decompose the iteration time on both parties into two parts: cryptographic operation and idle time (wherein parties wait for the intermediate results from other parties) and plot the decomposition results in Figure 14. Our analysis is as follows.

GeniBatch for horizontal PPML. GeniBatch achieves significant speedups (over 20×) in horizontal applications (Figures 14a and 14b) for two reasons. First, no multiplicative operators are required, enabling a high compression ratio (Figure 11) and a significant advantage in communication. Second, the cryptographic operations involved are Encrypt/Decrypt, VecAdd, and ScalarMul, all of which can be operated in the batch mode with a speedup similar to the compression ratio (25×).

GeniBatch for Vertical PPML. We take VLR (Figures 14c and 14d) as an example to analyze why GeniBatch is not as effective on vertical PPML compared to horizontal PPML. The reasons are two-fold. On the one hand, multiplicative operators (*i.e.*, HadmrdProd) are involved in VLR, resulting in a lower compression ratio (6 compared to 25 shown in Figure 11) and less improvement in communication. On the other hand, HadmrdProd occupies the majority of computation time, on which GeniBatch fails to accelerate as much as operators in batch mode (Figure 13).

Speedup variation across PPML applications. The variation in speedup can be attributed to two factors: (a) the compression ratio, which depends on the number of homomorphic operations; and (b) the percentage of communication and operations in batch mode. GeniBatch speedups approach the compression ratio as the percentage increases. Therefore, GeniBatch is more effective in horizontal PPML (compressed 25× and the percentage is over 95%) and is relatively less effective in vertical PPML (compressed 6-7× and the percentage is around 70%). In vertical PPML, the percentage also varies with the dataset (traffic size increases with sample size) thus GeniBatch performs better on SUSY (5M samples) than the synthetic dataset (1M samples).

5.3.4 Effectiveness of Dataflow Optimization

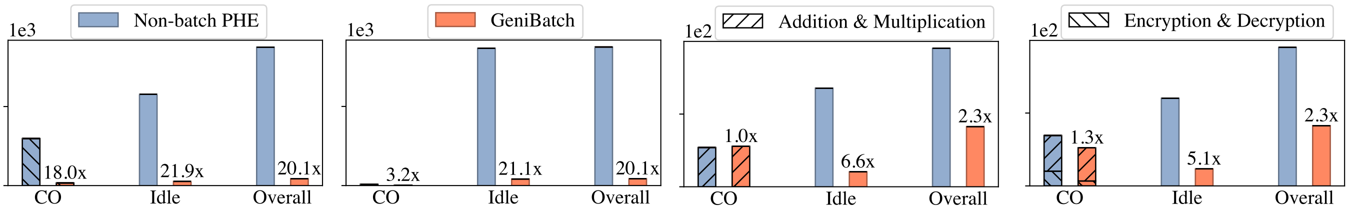
Next, we analyze how the dataflow optimization in §3.3 improves the end-to-end performance of real-world PPML applications. As discussed in §3.3, the dataflow optimization primarily reduces the number of ShiftAdd operations involved in replica modes. Therefore, horizontal PPML applications, where batch mode operations (ScalarMul and VectorAdd) suffice, do not require dataflow optimization. We select two representative vertical PPML applications, VLR and SBT, and report their average iteration time with and without dataflow optimization in Table 5. As observed, the dataflow optimization is effective for both VLR and SBT, delivering additional speedups of 0.9× and 0.4×, respectively.

	Non-batch PHE	GeniBatch w/o optimization	GeniBatch
VLR	475.7 (-)	332.0 (1.43×)	206.8 (2.30×)
SBT	435.1 (-)	367.2 (1.18×)	274.2 (1.58×)

Table 5. Iteration time (Speedup) of GeniBatch with and without dataflow optimization on VLR and SBT.

5.3.5 Scalability

Finally, we evaluate the scalability of GeniBatch by increasing participants to 4, 8 and 16. We select two representative PPML applications, ResNet50 with CIFAR10 for horizontal and VLR with SUSY for vertical. To be specific, we divide the original dataset randomly and assign each participant one copy. For example, each party randomly owns 25,000 samples of the CIFAR10 in ResNet50 and 9 features of the SUSY in VLR.



(a) ResNet50 Party A&B **(b)** ResNet50 Central Server **(c)** VLR Party A **(d)** VLR Party B

Figure 14. Decomposition of iteration time (s) for ResNet50 (horizontal PPML) and VLR (vertical PPML). CO is the acronym for cryptographic operations.

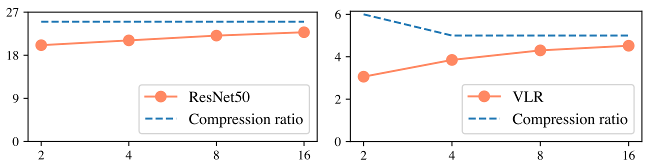


Figure 15. The speedups of GeniBatch with 2, 4, 8, and 16 participants.

We report the average speedups and compression ratio of GeniBatch (compared to non-batch PHE) in Figure 15 and observe that the speedups increase with #participants. The reason we have mentioned in §5.3.3: the percentage of communication and batch-mode operation increases with #participants, leading to the gradual approach of speedups towards the compression ratio. In the VLR setting, the compression ratio decreases from six to five, which affects the maximum theoretical speedup. However, this degradation is logarithmic, as shown in Figure 11. The next degradation occurs when the number of participants reaches 1024, whereas more than 4 participants rarely occur in vertical PPML.

6 Related Work

Basic technologies for PPML. In §2.1, we have discussed PHE [47] and DP [8] as the dominant building blocks for PPML applications. Besides, Secret Sharing (SS) [21, 45], Oblivious Transfer (OT) [49], and Fully Homomorphic Encryption (FHE) [20] are also applicable in PPML. For example, SecureNN [55] and Falcon [56] applied three-party and four-party SS protocols to enable secure neural network training; MiniONN [40] and GAZELLE [32] leveraged OT or combined OT with HE to construct privacy-preserving model inference frameworks; Sphinx [53] proposed a new FHE-with-DP protocol for secure online learning. However, SS, OT, and FHE are not efficient enough for geo-distributed PPML (which involves multiple datacenters) due to their high communication overhead and computation complexity [50, 70], while DP incurs model accuracy loss as we described on §2.1. This motivates us to focus on PHE optimizations in this work.

Batch optimizations for HE. In addition to batch optimization [12, 39, 63, 68] discussed in §2.3.2, VF2Boost [25] packs ciphertexts with batching after computation, does not support subsequent cryptographic operations; vector operations over ciphertexts in Gazelle [32] and SEAL [5] are not applicable, as their underlying cryptosystem is FHE which is inconsistent with PHE in mathematical properties.

Computing optimizations for PHE. To address the high computational complexity of PHE, several works have attempted to fully utilize multi-core CPUs [44], GPUs [18] and FPGAs [66, 69] to accelerate HE computation. GeniBatch is orthogonal to these works and we have implemented GeniBatch atop HAFLO (GPU-based acceleration) to further improve the performance of PPML.

Framework optimizations for ML QSGD [11] and Terngrad [59] leverage quantization to compress gradients with lower bits to reduce network traffic for distributed ML; however, they cannot address data inflation in PHE as quantized numbers still expand to large numbers after encryption. Works for model compression [51, 58] which reduces communication overhead with sparsification or sketching, and works for relaxed synchronization [30, 37, 41] which reduces communication frequency with stale information are orthogonal to our work. GeniBatch can be integrated with the above works as it imposes no constraints on input data and synchronization schemes.

Network optimizations for ML. Distributed ML can boost its performance via various domain-specific network optimizations, both intra-datacenter [36, 42, 48, 60] and inter-datacenter [35, 67]. For instance, DSA [57] utilized in-network devices (*i.e.* P4 switches) to accelerate parameter aggregation; [67] proposed a new congestion control protocol for cross-datacenter networks, reducing network latency. Since GeniBatch is a bit-level optimization that does not change packet headers, it can be easily integrated into the above network devices and protocols.

7 Conclusion

This paper presented GeniBatch, a batch compiler that translates a PHE program with general vector operations into a cross-silo PPML program with batching optimization. GeniBatch includes a Numpy-like frontend that can be used to write advanced programs with little programming effort and includes an optimizing compiler that generates correct and efficient dataflow of PPML applications. Extensive experiments have shown that GeniBatch is a viable solution to improve end-to-end performance for various cross-silo PPML applications from 1.59× to 22.6×.

Acknowledgments

We thank the anonymous reviewers for their constructive suggestions. This work is supported by the Hong Kong RGC TRS T41-603/20-R, GRF-16213621, ITF ACCESS project, the Key-Area R&D Program of Guangdong (2021B0101400001), the China NSFC 62062005, and the Turing AI Computing Cloud (TACC) [61]. Junxue Zhang and Kai Chen are the corresponding authors.

References

[1] Fate (federated ai technology enabler). https://fate.fedai.org/, 2019.

[2] Fedlearner. https://github.com/bytedance/fedlearner, 2019.

[3] Tf encrypted. https://github.com/tf-encrypted/tf-encrypted, 2019.

[4] Federatedml (federated machine learning). https://github.com/FederatedAI/FATE/tree/master/python/federatedml, 2019.

[5] Microsoft seal. https://github.com/Microsoft/SEAL, 2022.

[6] California consumer privacy act (ccpa). https://oag.ca.gov/privacy/ccpa, 2018.

[7] tc(8)–linux manual page. https://man7.org/linux/man-pages/man8/tc.8.html, 2022.

[8] Martin Abadi, Andy Chu, Ian Goodfellow, H Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In *CCS*, 2016.

[9] Martín Abadi, Andy Chu, Ian J. Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 308–318. ACM, 2016.

[10] Mohammad Al-Rubaie and J Morris Chang. Privacy-preserving machine learning: Threats and solutions. *IEEE Security & Privacy*, 17(2):49–58, 2019.

[11] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. Qsgd: Communication-efficient sgd via gradient quantization and encoding. *Advances in neural information processing systems*, 30, 2017.

[12] Yoshinori Aono, Takuya Hayashi, Lihua Wang, Shiho Moriai, et al. Privacy-preserving deep learning via additively homomorphic encryption. *IEEE Transactions on Information Forensics and Security*, 13(5):1333–1345, 2017.

[13] Eugene Bagdasaryan, Omid Poursaeed, and Vitaly Shmatikov. Differential privacy has disparate impact on model accuracy. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 15453–15462, 2019.

[14] Pierre Baldi, Peter Sadowski, and Daniel Whiteson. Searching for exotic particles in high-energy physics with deep learning. *Nature communications*, 5(1):4308, 2014.

[15] Chaochao Chen, Jun Zhou, Li Wang, Xibin Wu, Wenjing Fang, Jin Tan, Lei Wang, Alex X Liu, Hao Wang, and Cheng Hong. When homomorphic encryption marries secret sharing: Secure large-scale sparse logistic regression and applications in risk control. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pages 2652–2662, 2021.

[16] Jonathan H Chen and Steven M Asch. Machine learning and prediction in medicine—beyond the peak of inflated expectations. *The New England journal of medicine*, 376(26):2507, 2017.

[17] Kewei Cheng, Tao Fan, Yilun Jin, Yang Liu, Tianjian Chen, Dimitrios Papadopoulos, and Qiang Yang. Secureboost: A lossless federated learning framework. *IEEE Intelligent Systems*, 36(6):87–98, 2021.

[18] Xiaodian Cheng, Wanhang Lu, Xinyang Huang, Shuihai Hu, and Kai Chen. Haflo: Gpu-based acceleration for federated logistic regression. *arXiv preprint arXiv:2107.13797*, 2021.

[19] Yong Cheng, Yang Liu, Tianjian Chen, and Qiang Yang. Federated learning for privacy-preserving ai. *Communications of the ACM*, 63:33–36, 2020.

[20] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *ASIACRYPT*, 2017.

[21] Daniel Demmler, Thomas Schneider, and Michael Zohner. Aby-a framework for efficient mixed-protocol secure two-party computation. In *NDSS*, 2015.

[22] Olga Fink, Torbjørn Netland, and Stefan Feuerriegelc. Artificial intelligence across company borders. *Communications of the ACM*, 65:34–36, 2021.

[23] Farshad Firouzi, Bahareh J. Farahani, Mojtaba Barzegari, and Mahmoud Daneshmand. Ai-driven data monetization: The other face of data in iot-based smart and connected health. *IEEE Internet Things J.*, 9(8):5581–5599, 2022.

[24] Kyle Fritchman, Keerthanaa Saminathan, Rafael Dowsley, Tyler Hughes, Martine De Cock, Anderson C. A. Nascimento, and Ankur Teredesai. Privacy-preserving scoring of tree ensembles: A novel framework for AI in healthcare. In Naoki Abe, Huan Liu, Calton Pu, Xiaohua Hu, Nesreen K. Ahmed, Mu Qiao, Yang Song, Donald Kossmann, Bing Liu, Kisung Lee, Jiliang Tang, Jingrui He, and Jeffrey S. Saltz, editors, *IEEE International Conference on Big Data (IEEE BigData 2018), Seattle, WA, USA, December 10-13, 2018*, pages 2413–2422. IEEE, 2018.

[25] Fangcheng Fu, Yingxia Shao, Lele Yu, Jiawei Jiang, Huanran Xue, Yangyu Tao, and Bin Cui. Vf2boost: Very fast vertical federated gradient boosting for cross-enterprise learning. In *Proceedings of the 2021 International Conference on Management of Data*, pages 563–576, 2021.

[26] Michelle Goddard. The eu general data protection regulation (gdpr): European regulation that has a global impact. *International Journal of Market Research*, 59(6):703–705, 2017.

[27] Nils Gura, Arun Patel, Arvinderpal Wander, Hans Eberle, and Sheueling Chang Shantz. Comparing elliptic curve cryptography and RSA on 8-bit cpus. In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*, volume 3156 of *Lecture Notes in Computer Science*, pages 119–132. Springer, 2004.

[28] Stephen Hardy, Wilko Henecka, Hamish Ivey-Law, Richard Nock, Giorgio Patrini, Guillaume Smith, and Brian Thorne. Private federated learning on vertically partitioned data via entity resolution and additively homomorphic encryption. *arXiv preprint arXiv:1711.10677*, 2017.

[29] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[30] Kevin Hsieh, Aaron Harlap, Nandita Vijaykumar, Dimitris Konomis, Gregory R Ganger, Phillip B Gibbons, and Onur Mutlu. Gaia:{Geo-Distributed} machine learning approaching {LAN} speeds. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 629–647, 2017.

[31] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[32] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. {GAZELLE}: A low latency framework for secure neural network inference. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1651–1669, 2018.

[33] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[34] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[35] Fan Lai, Xiangfeng Zhu, Harsha V Madhyastha, and Mosharaf Chowdhury. Oort: Efficient federated learning via guided participant selection. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, pages 19–35, 2021.

[36] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. {ATP}: In-network aggregation for multi-tenant learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 741–761, 2021.

[37] Xiangru Lian, Yijun Huang, Yuncheng Li, and Ji Liu. Asynchronous parallel stochastic gradient for nonconvex optimization. *Advances in neural information processing systems*, 28, 2015.

[38] Changchang Liu, Supriyo Chakraborty, and Dinesh Verma. Secure model fusion for distributed learning using partial homomorphic encryption. In *Policy-Based Autonomic Data Governance*, pages 154–179. Springer, 2019.

[39] Changchang Liu, Supriyo Chakraborty, and Dinesh Verma. Secure model fusion for distributed learning using partial homomorphic encryption. In *Policy-Based Autonomic Data Governance*, pages 154–179. Springer, 2019.

[40] Jian Liu, Mika Juuti, Yao Lu, and Nadarajah Asokan. Oblivious neural network predictions via minionn transformations. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 619–631, 2017.

[41] WANG Luping, WANG Wei, and LI Bo. Cmfl: Mitigating communication overhead for federated learning. In *2019 IEEE 39th international conference on distributed computing systems (ICDCS)*, pages 954–964. IEEE, 2019.

[42] Yiqing Ma, Hao Wang, Yiming Zhang, and Kai Chen. Autobyte: Automatic configuration for optimal communication scheduling in dnn training. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*, pages 760–769. IEEE, 2022.

[43] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Artificial intelligence and statistics*, pages 1273–1282. PMLR, 2017.

[44] Zhaoe Min, Geng Yang, and Jingqi Shi. A privacy-preserving parallel and homomorphic encryption scheme. *Open Physics*, 15(1):135–142, 2017.

[45] Payman Mohassel and Peter Rindal. Aby3: A mixed protocol framework for machine learning. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 35–52, 2018.

[46] Valeria Nikolaenko, Udi Weinsberg, Stratis Ioannidis, Marc Joye, Dan Boneh, and Nina Taft. Privacy-preserving ridge regression on hundreds of millions of records. In *2013 IEEE symposium on security and privacy*, pages 334–348. IEEE, 2013.

[47] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *International conference on the theory and applications of cryptographic techniques*, pages 223–238. Springer, 1999.

[48] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed dnn training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 16–29, 2019.

[49] Michael O Rabin. How to exchange secrets with oblivious transfer. *Cryptology ePrint Archive*, 2005.

[50] Zhenghang Ren, Xiaodian Cheng, Mingxuan Fan, Junxue Zhang, and Cheng Hong. Communication efficient secret sharing with dynamic communication-computation conversion. In *IEEE INFOCOM 2023-IEEE Conference on Computer Communications*, pages 1–10. IEEE, 2023.

[51] Daniel Rothchild, Ashwinee Panda, Enayat Ullah, Nikita Ivkin, Ion Stoica, Vladimir Braverman, Joseph Gonzalez, and Raman Arora. Fetchsgd: Communication-efficient federated learning with sketching. In *International Conference on Machine Learning*, pages 8253–8265. PMLR, 2020.

[52] Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning*, pages 6105–6114. PMLR, 2019.

[53] Han Tian, Chaoliang Zeng, Zhenghang Ren, Di Chai, Junxue Zhang, Kai Chen, and Qiang Yang. Sphinx: Enabling privacy-preserving online learning over the cloud. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 2487–2501. IEEE, 2022.

[54] Tijmen Tieleman, Geoffrey Hinton, et al. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

[55] Sameer Wagh, Divya Gupta, and Nishanth Chandran. Securenn: 3-party secure computation for neural network training. *Proc. Priv. Enhancing Technol.*, 2019(3):26–49, 2019.

[56] Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. Falcon: Honest-majority maliciously secure framework for private deep learning. *arXiv preprint arXiv:2004.02229*, 2020.

[57] Hao Wang, Yuxuan Qin, ChonLam Lao, Yanfang Le, Wenfei Wu, and Kai Chen. Preemptive switch memory usage to accelerate training jobs with shared in-network aggregation.

[58] Jianqiao Wangni, Jialei Wang, Ji Liu, and Tong Zhang. Gradient sparsification for communication-efficient distributed optimization. *Advances in Neural Information Processing Systems*, 31, 2018.

[59] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Terngrad: Ternary gradients to reduce communication in distributed deep learning. *Advances in neural information processing systems*, 30, 2017.

[60] Jiacheng Xia, Gaoxiong Zeng, Junxue Zhang, Weiyan Wang, Wei Bai, Junchen Jiang, and Kai Chen. Rethinking transport layer design for distributed machine learning. In *Proceedings of the 3rd Asia-Pacific Workshop on Networking 2019*, pages 22–28, 2019.

[61] Kaiqiang Xu, Xinchen Wan, Hao Wang, Zhenghang Ren, Xudong Liao, Decang Sun, Chaoliang Zeng, and Kai Chen. Tacc: A full-stack cloud computing infrastructure for machine learning tasks. *arXiv preprint arXiv:2110.01556*, 2021.

[62] Runhua Xu, Nathalie Baracaldo, and James Joshi. Privacy-preserving machine learning: Methods, challenges and directions. *CoRR*, abs/2108.04417, 2021.

[63] Wuxing Xu, Hao Fan, Kaixin Li, and Kai Yang. Efficient batch homomorphic encryption for vertically federated xgboost. *arXiv preprint arXiv:2112.04261*, 2021.

[64] Liu Yang, Di Chai, Junxue Zhang, Yilun Jin, Leye Wang, Hao Liu, Han Tian, Qian Xu, and Kai Chen. A survey on vertical federated learning: From a layered perspective. *arXiv preprint arXiv:2304.01829*, 2023.

[65] Qiang Yang, Yang Liu, Tianjian Chen, and Yongxin Tong. Federated machine learning: Concept and applications. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 10(2):1–19, 2019.

[66] Zhaoxiong Yang, Shuihai Hu, and Kai Chen. Fpga-based hardware accelerator of homomorphic encryption for efficient federated learning. *arXiv preprint arXiv:2007.10560*, 2020.

[67] Gaoxiong Zeng, Wei Bai, Ge Chen, Kai Chen, Dongsu Han, Yibo Zhu, and Lei Cui. Congestion control for cross-datacenter networks. In *2019 IEEE 27th International Conference on Network Protocols (ICNP)*, pages 1–12. IEEE, 2019.

[68] Chengliang Zhang, Suyi Li, Junzhe Xia, Wei Wang, Feng Yan, and Yang Liu. {BatchCrypt}: Efficient homomorphic encryption for {Cross-Silo} federated learning. In *2020 USENIX annual technical conference (USENIX ATC 20)*, pages 493–506, 2020.

[69] Junxue Zhang, Xiaodian Cheng, Wei Wang, Liu Yang, Jinbin Hu, and Kai Chen. {FLASH}: Towards a high-performance hardware acceleration architecture for cross-silo federated learning. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1057–1079, 2023.

[70] Junxue Zhang, Xiaodian Cheng, Liu Yang, Jinbin Hu, Ximeng Liu, and Kai Chen. Sok: Fully homomorphic encryption accelerators. *arXiv preprint arXiv:2212.01713*, 2022.

[71] Benjamin Zi Hao Zhao, Mohamed Ali Kâafar, and Nicolas Kourtellis. Not one but many tradeoffs: Privacy vs. utility in differentially private machine learning. In Yinqian Zhang and Radu Sion, editors, *CCSW'20, Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop, Virtual Event, USA, November 9, 2020*, pages 15–26. ACM, 2020.

[72] Jun Zhou, Xiaolong Li, Peilin Zhao, Chaochao Chen, Longfei Li, Xinxing Yang, Qing Cui, Jin Yu, Xu Chen, Yi Ding, et al. Kunpeng: Parameter server based distributed learning systems and its applications in alibaba and ant financial. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1693–1702, 2017.