Computers, Materials & Continua

Tech Science Press

DOI: 10.32604/cmc.2024.049233

ARTICLE

EG-STC: An Efficient Secure Two-Party Computation Scheme Based on Embedded GPU for Artificial Intelligence Systems

Zhenjiang Dong[1], Xin Ge[1], Yuehua Huang[1], Jiankuo Dong[1] and Jiang Xu[2,*]

[1]School of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing, 210023, China

[2]School of Computer, Nanjing University of Information Science and Technology, Nanjing, 210044, China

*Corresponding Author: Jiang Xu. Email: 01136@nuist.edu.cn

Received: 31 December 2023 Accepted: 28 March 2024 Published: 20 June 2024

ABSTRACT

This paper presents a comprehensive exploration into the integration of Internet of Things (IoT), big data analysis, cloud computing, and Artificial Intelligence (AI), which has led to an unprecedented era of connectivity. We delve into the emerging trend of machine learning on embedded devices, enabling tasks in resource-limited environments. However, the widespread adoption of machine learning raises significant privacy concerns, necessitating the development of privacy-preserving techniques. One such technique, secure multi-party computation (MPC), allows collaborative computations without exposing private inputs. Despite its potential, complex protocols and communication interactions hinder performance, especially on resource-constrained devices. Efforts to enhance efficiency have been made, but scalability remains a challenge. Given the success of GPUs in deep learning, leveraging embedded GPUs, such as those offered by NVIDIA, emerges as a promising solution. Therefore, we propose an Embedded GPU-based Secure Two-party Computation (EG-STC) framework for Artificial Intelligence (AI) systems. To the best of our knowledge, this work represents the first endeavor to fully implement machine learning model training based on secure two-party computing on the Embedded GPU platform. Our experimental results demonstrate the effectiveness of EG-STC. On an embedded GPU with a power draw of 5 W, our implementation achieved a secure two-party matrix multiplication throughput of 5881.5 kilo-operations per millisecond (kops/ms), with an energy efficiency ratio of 1176.3 kops/ms/W. Furthermore, leveraging our EG-STC framework, we achieved an overall time acceleration ratio of 5–6 times compared to solutions running on server-grade CPUs. Our solution also exhibited a reduced runtime, requiring only 60% to 70% of the runtime of previously best-known methods on the same platform. In summary, our research contributes to the advancement of secure and efficient machine learning implementations on resource-constrained embedded devices, paving the way for broader adoption of AI technologies in various applications.

KEYWORDS

Secure two-party computation; embedded GPU acceleration; privacy-preserving machine learning; edge computing

1 Introduction

With the rapid development of information technology, private data is experiencing explosive growth on the internet, termed as the new oil of the data age. Artificial Intelligence (AI) has emerged as a significant tool to tap into the potential of this data. While centralized machine learning based on cloud computing has been successful in various fields, it faces challenges in latency, data privacy, and model confidentiality. Edge computing [1] and edge learning have emerged to address these issues. They utilize embedded devices to execute machine learning tasks locally, significantly reducing communication latency. According to a GSMA report [2], the global number of Internet of Things (IoT) connections is expected to reach 24.6 billion by 2025. The integration of IoT, big data, cloud computing, and AI has brought unprecedented connectivity and data utilization. Massive information generated by smart devices, when analyzed, reveals patterns and trends providing deep insights into user behavior and industrial development. Utilizing embedded devices for machine learning has become an increasingly popular trend, enabling the execution of machine learning tasks in resource-constrained environments. Edge computing is a hierarchical distributed cloud-edge-device AI computing architecture. Data is processed locally or at the closest edge computing server, requiring only essential model parameter updates to the cloud server during local execution of machine learning model training and prediction, thus greatly reducing communication delays. As machine learning and neural network applications become more widespread, the use of private data involving sensitive information and industrial details for training models has intensified public concerns about privacy breaches, leading to stringent data protection laws such as the European General Data Protection Regulation (GDPR) [3].

Against this backdrop, to ensure the safety of AI tasks, a variety of Privacy Enhancing Technologies [4–6] have been proposed. Among them, Secure Multi-Party Computation (MPC/SMPC) [6–9] is an emerging cryptographic technology that allows multiple participants to engage in collaborative computation while preserving the privacy of their individual data. The essence of MPC lies in its underlying encryption protocols that distribute the computational load among parties, ensuring that no single party can access the input information of others. During the process of protecting machine learning training data and models, MPC ensures that while information is shared, individual privacy is not compromised, meaning that during the computation, each party only knows the result and remains unaware of input data of other parties. However, the performance of MPC protocols is often less than ideal due to their involvement in complex computations and frequent communication between participating parties. Currently, enhancing the performance of MPC is focused mainly on two directions: On one hand, optimizing algorithms and reducing the number of communications in secure protocols to decrease computational and communication costs; on the other hand, employing hardware acceleration technologies, such as dedicated encryption processors, to improve computational efficiency.

Currently, many research efforts [10,11] are exploring algorithmic optimization [12–15] and performance enhancement of MPC protocols, aiming to advance their practical application through efficient algorithms and hardware acceleration. Recently, in response to its computationally intensive nature, some innovative studies have begun to utilize Graphics Processing Units (GPUs) to accelerate the computational process of MPC. Frederiksen et al. [16] presented a new protocol for maliciously secure two-party computation using a GPU as a massive SIMD (Single Instruction, Multiple Data) device. Chen et al. [17] proposed the first GPU-based secure machine learning framework. Srinivasan et al. [18] has implemented a method to accelerate convolution operations using GPUs, then continuing the remaining computations on CPUs. Tan et al. [19] has customized specific secure three-party computation protocols for its applications, adapting to the parallel processing capabilities of

GPUs. Watson et al. [20] utilized the CUTLASS library to implement parallel computation of general matrix multiplication (GEMM) and convolutional kernels, significantly boosting the computational efficiency of MPC on GPUs. Meng et al. [21] explored distributing the computation of a single party among multiple GPU.

However, these implementations are mostly focused on server-level or desktop-level GPUs, which typically possess higher computational capabilities and more resources, and thus are not entirely suitable for resource-constrained and low-power edge computing environments. Edge computing often relies on embedded GPUs, which to meet the requirements of low power consumption and cost-effectiveness, usually trim some advanced features such as tensor cores. Therefore, while the existing GPU-accelerated MPC solutions have made significant strides in improving computational efficiency, they are not directly applicable to embedded GPU environments. With the continuous development of IoT technology, high-performance, low-power embedded GPUs are being widely applied in edge computing devices. These devices, despite their low cost, have computing capabilities comparable to desktop CPUs, becoming a vital force in supporting edge intelligence. Moreover, many research efforts [22–24] have focused on leveraging the parallel capabilities of embedded GPUs to accelerate cryptographic algorithms. Hence, utilizing embedded GPUs to accelerate MPC-based AI tasks on edge devices is a feasible and practical approach.

1.1 Contributions and Paper Organization

The widespread use of machine learning raises privacy concerns, prompting the development of privacy-preserving techniques. MPC is one such technique, allowing collaborative computations without exposing private inputs. Despite MPC having potential, its complex protocols and communication interactions lead to lower performance, especially on resource-constrained devices. The point of this article is to apply the embedded GPU to the acceleration of the Secure Two-party Computation Scheme, called embedded GPU-based Secure Two-party Computation (EG-STC). Our various optimization methods offer more efficient and secure acceleration of machine learning model training compared to other platforms.

The contributions of this paper are fourfold:

- Firstly, we analyze the complexity of each calculation step and categorize them into two parts: Simple calculations and complex calculations. Simple calculations such as addition and subtraction can be performed on the CPU, and complex calculations such as multiplication can be performed on the GPU, so that it can be effectively to reduce the overhead caused by partial data communication, only part of the data needs to be transmitted between the CPU and GPU for subsequent operations.
- Secondly, we will first compress the matrix data that needs to be transmitted between the CPU and the GPU before transmitting it. Before transmitting, we first determine whether the matrix change value that needs to be transmitted is sparse. If it is sparse, we only need to transmit the change value and use compressed sparse rows format (CSR) to store it, and then transmit the compressed changed values, or directly transmit the original data if it is not sparse.
- Thirdly, we further improve performance by overlapping communication and calculation. Through observation, we found that the processing of subsequent layers depends on the forward propagation of the current layer, so the forward propagation reconstruct step of the next layer cannot be combined with the forward propagation GPU operation of the current layer. But the reconstruct step in back propagation does not need to wait for subsequent layers, so it can be performed together with the propagation of the next layer in the pipeline. For the back

propagation reconstruct step, part of the calculation depends on the forward propagation GPU operation step of the same layer, so these calculations can be kept in the back propagation reconstruct step, and the other part of the calculation depends on the next layer The result is therefore assigned to the back propagation GPU operation step. Therefore, the forward propagation reconstruct step of the current layer can overlap with the backward propagation reconstruct step of the previous layer.

- Finally, we have further optimized the calculation on the CPU and the calculation on the GPU. We use the advanced random number generator MT19937, which can generate random numbers concurrently in multiple threads and ensure thread safety. According to the structural characteristics of embedded GPUs, we experimentally select the fastest batch processing block size and thread and block configuration method.

The rest of our paper is organized as follows. Section Preliminaries explains some preliminaries including some cryptographic primitives, secure edge learning, and GPU acceleration and so on. Section Methodology describes the proposed the insights and strategies of our system implementation. Section Performance Evaluation performs our optimized implementation and compares it with related works. Section Conclusion concludes the paper.

2 Preliminaries

2.1 Cryptographic Primitives

Threat Model. According to the behavior of adversaries, security models of MPC protocols can be categorized into semi-honest, malicious, and covert security models. In the semi-honest security model, an adversary follows the protocol description but attempts to infer private data information of other participants from the protocol output records. In the malicious security model, an adversary is not bound by the protocol constraints and can execute various attacks, such as sending arbitrary messages, prematurely interrupting protocol execution, or inputting incorrect data sets. The covert security model falls between the semi-honest and malicious security models, presenting a more practically applicable security model that achieves a better balance between security and performance. It can detect malicious behaviors of adversaries with a certain probability. In this paper, our system falls under the semi-honest security model, assuming that all parties (servers) will comply with the protocol execution. The system hardware architecture comprises a CPU as the main processor and an embedded GPU as the coprocessor. All parties communicate with each other through the CPU. Consequently, our system is suitable for use in both local and wide area networks. Through the underlying security protocol implementation provided by our system, we can ensure the security of upper-layer applications or program modules. In this paper, we have successfully accomplished several secure AI tasks, including operations involving convolutional neural networks.

Secret Sharing. Secret sharing (SS) is one of the basic primitives in MPC, with its concept initially proposed by Shamir [25] and Blakley [26], respectively. Secret sharing involves the input sharing (called *Share( )* function), computation, and secret reconstruction (called *Reconstruction( )* function) phases of secret data. In the threshold secret sharing scheme attributed to Shamir, data holders divide the secret data S into multiple secret shares in some manner, and the original secret data can be reconstructed only when a sufficient number of secret shares are combined. Let n denote the number of participating parties, and t denote a bound on the number of corrupted parties. In the case of $t < n/2$, it is referred to as an honest majority; for $t \geq n/2$, it is termed a dishonest majority. Secret sharing protocols have low overhead, are straightforward to implement, and exhibit high computational efficiency, making them widely applied in MPC protocols. However, their drawbacks

include the need for generating and storing numerous random numbers, high communication costs, and the requirement for point-to-point secure channels for transmitting secret shares.

Additive Secret Sharing. The linear secret sharing (LSSS) MPC protocols primarily employ three secret sharing schemes: Shamir, additive, and replicated. These secret sharing schemes are mainly defined over field F but can also be extended to the ring $\mathbb{Z}_n$, where $n = 2^k$ and $k = 32$ *or* 64. In our work, $k = 64$. For secret data $x \in \mathbb{Z}_n$, let $< x >_i$ represent the secret sharing shares, then $x = \sum_{i=1}^{t} < x >_i$, where $< x >_i \in \mathbb{Z}_n$, and t represents the minimum required number of shares. In our work, we employ the 2-out-of-2 additive secret sharing scheme. For secret data x on the ring, a random number r is chosen such that $r \in \mathbb{Z}_n$, Set $< x >_1 = r$ and $< x >_2 = x - r$, ensuring that $< x >_1 + < x >_2 = x$.

Beaver Multiplicative Triples. Beaver triples [27], introduced in 1991, are pivotal in secure multiplication computations within Multi-Party Computation (MPC) using secret sharing. They enable collaborative computation of the product of private values while preserving privacy. Each triple comprises three elements (a, b, and c), allowing two parties to jointly compute $a \times b = c$ without exchanging sensitive information. The significance of Beaver triples lies in their role in secure multiplication, ensuring confidentiality in scenarios like cryptographic protocols and privacy-preserving data analysis. Their incorporation into MPC schemes enhances the security of multi-party computations, facilitating complex operations while safeguarding the privacy of individual inputs. They typically encompass the following two phases:

(1) Offline Phase (Preprocessing Phase):

In this preparatory stage, random numbers (a, b, and c) are chosen such that $c = a \times b$ within field F. All participants P_1, P_2,..., P_n generate additive secret sharing shares $< a >_i$, $< b >_i$, $< c >_i$, where $1 \leq i \leq n$, and each of $< a >_i$, $< b >_i$, $< c >_i$ are random numbers in field F.

(2) Online Phase:

Each participant P_i possesses additive secret sharing shares $< x >_i$ and $< y >_i$ of their respective secret data x and y. Participant P_i locally computes $< x >_i - < a >_i$ and $< y >_i - < b >_i$ and broadcasts these intermediate values to other participants. Upon receiving these broadcasted shares, the others aggregate them, deriving $x - a$ and $y - b$. They then compute $z_i = (x - a) \times b_i + (y - b) \times a_i + c_i$ using the available shares. Ultimately, in the secret data reconstruction phase, by summing up each z_i and adding $(x - a) \times (y - b)$, they successfully reconstruct $x \times y$.

2.2 Secure Edge Learning

Privacy-Preserving Edge Learning. Privacy-preserving edge learning is an efficient and privacy-focused computing framework with multi-layer collaboration, performing model training and inference at the edge layer. It balances local privacy data processing with global model updating. Locally, it processes data at the source (e.g., edge devices) to train local machine learning models, effectively protecting data privacy. Meanwhile, edge devices exchange local model-related information (like model parameters) with cloud computing centers or other edge devices, using encryption or obfuscation methods for model updates and aggregation, forming a global model. This approach ensures that neither the cloud nor other edge servers can infer the content of local private data, offering robust privacy protection. By conducting neural network model training directly on edge devices, the framework aims to reduce operational latency and mitigate the risks of privacy data leakage. In model design, reducing neural network parameters is crucial for devices with limited computing power and memory to achieve high precision and minimize memory use and latency. For hardware acceleration, integrating resources such as GPUs and CPUs, as demonstrated by NVIDIA [28] GPUs,

along with parallel programming platforms like CUDA [29], maximizes the effectiveness of hardware acceleration.

2.3 GPU Acceleration

NVIDIA Jetson Nano. The NVIDIA Jetson Nano [30] is a powerful single-board computer launched by NVIDIA, specifically designed for edge computing and AI applications. As shown in Fig. 1, it displays the main components of the Nano. The Nano is built-in with a quad-core ARM Cortex-A57 CPU, sharing 4GB LPDDR4 memory with the GPU, ensuring a balanced computing environment for various applications. The Nano utilizes NVIDIA Maxwell GPU architecture, featuring one Streaming Multiprocessor (SM) with data parallel processing capabilities and containing 128 CUDA cores. NVIDIA provides a comprehensive software stack, including the JetPack SDK, which contains CUDA-X AI libraries for accelerated computing. Additionally, the Nano is designed with power efficiency in mind, consuming only a few watts of power while delivering exceptional AI processing capabilities. The Nano is widely applicable due to its low power consumption and compact size, making it particularly suitable for deployment in edge devices such as smart cameras, drones, and other IoT devices requiring on-device AI processing.

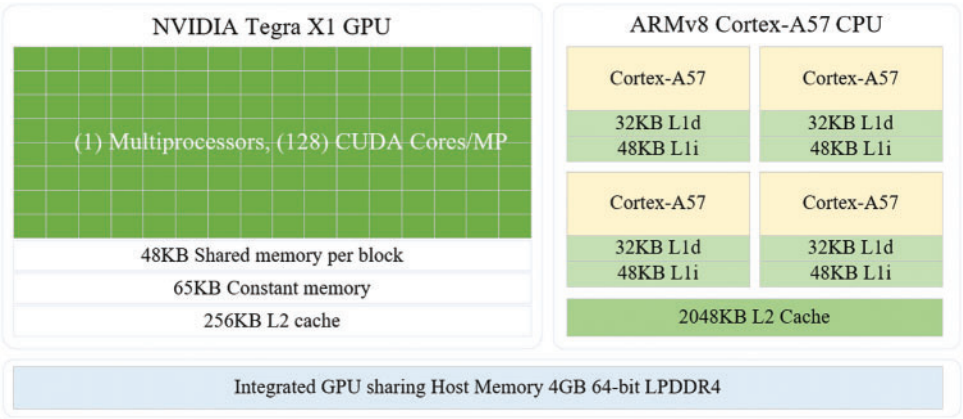


Figure 1: Main components of NVIDIA Jetson Nano

CUDA. CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model by NVIDIA. It allows developers to utilize NVIDIA GPUs parallel computing capabilities for accelerating general computing tasks. This architecture adopts the SIMT (Single Instruction, Multiple Threads) model, breaking down large-scale problems into smaller ones for simultaneous GPU processing. The many processing units of GPU execute the same instruction concurrently, enhancing computational performance. CUDA includes memory types like global, shared, and registers. Optimizing memory access patterns in GPU memory hierarchy enhances performance. CUDA supports languages such as C/C++, Java, and Python. NVIDIA provides tools like the CUDA Toolkit, CUDA compiler (nvcc), and runtime libraries for debugging, performance analysis, and optimization on GPUs. CUDA enables efficient GPU parallel code writing, widely used in scientific computing and deep learning, offering computational support for a broad range of applications. Intensive tasks like matrix operations and numerical simulations see significant performance improvements through CUDA.

The Communication of GPGPUs. GPGPU communication is an integral aspect of heterogeneous computing, where both CPUs and GPUs collaborate rather than operating in isolation. In this

paradigm, a single GPU alone cannot handle all computational tasks independently; instead, the CPU orchestrates and schedules these operations. The CPU, termed the host, and the GPU, providing acceleration, known as the device, each possess their dedicated DRAM and are interconnected via the PCIe bus. The GPU's storage space is referred to as VRAM (Video RAM). Notably, communication between these devices with GPUs necessitates involving the CPU in data transfers.

In our research, we employ MPI (Message Passing Interface) to manage communication between parallel programs. MPI, a standard and library for developing parallel applications, streamlines communication by exchanging messages between distinct processes, enabling collaboration among multiple processes. It furnishes a suite of standard library functions catering to message passing, process management, and collective communication. Each process operates within its own address space and communicates through a unique rank identifier. MPI accommodates parallel computing with numerous processes, capable of running on the same or different networked machines. The library provides essential message-passing functions such as *MPI_Send* and *MPI_Recv* for point-to-point communication, further supporting asynchronous communication and non-blocking operations to augment performance and efficiency. Widely adopted in distributed computing environments, MPI plays a pivotal role in large-scale, high-performance parallel computing.

2.4 Matrix Compression

CSR. Compressed Sparse Row (CSR) is a sparse matrix storage format designed to efficiently represent matrices with a significant number of zero elements. The fundamental idea behind CSR is to store only the non-zero values of the matrix along with their corresponding column indices, while the row information is represented implicitly through a row pointer array. The key steps for implementing CSR involve extracting the non-zero elements and their column indices, computing row pointers to denote the starting position of each row in the value and column index arrays, and storing these arrays compactly. This format is particularly advantageous for sparse matrices, offering reduced storage requirements and more efficient matrix-vector multiplication operations. The CSR representation is widely used in numerical computations and linear algebra algorithms, contributing to improved computational efficiency when dealing with large, sparse datasets.

3 Insights and Methodology

Multi-party computing divides the data into multiple servers. Each server contains only a part of the encrypted data to prevent data leakage. Unfortunately, the more participants, the more obvious the performance decline, which is due to the increased cost of computing and communication costs. In recent years, there have been many work studies applying GPU in high performance computing and machine learning. As the most widely used accelerator in machine learning, GPU has been used for the performance acceleration of two-party computation. However, applying GPUs to machine learning training in secure multi-party computation introduces potential challenges, including increased data transfer and communication overhead, adaptability of algorithms to GPU architectures, concerns regarding privacy and security and so on. To address these issues, this section details a holistic approach that considers factors such as algorithmic complexity, secure communications, and potential information leakage risks, emphasizing a careful balance between performance gains and security considerations. The layer architecture of our scheme is shown in Fig. 2. The device layer abstracts GPU-specific code, the protocol layer implements various MPC protocols along with their secret sharing schemes and adversary models, while the application layer employs these protocols for high-level computations in an undifferentiated manner.

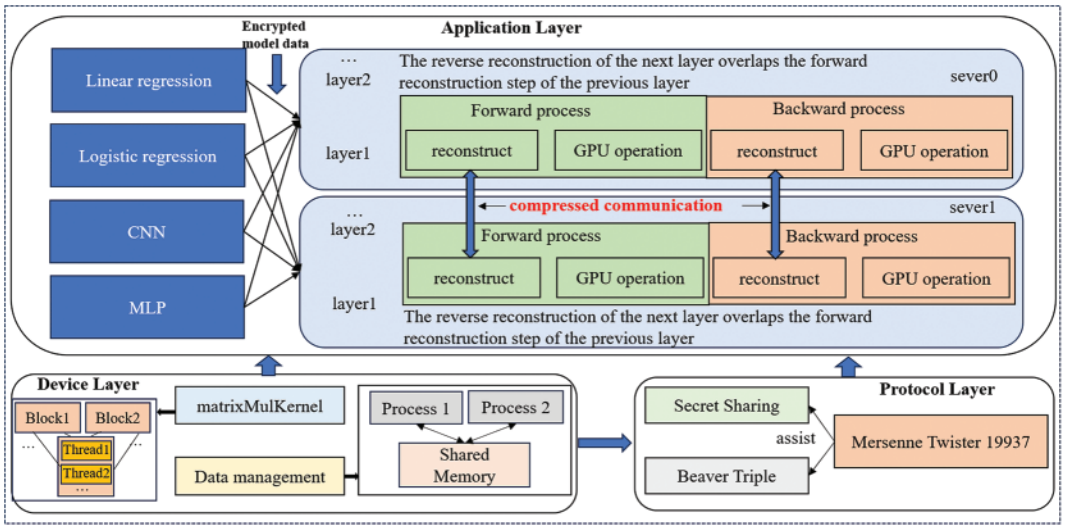


Figure 2: The layer architecture of EG-STC

3.1 Optimizing Two-Party Interactions

3.1.1 Two-Party Calculation Optimization

For safe machine learning details based on two calculation, we chose to illustrate the approach using the machine learning method of Multi-Layer Perceptron (MLP) neural networks and Modified National Institute of Standards and Technology database (MNIST) dataset as an example. We can observe the offline stage data distribution step and the last calculation step spent the most time, so we can try to use GPU to accelerate these steps. Below are details on how we can leverage GPU acceleration for these specific steps:

Early Data Distribution: 1) Utilize the parallel processing power of GPU for data preprocessing tasks. Tasks like normalization, resizing, or augmentation can be performed efficiently on GPU, reducing the time spent on preparing the data for training. 2) Leverage GPU parallelism for batch processing. Processing multiple batches simultaneously on the GPU can enhance data distribution efficiency.

Final Calculation Step: 1) Ensure that the layers of the MLP are parallelized effectively on the GPU. Modern deep learning frameworks automatically distribute computations across GPU cores, but optimizing the model architecture can further enhance parallel processing. 2) Experiment with mixed precision training on GPU. By using lower precision data types, the training process can potentially be accelerated without sacrificing too much model accuracy. 3) Monitor and optimize GPU memory usage. Ensure that the GPU memory is efficiently managed during training to prevent unnecessary overhead.

Additional Considerations: 1) Manage GPU resources effectively, especially when dealing with large datasets. This includes optimizing the data loading pipeline to keep the GPU fed with data. 2) Experiment with batch sizes to find the optimal balance between computation efficiency and memory usage. Larger batch sizes often lead to more efficient GPU utilization. By implementing these GPU acceleration strategies, you can significantly reduce the time spent on the data distribution step and the final calculation step, thereby improving the overall efficiency of training an MLP neural network on the MNIST dataset.

The principle of implementing matrix multiplication on GPUs using additive secret sharing and multiplication triplets primarily relies on decomposing the involved matrices through additive secret sharing. In this process, a matrix X is divided into multiple shares, for instance, $< X >_1$, $< X >_2, \ldots, < X >_n$, such that the sum of these shares equals the original matrix X. This allows for individual processing of each share without revealing information about the original matrix. Additionally, the concept of multiplication triplets plays a crucial role. In the preprocessing stage, a set of random matrix pairs (A, B, C) is generated, where C is the product of A and B. These triplets are used during the multi-party computation to facilitate calculations without directly handling the original data. When the multiplication of two secret matrices X and Y is required, they are first decomposed into shares of additive secret sharing. Subsequently, each party calculates the difference matrices $< e >_i = < X >_i - < A >_i$ and $< f >_i = < Y >_i - < B >_i$ and exchanges these matrices among the participants. Leveraging the high parallel processing capability of GPUs, each participant can quickly compute their respective partial products and sums. Ultimately, by combining these partial products, the product of the secret matrices X and Y can be reconstructed. The advantage of this method lies in its combination of the efficiency of GPUs with the privacy protection characteristics of additive secret sharing and multiplication triplets. Through this approach, complex matrix multiplication operations can be securely and efficiently performed even in distributed and potentially untrusted environments. This is particularly useful in scenarios that require processing large volumes of data with strict privacy and security requirements, such as financial analysis, medical data processing, and secure multi-party computation. Additionally, this method significantly reduces computation time and resource consumption when handling large datasets compared to traditional secure computing methods, thereby enhancing overall efficiency.

Algorithm 1: Offline stage: Generation of multiplicative triples (a, b, c)

Output: $(a, b, c) \rightarrow [< a >_0, < a >_1], [< b >_0, < b >_1], [< c >_0, < c >_1]$;

Step(1):

1: **for** $i = 0$ to 1 **do**

2: $< a >_i = intRand(0, Rand\_MAX)$;
$< b >_i = intRand(0, Rand\_MAX)$;

3: **end for**

Step(2):

4: $a = \sum_{i=0}^{1} < a >_i$;
$b = \sum_{i=0}^{1} < b >_i$;

Step(3):

5: $CopyHtoD(gpu\_a, a, row1 \times col1)$;
$CopyHtoD(gpu\_b, b, row2 \times col2)$;

6: $gpu\_c = GPU\_Mul(a, b)$;

7: $CopyDtoH(c, gpu\_c, row1 \times col2)$;

Step(4):

8: **for** $i = 0$ to $row1 \times col2$ **do**

9: $< c[i] >_1 = intRand(0, Rand\_MAX)$;

10: $< c|i| >_2 = c|i| - < c|i| >_1$;

11: **end for**

Since two-party computing includes offline and online parts, GPU acceleration can also be divided into offline and online parts. For the offline part, the goal is to prepare encrypted data for both servers. As shown in Fig. 3, the user inputs matrix X and matrix Y, and then performs various

processing on these two matrices in the offline part. Matrix X is divided into matrices $< X >_0$ and $< X >_1$ and matrix Y is divided into matrices $< Y >_0$ and $< Y >_1$, where matrix $X >_0$ and $< Y >_0$ is randomly generated and where $<>$ is used to represent the input secret sharing value. This step only takes a short time. a and b in the randomly generated triplet (a, b, c) are divided into $< a >_0$, $< a >_1$ and $< b >_0$, $< b >_1$, respectively, this step takes a very short time, and among them the matrix c is obtained by multiplying them. This step takes a long time, accounting for more than 90% of the total offline time. This step includes dense matrix multiplication calculations that can be accelerated using GPUs. Algorithm 1 shows the entire process of generating multiplication triples and spliting them. After completing this step, the data with index 0 is distributed to server 0, and the data with index 1 is distributed to server 1.

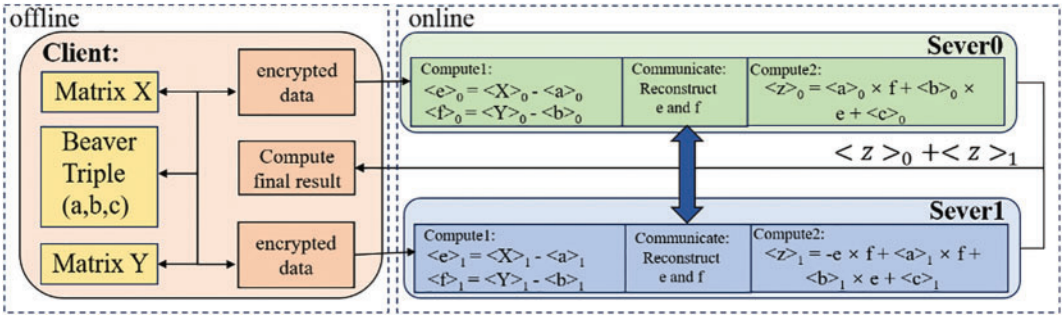


Figure 3: Data preparation stage and part of the data reconstruction process

For the online part of the calculation, the matrix multiplication operation still takes up a large proportion of the time. The main calculations are shown in Algorithm 2. The function *intRand( )* in steps 2 and 9 serves the purpose of generating random numbers of integer type. The functions *CopyHtoD( )* and *CopyDtoH( )* facilitate the transfer of data between the GPU and CPU and the function *GPU_Mul( )* indicates performing multiplication operations on the GPU. The calculations related to the online part are the step (4) and (5) in Algorithm 2. The calculation amount of step (4) is small, and the calculation amount of step (5) is large, so the CPU is used to process step (4) and the GPU is used to process step (5). Because the GPU is used to process step (5), the matrices e, f, $< X >_i$, $< Y >_i$, and $< c >_i$ need to be sent to the GPU. Therefore, we transplant the matrix multiplication operation to the GPU and divide the matrix into n sub-matrices to make full use of the cache and use the parallel performance of the GPU to calculate the results of each matrix block separately, and summarize the results of each part to obtain the final matrix multiplication result as shown in step (6) in Algorithm 2 to improve the performance of matrix calculations. Among them, in order to further improve cache utilization, we performed a matrix transpose on one of the matrices before calculation.

Algorithm 2: Online stage: The main calculation steps of matrix

Input: Server i takes as input $< X >_i$, $< Y >_i$, $< a >_i$, $< b >_i$, $< c >_i$ distributed by the client;
Output: Server i outputs a partial result of the final result of $X \times Y$;
Compute with triples:
1: **for** $i = 0$ to 1 **do**
2: $< e >_i =< X >_i - < a >_i$;
$< f >_i =< Y >_i - < b >_i$;
3: **end for**

(Continued)

Algorithm 2 (continued)

Reconstruct e, f through communication:

4: $e = \sum_{i=0}^{1} < e >_i$;
$f = \sum_{i=0}^{1} < f >_i$;

The two servers get e, f and calculate separately on GPU:

5: $< Z >_i = (-i) \times e \times f + < a >_i \times f + < b >_i \times e + < c >_i$;

The client calculates the final result:

6: $Z = \sum_{i=0}^{1} < Z >_i = X * Y$;

3.1.2 Two-Party Communication Optimization

In order to calculate e and f, servers need to communicate $< e >_i$ and $< f >_i$, and from the step (2) in Algorithm 2 we can know that $< e >_i$ and $< f >_i$ depend on $< X >_i$ and $< Y >_i$ as shown in Fig. 3. In machine learning tasks, the above matrices X and Y can be model parameters, and their iteration matrices are usually sparse. Let $< X >_{i,j}$ and $< Y >_{i,j}$ represent the j-th iteration of X and Y on server i, so $< X >_{i,j+1}$ and $< Y >_{i,j+1}$ can be expressed as Eq. (1), where $\delta_{ij}^{<X/Y>}$ represents the change between two iterations, that is, the gradient. If $\delta_{ij}^{<X>}$ and $\delta_{ij}^{<Y>}$ are sparse (the default setting is that 75% of the elements in the matrix are 0), then use the CSR [31] to store it and then transmit the compressed $\delta_{ij}^{<X>}$ or $\delta_{ij}^{<Y>}$. If not sparse, the original data is transferred. Experiments have shown that the overhead caused by communication can be greatly reduced by first performing a sparsity check similar to the above on the matrix that needs to be transmitted to the GPU before transmitting it, and the checking time is negligible.

$$< X >_{i,j+1} = < X >_{i,j} + \delta_{ij}^{<X>}, < Y >_{i,j+1} = < Y >_{i,j} + \delta_{ij}^{<Y>} \tag{1}$$

After getting the expressions of $< X >_{i,j}$ and $< Y >_{i,j}$, we can get the expressions of $< e >_{i,j}$ and $< f >_{i,j}$ as shown in Eqs. (2) and (3). So, if we can tell ahead of time that $\delta_{ij}^{<X/Y>}$ is sparse, then we only need to transfer changing values of $\delta_{ij}^{<X/Y>}$ between the two servers.

$$< e >_{i,j+1} = < X >_{i,j+1} - < a >_i = < X >_{ij} + \delta_{ij}^{<X>} - < a >_i = < e >_{ij} + \delta_{ij}^{<X>} \tag{2}$$

$$< f >_{i,j+1} = < Y >_{i,j+1} - < b >_i = < Y >_{ij} + \delta_{ij}^{<Y>} - < b >_i = < e >_{ij} + \delta_{ij}^{<Y>} \tag{3}$$

3.1.3 Communication and Computing Parallelism

After conducting an in-depth analysis of various machine learning tasks, particularly those associated with deep learning, two prominent characteristics have been identified. Firstly, these tasks often involve multiple interconnected neural network layers, each contributing to the extraction of complex nonlinear relationships. Secondly, machine learning tasks encompass both forward propagation and back propagation phases. To address the intricacies of these tasks, a parallel computing and communication propagation design [17] has been employed. This design aims to exploit parallelism and overlap specific steps across different layers, thereby optimizing the overall efficiency of the machine learning process.

In the context of deep learning, where the number of neural network layers is typically determined empirically and exceeds two layers to capture intricate nonlinear patterns, it becomes crucial to streamline the forward and backward propagation phases. Notably, the forward propagation of a given layer serves as a prerequisite for the subsequent layers, creating a sequential dependency. However, this limitation is mitigated during the back propagation phase, where certain steps, such

as the reconstruct step, do not necessarily need to wait for the completion of subsequent layers. During the back propagation reconstruct step, a portion of the calculations relies on the forward propagation GPU operation step of the same layer. These calculations are seamlessly integrated into the backward propagation reconstruct step. Meanwhile, another portion of the calculation is contingent upon the next layer. Consequently, the allocation of computational tasks is strategically managed, with some computations assigned to the backward pass GPU operation step. This strategic distribution of computational tasks enables the concurrent execution of the forward propagation reconstruct step of the current layer with the backward propagation reconstruct step of the previous layer. This intricate overlap, as illustrated in Fig. 4, optimizes the computational workflow, leveraging parallelism and communication propagation to enhance the overall efficiency of the machine learning process. This design ensures that certain steps are synchronized and executed concurrently, maximizing the utilization of computational resources and minimizing idle time during the training of deep neural networks.

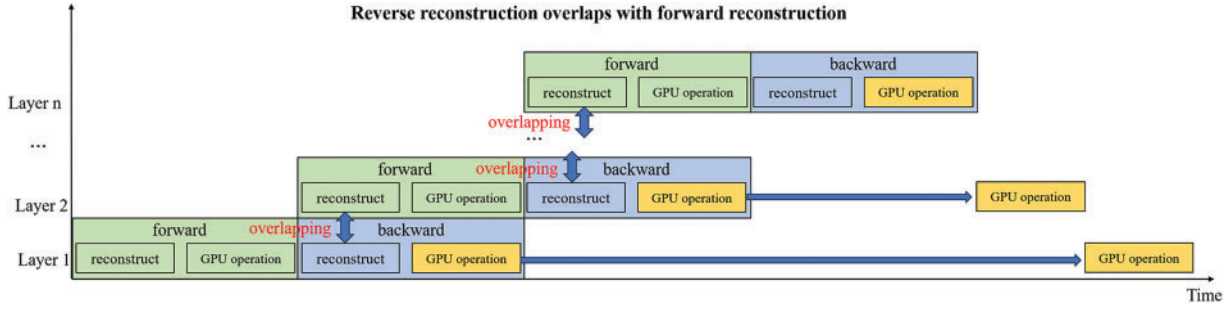


Figure 4: Overlapping reconstruction steps of different layers

3.2 Hardware Optimization

3.2.1 Optimization of Related Calculations on CPU

As discussed in Section 3.1.1, we ported the compute-intensive parts to GPUs and the rest to CPUs. The main computational tasks on the CPU include: 1) generating the random matrices X_0, Y_0, a_0, a_1, b_0 and b_1, and 2) calculating the matrices X_1, Y_1, a and b by addition and subtraction. We optimize related calculations performed on the CPU, including optimization of random number generation and optimization of matrix addition and subtraction to reduce the cost of computing time using the CPU and we optimize these operations in parallel to fully utilize the CPU computing resources.

Random Number Generation Optimization. EG-STC involves random number generations. To generate random numbers concurrently in multiple threads, we applied a thread-safe random number generator, the Melson Twisted 19937 Generator (MT19937) [32], from the C++11 random library. Melson Twister is a widely used pseudorandom number generator (PRNG) known for its long period and good statistical properties. The 19937 in its name refers to its 32-bit word size and a state space of $2^{19937} - 1$.

Matrix Addition and Subtraction Optimization. The addition and subtraction operations of matrices in EG-STC mainly involve steps (2) and (4) in Algorithm 1 and step (2) in Algorithm 2. These operations are completed by using a for loop to traverse two matrices, and can be directly optimized into multi-threaded for loop parallel operations. When ParSecureML [17] writes the calculation results into the sum-difference matrix, it needs to avoid the overhead caused by multiple threads writing to the same cache line. Each cache line can store 16 FP32 values, and cache line write contention can be

avoided by scheduling at least 16 cyclic tasks per thread. In addition, to reduce the overhead caused by opening multiple parallel regions, multiple parallel regions should be merged.

3.2.2 GPU Optimization

In order to further improve the performance of GPU, we first analyze the GPU utilization process. We use the Jtop performance analysis tool provided by Jetson Systems to collect and view GPU activities, including kernel execution, data transfer, and CUDA API calls. We collected GPU performance data for EG-STC from both client and server and observed three main activities: CUDA memory copy from host to device, general matrix multiplication (GEMM) operations, and CUDA memory copy from device to host. Since the CUDA memory copy overhead between host and device depends on the memory transfer channel, we mainly performed GEMM optimization operations.

To optimize the GEMM operations in EG-STC on the NVIDIA Nano native GPU without utilizing Tensor Cores, we adopt a different approach. Tensor Cores are programmable matrix-multiply-and-accumulate units designed for enhanced performance, but since the NVIDIA Nano GPU may not have dedicated Tensor Cores, we focus on conventional techniques. In our implementation, we employ standard GPU operations without invoking Tensor Cores. Specifically, we use the available CUDA libraries and functions suitable for the GPU architecture of NVIDIA Nano, ensuring compatibility and efficiency in matrix multiplication tasks. This alternative approach optimizes GEMM operations on the NVIDIA Nano without relying on Tensor Cores, making the implementation tailored to the capabilities of the GPU within the NVIDIA Nano platform.

Memory Layout Optimization. Memory layout optimization is a critical aspect in enhancing the performance of General Matrix Multiplication (GEMM) operations on the NVIDIA Nano platform. The primary goal is to improve access efficiency and minimize memory transfer overhead by strategically arranging data in memory. Given the limited memory bandwidth of the NVIDIA Nano, effective utilization of shared memory becomes pivotal in achieving optimal performance. One key strategy is to carefully choose the size of shared memory, ensuring it aligns with the specific requirements of the GEMM operations. Additionally, designing a data layout that takes advantage of shared memory locality is essential. This involves organizing data in a way that maximizes the reuse of shared memory, thereby reducing the reliance on slower global memory access. To further enhance performance, it is crucial to avoid write conflicts in shared memory. Conflicts can occur when multiple threads attempt to write to the same location simultaneously, leading to inefficiencies. By implementing synchronization mechanisms or optimizing the algorithm to minimize such conflicts, the overall performance of GEMM operations can be significantly improved. Another important consideration is the judicious utilization of shared memory for frequently accessed data. By identifying and prioritizing the most critical data for storage in shared memory, the algorithm can exploit the faster access times offered by shared memory, thus reducing latency and improving overall efficiency. Furthermore, constant memory can be leveraged for immutable data shared across the entire thread block. This allows for efficient sharing of read-only data among threads without incurring the overhead associated with redundant storage in shared memory.

Batch Processing. Batch processing necessitates a judicious decision-making process regarding the selection of an appropriate block size, a crucial determinant in achieving optimal performance through effective utilization of GPU parallelism. The configuration of block size and grid size can be subjected to experimentation, tailored to the unique characteristics of the problem at hand and the underlying GPU architecture. On the NVIDIA Nano platform, the careful selection of a block size for computations involves a nuanced understanding of GPU architecture, considerations for

thread-level parallelism, shared memory utilization, and optimal register usage. In the pursuit of optimizing collaborative thread execution, it is imperative to comprehend the specific nuances of the GPU architecture, given the sensitivity of the NVIDIA Nano GPU to thread-level parallelism. The block size plays a pivotal role in orchestrating parallel threads, and a well-chosen size can significantly impact overall performance. Striking the right balance requires a keen awareness of how the GPU architecture handles parallel execution, ensuring that the selected block size aligns seamlessly with the inherent characteristics of the computational workload. Moreover, the thoughtful consideration of shared memory and register usage is paramount to prevent exceeding hardware resource limits. Efficient utilization of shared memory is crucial for facilitating communication and data exchange among threads, while judicious register allocation ensures optimal usage of on-chip resources. This delicate balancing act is essential for preventing bottlenecks and resource contention that could hamper the parallel processing capabilities of the GPU. The nature of the computational task and its associated memory access patterns also introduces considerations into the determination of the ideal block size. Adapting the block size to the specific memory requirements and access patterns of the application ensures that the GPU can seamlessly access and process data, minimizing latency and maximizing throughput.

Thread and Block Configuration. In the context of EG-STC utilizing the CUDA dim3 structure on the NVIDIA Nano platform for configuring thread blocks and grids efficiently, the optimization process involves a systematic exploration of diverse thread block sizes through experimentation and iteration. The objective is to observe and quantify the impact of various thread block configurations on overall performance. To facilitate this exploration, the Jetson Stats performance analysis tool is leveraged, enabling real-time data collection on different configurations and contributing to the identification of optimal setups. The iterative optimization process adopted by EG-STC encompasses a range of considerations. First and foremost, task parallelism is a key factor, ensuring that the chosen configurations align with the nature of the computational tasks at hand. This involves assessing how tasks can be effectively parallelized across threads within a block and across different blocks in the grid. Avoidance of thread conflicts is another crucial aspect of the optimization process. Conflicts can arise when multiple threads attempt to access or modify the same data simultaneously, leading to inefficiencies and performance bottlenecks. The iterative testing helps identify configurations that mitigate or eliminate these conflicts, enhancing overall execution efficiency. Memory access patterns are also evaluated during the optimization process. Understanding how data is accessed and shared among threads within a block and across blocks is essential for maximizing throughput. The chosen configurations aim to align with these access patterns, minimizing latency and optimizing data transfer within the GPU's memory hierarchy. Importantly, the optimization process is tailored to the architecture of the NVIDIA Nano GPU. This involves considering the specific features, limitations, and strengths of the GPU architecture to ensure that the selected thread block and grid configurations make efficient use of the available resources. By adopting this comprehensive approach, EG-STC can dynamically adjust thread block and grid configurations based on the specific requirements of different computational tasks. This adaptability maximizes GPU utilization and, in turn, achieves optimal performance for a diverse range of scenarios. The synergy of task parallelism, conflict avoidance, consideration of memory access patterns, and alignment with GPU architecture collectively contributes to the effectiveness of EG-STC's dynamic thread and block configuration strategy.

Utilizing the features of the Jetson Nano, which boasts 128 CUDA cores suitable for handling extensive computational requests, we conducted experiments involving matrix multiplication tasks sized at 1000×1000. This particular size choice aligns with the parallel processing capabilities of the GPU, maximizing its computational efficiency. Through rigorous experimentation and analysis,

we identified the optimal thread block configuration to be block(32,32), ensuring a balance between parallel processing capability and memory utilization. Thread block size plays a pivotal role in determining the efficiency of parallel computation on the GPU. Larger thread blocks can exploit data locality, reducing the need for accessing global memory frequently. However, they may also lead to excessive usage of shared memory, potentially hindering performance. To address this, we carefully adjusted the thread block size to strike a balance between memory access patterns and computational efficiency. In determining the dimensions of the computation grid, we employed a fixed value derived from the thread block size and the dimensions of the matrices. This ensured that each output element received adequate computational resources, even if the matrix dimensions did not perfectly align with the thread block size. This approach optimized resource allocation and minimized wastage, contributing to overall performance enhancement.

In summary, selecting the correct thread block configuration is crucial for efficient resource utilization and performance optimization on the GPU. By meticulously adjusting thread block sizes and computation grid dimensions, we were able to harness the full potential of the Jetson Nano computational prowess for our matrix multiplication tasks.

4 Performance Evaluation

In this section, we primarily evaluate the implementation performance of EG-STC, including the performance of executing various secure machine learning tasks on the GPU, and compare it with related works.

4.1 Experiment Setup and Evaluation Methods

Our experimental platform configuration is shown in Table 1. On the embedded NVIDIA Nano GPU, the CPU and GPU share 4GB of memory. This setup provides ample computing resources and reasonable memory sharing in a limited environment. In the field of cryptography, we typically focus on the throughput and computation latency of cryptographic calculation requests, which are commonly used performance metrics. Additionally, in edge computing environments composed of resource-constrained embedded devices, power consumption becomes a significant performance metric closely related to production costs and standby times. Therefore, to comprehensively assess the experimental results, we introduce a additional performance metric: Energy efficiency ratio. The performance metrics used in this experiment are evaluated as follows:

- Throughput: The amount of data the system can process in a unit of time, commonly used to measure system workload capacity and performance level.
- Latency: The time required to complete a specific computational task, reflecting the total duration from the start to the completion of the task, used to measure system response speed and real-time performance.
- Energy Efficiency Ratio: The efficiency of the system in completing work per unit of energy consumed, commonly used to measure the relationship between energy consumption and performance, and is an important indicator for evaluating system environmental friendliness and cost-effectiveness.

Table 1: The platform configuration of EG-STC

Type	Model
OS	Linux Ubuntu 18.04
Tool chain	CUDA 10.0
CPU	4-core ARM Cortex-A57 64-bit CPU
GPU	NVIDIA Tegra X1 with SM×1 (128 CUDA Cores/MP)
Memory	4GB 64-bit
Power	About 5 W

Experimental Benchmarks. We have chosen 5 typical AI components or tasks as benchmarks for the experiment to comprehensively evaluate our system performance. The following are brief descriptions of these benchmark tasks:

- **Matrix multiplication** is a classic linear algebra operation. In neural networks, matrix multiplication is used for multiplying weights with input data. Suppose we have an input vector x and a weight matrix W; the output y obtained through matrix multiplication is the corresponding weighted sum. Matrix multiplication provides a foundation for implementing complex models and algorithms. Our work, based on secret sharing and secure triplet generation, has implemented secure matrix multiplication and further enhanced performance by utilizing the parallel computing capabilities of GPUs. We use this benchmark to evaluate system performance in handling secure matrix operations.
- **CNNs** [33] are particularly suited for tasks such as image classification, object detection, and face recognition. The core function of CNNs is to efficiently abstract and represent images by learning local features and structural information within the images. Compared to traditional neural networks, CNNs introduce key components like convolutional layers. Convolutional layers capture local features in images through the sliding operation of kernels. The convolution operation in CNNs can be implemented through matrix multiplication. We choose CNNs as a benchmark to examine system learning and inference capabilities on complex visual data.
- **MLP** [34] is a classic type of feedforward neural network structure, widely used in tasks such as classification, regression, and pattern recognition. MLP learns by training a model to map relationships from inputs to outputs. During the training process, MLP uses the backpropagation algorithm for optimization, adjusting the connection weights to minimize model error. We use MLP as a benchmark task to evaluate the system adaptability to complex nonlinear data.
- **Linear regression** [35] is a basic statistical method used to model linear relationships, widely applied in regression analysis. It is a common predictive model in the field of artificial intelligence. By establishing a linear relationship, it models the association between input features and output targets. We use linear regression as a benchmark to understand system performance level in handling linear relationship modeling tasks.
- **Logistic regression** [36] is a classic method used for modeling binary classification problems. It establishes a linear decision boundary and applies a logistic function to map linear outputs to probability values between 0 and 1. During training, the model adjusts parameters by maximizing the likelihood function or minimizing the cross-entropy loss. We choose logistic

regression as a benchmark task to evaluate system effectiveness in handling binary classification problems.

Dataset. In the experiments of this paper, considering that the NVIDIA Nano has only 4 GB of memory, we chose to use the MNIST dataset for secure machine learning tasks. The MNIST dataset contains 60,000 training images and 10,000 testing images, each a grayscale image with a size of 28×28 pixels. The dataset is commonly used in tasks involving image recognition and classification. In edge computing scenarios, using a relatively small and lightweight dataset helps to conduct efficient model training and inference under limited resources. As a classic handwritten digit recognition dataset, MNIST has a moderate size and sufficient representativeness, effectively validating secure machine learning tasks on edge devices.

4.2 Experiment Results

In a multi-party edge computing environment where resources are limited and data privacy is strictly required, we have successfully implemented secure matrix multiplication computation and its associated secure AI tasks on an embedded GPU platform. While ensuring the security and practicality of the algorithm, we have thoroughly evaluated the performance of the system, particularly in terms of peak throughput, based on our designed GPU core optimization scheme. As shown in Fig. 5a, we assessed the performance of secure matrix multiplication involving two parties by executing the multiplication of two $N \times N$ matrices. For smaller matrices with a dimension of $N = 10$, the impact of data transfer overhead (that is, copying data from the host to the device and back to the host) on overall performance is particularly significant, indicating that the performance gains from GPU acceleration are relatively limited for small-sized matrices. Conversely, as the dimension of the matrix increases, which means the problem size is getting larger, the impact of GPU acceleration on performance enhancement becomes more pronounced. Specifically, for matrix multiplication with a dimension of $N = 1000$, the latency is reduced compared to the case of $N = 500$.

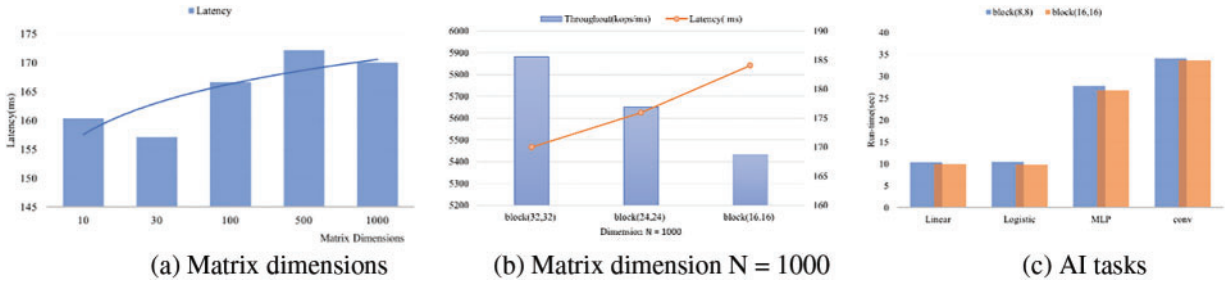


Figure 5: Performance of secure matrix multiplication and machine learning tasks

As indicated in Table 1, the NVIDIA Nano is equipped with 128 CUDA cores, making it particularly suitable for large-scale computational requests. The overall performance gain from GPU acceleration is influenced by the configuration of parallel execution, which includes the sizes of thread blocks and grid dimensions. Specifically, the product of the grid and thread block sizes determines the size of the batch being processed. The size of a thread block refers to the number of threads it contains; for example, block(16,16) indicates a thread count of $16 \times 16 = 256$. Thread blocks are typically set to 16×16, 32×32, or other values divisible by the thread warp size. In most NVIDIA GPUs, a warp consists of 32 threads. The size of the thread block influences parallel computing capability on the GPU. Larger thread blocks can reduce the number of global memory accesses but may also lead to

excessive use of shared memory, thus affecting performance. When computing grid dimensions, fixed values determined by the sizes of the thread blocks and the matrix row and column dimensions ensure that each output element is processed by at least one thread, even when the matrix dimensions are not multiples of the thread block size.

As shown in Fig. 5b, we conducted a comprehensive experimental evaluation of secure matrix multiplication for $N = 1000$ using different thread block configurations: Block(32,32), block(24,24), and block(16,16). The results indicated an increasing trend in throughput as the sizes of the thread blocks increased. Specifically, when the number of thread blocks was set to 1024, we etobserved a throughput of 5881.5 kops/ms, a latency of 170.02 ms, and an energy efficiency ratio of 1176.3 kops/ms/W. As demonstrated in Fig. 5c, we extended our experimental scope to evaluate the performance of secure AI tasks with thread block configurations of block(8,8) and block(16,16).The results revealed that, generally, as the number of thread blocks increased, the execution time of tasks tended to decrease. Choosing the right thread block configuration when optimizing computing resource allocation is crucial, and generally, larger thread block configurations can more efficiently process data and reduce latency.

4.3 Performance Comparison

Table 2 summarizes our implementation and compares it with other works.

Table 2: Secure AI tasks performance comparison

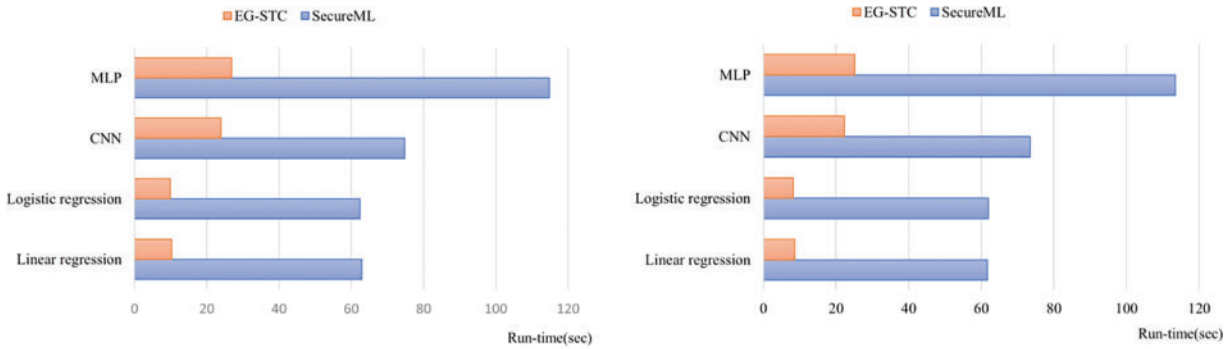
	Platform	AI tasks	Run-time (sec)
Mohassel et al. [37]	2 Intel(R) Xeon(R) CPU E5-2670 v3 TDP 120 W	Linear regression	62.94
		Logistic regression	63.29
		CNN	74.77
		MLP	114.85
Chen et al. [17]	NVIDIA Tesla V100 GPU with 80 SMs 64 CUDA Cores/MP, 250 W	Linear regression	6.62
		Logistic regression	6.77
		CNN	16.88
		MLP	8.48
Chen et al. [17]	NVIDIA Tegra X1 with 1 SM 128 CUDA Cores/MP, 5 W	Linear regression	15.52
		Logistic regression	14.85
		CNN	36.3
		MLP	29.11
Ours	NVIDIA Tegra X1 with 1 SM 128 CUDA Cores/MP, 5 W	Linear regression	8.66
		Logistic regression	8.2
		CNN	22.31
		MLP	25.16

4.3.1 Comparison vs. CPU Implementations

SecureML [37] is the first secure two-party privacy-preserving machine learning system based on MPC. It adopts a semi-honest security model and supports the secure training and prediction of

machine learning models. However, SecureML is about twice as slow as traditional plaintext machine learning, which somewhat limits its widespread application in practice. Additionally, SecureML has not released its source code implementation. Nevertheless, recent work Chen et al. [17] has built upon SecureML to implement related machine learning tasks and conducted experiments on high-performance server-grade processor nodes equipped with 2 Intel(R) Xeon(R) CPU E5-2670 v3 and 128 GB of memory. Compared to SecureML running only on CPUs, our secure two-party machine learning scheme based on embedded GPU acceleration opens up new possibilities for secure machine learning at edge nodes, greatly enhancing its feasibility for practical applications. Our method not only enhances computational power but also speeds up processing, making it possible to carry out complex machine learning tasks in edge computing environments.

In terms of performance, our method has shown great advantages. We deeply tested it on four different secure machine learning tasks. As shown in Fig. 6, Fig. 6a compares the overall time consumption between SecureML and our EG-STC. Fig. 6b focuses on their time consumption in the online phase. Compared to SecureML running on high-performance server-grade CPUs, our EG-STC system, based on embedded GPUs, achieved a substantial improvement in performance, with the overall time acceleration ratio reaching $5-6\times$. Particularly in the online phase, involving computation-intensive secure matrix operations, the performance was notably enhanced by the parallel acceleration provided by GPUs. However, in the offline phase, as it accounts for only a small portion of the overall time and involves lower computational load, it was not suitable for GPU acceleration.



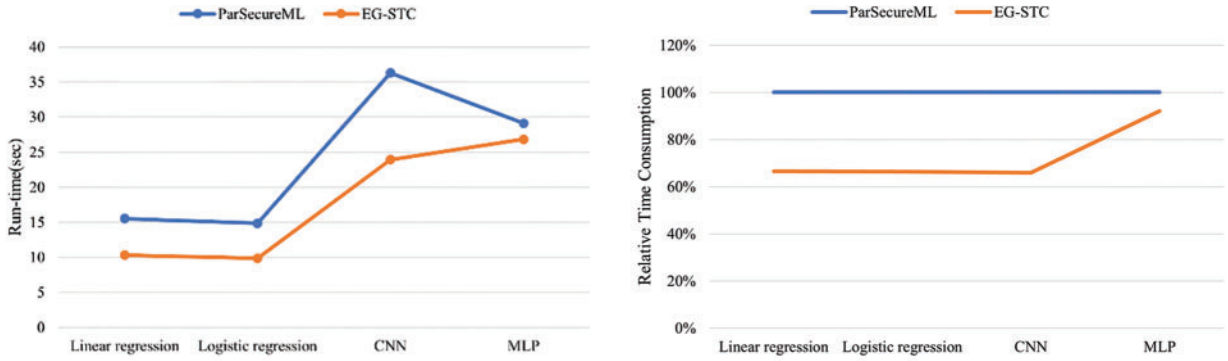
(a) Overall Time Comparison (b) Online Time Comparison

Figure 6: Comparison of time costs between EG-STC and SecureML

From our observations and analysis, we have reached the following conclusions: 1) In common secure machine learning tasks, EG-STC achieved a performance improvement of about 5–6×. This shows that using embedded GPUs to accelerate MPC-based privacy-preserving machine learning tasks in resource-limited edge computing environments is very effective; 2) Despite the MNIST dataset is relatively small and may not fully utilize the parallel computing power of GPUs, the experimental results show that using GPUs still significantly speeds up performance. The size of the MNIST dataset is also suitable for edge computing scenarios; 3) The overall improvement in time performance mainly comes from significant enhancements in the online phase. This substantial improvement not only proves the efficiency of GPU acceleration in ensuring the security of two-party computations but also opens a new path for enhancing the processing efficiency of machine learning tasks in edge computing environments that protecting data privacy.

4.3.2 Comparison vs. GPU Implementations

Recent studies [17, 19, 20] in privacy-preserving machine learning based on MPC have also begun to utilize GPU acceleration. However, in the field of edge computing, our work is the first to explore the potential of using embedded GPUs to accelerate computation-intensive modules in secure computing. Specifically, ParSecureML [17], a parallel secure machine learning architecture running on GPUs focusing on two-party computations, is closely related to our research. Since Chen et al. [17] used the high-end NVIDIA Tesla V100 GPU, designed for scientific computing, directly comparing it with our embedded GPU might not be entirely fair. Therefore, when comparing with advanced GPUs, we consider not only performance but also power consumption. To comprehensively compare the performance of the two systems, we ran the ParSecureML system from Chen et al. [17] on our embedded GPU platform and used the same dataset to compare four different secure machine learning tasks like CNNs in the same experimental environment. As shown in Fig. 7, Fig. 7a shows the difference in overall time consumption between ParSecureML and our EG-STC, while Fig. 7b introduces Relative Time Consumption as an evaluation metric, indicating the time our system takes to complete tasks compared to the system from Chen et al. [17]. The experimental results show that on the same platform, for machine learning tasks requiring extensive secure matrix operations, our system relative time consumption is about 60% to 70%. This means our system can complete tasks in 60% to 70% of the time required by the system of Chen et al. [17], demonstrating higher efficiency. This not only proves the effectiveness of our proposed GPU optimization method but also indicates that our approach is more advanced than the approach of Chen et al. [17]. To enhance the adaptability and universality of our system, we have employed the PTX ISA instruction set and meticulously adjusted the parallel computing parameters. This strategy enables our system to not only operate efficiently on specific embedded GPUs but also to be compatible and adaptable to various other embedded GPU platforms, such as NVIDIA TX2 or AGX GPUs series. By doing so, we ensure that our system performs well in different hardware environments and can be seamlessly integrated into various application scenarios. Moreover, the flexibility offered by the PTX ISA instruction set allows us to optimize and adjust according to the distinct characteristics of different hardware, further enhancing the efficiency and stability of our system when handling complex tasks.



(a) Overall Time Comparison (b) Relative Time Consumption

Figure 7: Comparison of time costs between EG-STC and ParSecureML

In real-world scenarios, the EG-STC framework finds application across various domains within edge computing. For instance, in intelligent surveillance systems, where real-time video analysis is crucial for anomaly detection or object recognition, EG-STC ensures secure computation of video

data between edge devices while preserving user privacy. Similarly, in healthcare, where edge devices gather and process sensitive medical data like heart rate or blood pressure monitoring, EG-STC guarantees secure computation of this information, facilitating tasks such as patient diagnosis or disease prediction while maintaining patient confidentiality. Moreover, in smart traffic systems, EG-STC enables secure computation of traffic data among edge devices, facilitating tasks such as traffic flow prediction or vehicle identification while safeguarding user privacy. Lastly, in industrial automation, where edge devices monitor and control various sensors and actuators, EG-STC ensures secure computation of industrial data, allowing tasks like quality control or fault diagnosis to be performed securely while protecting enterprise trade secrets and intellectual property. These cases highlight the flexibility and efficiency of the EG-STC framework in tackling security and privacy issues across various edge computing applications.

4.4 Discussion

Comparison. Through our extensive experimental evaluations, compared to the first server-grade CPU-based MPC privacy-preserving machine learning solution, SecureML [37], its processing speed is slower due to additional computational and communication costs, limiting its practicality. ParSecureML [17] improved upon SecureML, reducing the performance gap to 5.8 times, but it still relies on server-grade GPU resources. While this improvement is a step forward for environments that can access high-performance computing resources, it is not suitable for resource-constrained edge computing environments. In contrast to these existing solutions, our EG-ETC scheme is specifically designed for embedded GPU platforms, targeting the needs of privacy-preserving machine learning in edge computing environments. On the same embedded GPU platform, The operational time of EG-ETC is only 60% to 70% of what ParSecureML requires, significantly improving processing speed while maintaining privacy protection features.

Scalability. The EG-etc scheme leverages GPU-accelerated secure matrix multiplication to balance security and efficiency in machine learning tasks. This approach is not only applicable to the machine learning tasks tested in this paper but also extends to other programs that involve similar matrix operations. Thus, our scheme can ensure both security and efficiency for a broader range of applications.

Limitation. Our solution is tailored for resource-constrained edge computing environments, particularly for embedded systems, which necessitates a focus on smaller-scale datasets. This focus inherently limits the applicability of our scheme to handling large-scale datasets typically processed in server-grade GPU clusters. However, for the context of edge computing environments, smaller datasets are often sufficient to meet the operational needs.

5 Conclusion

In this paper, we present a solution for secure two-party computation on embedded GPUs, named EG-STC, designed for executing secure AI tasks at edge computing nodes. To the best of our knowledge, this is the first effort to use embedded GPUs for efficient and secure two-party computation. EG-STC offers a more efficient and practical approach to secure matrix multiplication and secure AI tasks compared to other works in the field. On an embedded GPU with a power draw of 5 W, our implementation achieved a secure two-party matrix multiplication throughput of 5881.5 kops/ms and an energy efficiency ratio of 1176.3 kops/ms/W. Leveraging our EG-STC framework, we have attained an overall time acceleration ratio of $5 - 6\times$ compared to solutions running

on server-grade CPUs; on the same platform, our solution required only 60% to 70% of the runtime of the previously best-known methods. In edge computing environments, due to the limited resources such as memory, we generally perform small-scale machine learning tasks, such as image recognition and sensor data processing in smart manufacturing. Therefore, our scheme for secure AI tasks based on embedded GPUs is effective. The practical implications of EG-STC extend beyond the technical achievements presented in this paper. This innovative framework holds significant potential for various industries and scenarios, offering enhanced security and efficiency in AI tasks at the edge. The adaptability and performance gains showcased by EG-STC make it a promising solution for real-world applications, particularly in contexts where edge computing plays a crucial role.

Acknowledgement: The authors wish to express their appreciation to the reviewers for their helpful suggestions which greatly improved the presentation of this paper.

Funding Statement: This work was supported in part by Major Science and Technology Demonstration Project of Jiangsu Provincial Key R & D Program under Grant No. BE2023025, in part by the National Natural Science Foundation of China under Grant No. 62302238, in part by the Natural Science Foundation of Jiangsu Province under Grant No. BK20220388, in part by the Natural Science Research Project of Colleges and Universities in Jiangsu Province under Grant No. 22KJB520004, in part by the China Postdoctoral Science Foundation under Grant No. 2022M711689.

Author Contributions: The authors confirm contribution to the paper as follows: Study conception and design: Z. Dong, J. Dong; data collection: X. Ge; analysis and interpretation of results: X. Ge, H. Yue, J. Xu; draft manuscript preparation: Z. Dong, J. Xu. All authors reviewed the results and approved the final version of the manuscript.

Availability of Data and Materials: The data that support the findings of this study are available from the corresponding author, Jiang Xu, upon reasonable request.

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] P. Habibi, M. Farhoudi, S. Kazemian, S. Khorsandi, and A. Leon-Garcia, "Fog computing: A comprehensive architectural survey," *IEEE Access*, vol. 8, pp. 69105–69133, 2020. doi: 10.1109/ACCESS.2020.2983253.

[2] R. Ande, B. Adebisi, M. Hammoudeh, and J. Saleem, "Internet of things: Evolution and technologies from a security perspective," *Sustain. Cities Soc.*, vol. 54, pp. 101728, 2020. doi: 10.1016/j.scs.2019.101728.

[3] P. Regulation, "Regulation (EU) 2016/679 of the European parliament and of the council," *Regulation (EU)*, vol. 679, pp. 2016, 2016.

[4] M. M. Yang, T. L. Guo, T. Q. Zhu, I. Tjuawinata, J. Zhao and K. Y. Lam, "Local differential privacy and its applications: A comprehensive survey," *Comput. Stand. Inter.*, vol. 89, pp. 103827, 2023. doi: 10.1016/j.csi.2023.103827.

[5] C. Gentry, "A fully homomorphic encryption scheme," Ph.D. thesis, Stanford Univ., USA, 2009.

[6] A. C. Yao, "Protocols for secure computations," in *23rd Annu. Symp. Found. Comput. Sci. (SFCS 1982)*, Chicago, IL, USA, 1982, pp. 160–164.

[7] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl and P. Scholl, "Efficient pseudorandom correlation generators: Silent OT extension and more," in *Adv. Cryptol.–CRYPTO 2019: 39th Annu. Int. Cryptol. Conf.*, Santa Barbara, CA, USA, Aug. 18–22, 2019, vol. 39, pp. 489–518.

[8] M. Keller, "MP-SPDZ: A versatile framework for multi-party computation," in *Proc. 2020 ACM SIGSAC Conf. Comp. Commun. Secur.*, 2020, pp. 1575–1590.
[9] P. Rindal, "libOTe: An efficient, portable, and easy to use oblivious transfer library," 2018. Accessed: Oct. 15, 2023. [Online]. Available: https://github.com/osu-crypto/libOTe
[10] S. Wagh, D. Gupta, and N. Chandran, "SecureNN: 3-party secure computation for neural network training," in *Proc. Priv. Enhanc. Technol.*, vol. 2019, no. 3, pp. 26–49, 2019. doi: 10.2478/popets-2019-0035.
[11] J. Liu, M. Juuti, Y. Lu, and N. Asokan, "Oblivious neural network predictions via MiniONN transformations," in *Proc. 2017 ACM SIGSAC Conf. Comput. Commun. Secur.*, Dallas, TX, USA, 2017, pp. 619–631.
[12] B. Knott, S. Venkataraman, A. Hannun, S. Sengupta, M. Ibrahim and L. van der Maaten , "CrypTen: Secure multi-party computation meets machine learning," *Adv. Neural Inf. Process Syst.*, vol. 34, pp. 4961–4973, 2021.
[13] N. Kumar, M. Rathee, N. Chandran, D. Gupta, A. Rastogi and R. Sharma, "CryptFlow: Secure TensorFlow inference," in *2020 IEEE Symp. Secur. Priv. (SP)*, San Francisco, CA, USA, IEEE, 2020, pp. 336–353.
[14] D. Rathee *et al.*, "CryptFlow2: Practical 2-party secure inference," in *Proc. 2020 ACM SIGSAC Conf. Comput. Commun. Secur.*, 2020, pp. 325–342.
[15] Y. Li and W. Xu, "PrivPy: General and scalable privacy-preserving data mining," in *Proc. 25th ACM SIGKDD Int. Conf. Knowl. Discov. Data Min.*, Anchorage, Alaska, USA, 2019, pp. 1299–1307.
[16] T. K. Frederiksen, T. P. Jakobsen, and J. B. Nielsen, "Faster maliciously secure two-party computation using the GPU," in *Secur. Cryptogr. Netw.: 9th Int. Conf.*, Amalfi, Italy, 2014, pp. 358–379.
[17] Z. Chen, F. Zhang, A. C. Zhou, J. Zhai, C. Zhang and X. Du, "ParSecureML: An efficient parallel secure machine learning framework on GPUs," in *Proc. 49th Int. Conf. Parallel Process.*, Edmonton, Alberta, Canada, 2020, pp. 1–11.
[18] W. Z. Srinivasan, P. Akshayaram, and P. R. Ada, "DELPHI: A cryptographic inference service for neural networks," in *Proc. 29th USENIX Secur. Symp.*, Santa Clara, CA, USA, 2019, pp. 2505–2522.
[19] S. Tan, B. Knott, Y. Tian, and D. J. Wu, "CryptGPU: Fast privacy-preserving machine learning on the GPU," in *2021 IEEE Symp. Secur. Priv. (SP)*, San Francisco, CA, USA, IEEE, 2021, pp. 1021–1038.
[20] J. L. Watson, S. Wagh, and R. A. Popa, "Piranha: A GPU platform for secure computation," in *31st USENIX Secur. Symp. (USENIX Security 22)*, Boston, MA, USA, 2022, pp. 827–844.
[21] Y. Meng, S. Wang, and A. Schedel, "Multi-GPU for piranha, a multiparty computation framework," 2022. Accessed: Nov. 25, 2023. [Online]. Available: https://people.eecs.berkeley.edu/~kubitron/courses/cs262a-F22/projects/reports/project8_report_ver3.pdf
[22] J. Dong, G. Fan, F. Zheng, T. Mao, F. Xiao and J. Lin, "TEGRAS: An efficient tegra embedded GPU-based RSA acceleration server," *IEEE Internet Things J.*, vol. 9, no. 18, pp. 16850–16861, 2022. doi: 10.1109/JIOT.2022.3152203.
[23] J. Dong, P. Zhang, K. Sun, F. Xiao, F. Zheng and J. Lin, "EG-FourQ: An embedded GPU based efficient ECC cryptography accelerator for edge computing," *IEEE Trans. on Industrial Informatics*, vol. 19, no. 6, pp. 7291–7300, 2023. doi: 10.1109/TII.2022.3205355.
[24] J. Dong, S. Lu, P. Zhang, F. Zheng, and F. Xiao, "G-SM3: High-performance implementation of GPU-based SM3 hash function," in *2022 IEEE 28th Int. Conf. Parallel Distrib. Sys. (ICPADS)*, Xi'an, China, IEEE, 2023, pp. 201–208.
[25] A. Shamir, "How to share a secret," *Commun. ACM*, vol. 22, no. 11, pp. 612–613, 1979. doi: 10.1145/359168.359176.
[26] G. R. Blakley, "Safeguarding cryptographic keys," in *Manag. Require. Knowl., Int. Workshop on*, Orlando, FL, USA, IEEE Computer Society, 1979, pp. 313–313.
[27] D. Beaver, "Efficient multiparty protocols using circuit randomization," in *Adv. Cryptol. CRYPTO 91: Proc. 11*, Santa Barbara, CA, USA, Springer, 1992, pp. 420–432.
[28] "NVIDIA," 2024. Accessed: Dec. 31, 2023. [Online]. Available: https://www.nvidia.com/en-sg/
[29] "NVIDIA CUDA," 2017. Accessed: Dec. 31, 2023. [Online]. Available: https://docs.nvidia.com/cuda/doc/index.html.

[30] "NVIDIA Jetson Nano," 2024. Accessed: Dec. 31, 2023. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-nano/product-development/.
[31] S. AlAhmadi, T. Muhammed, R. Mehmood, and A. Albeshri, "Performance characteristics for sparse matrix-vector multiplication on GPUs," in *Smart Infrastructure and Applications: Foundations for Smarter Cities and Societies*, 2020, pp. 409–426. doi: 10.1007/978-3-030-13705-2_17.
[32] J. di Mauro, E. Salazar, and H. D. Scolnik, "Design and implementation of a novel cryptographically secure pseudorandom number generator," *J. Cryptogr. Eng.*, vol. 12, no. 3, pp. 255–265, 2022. doi: 10.1007/s13389-022-00297-8.
[33] S. Lawrence, C. L. Giles, A. C. Tsoi, and A. D. Back, "Face recognition: A convolutional neural-network approach," *IEEE Trans. Neural Netw.*, vol. 8, no. 1, pp. 98–113, 1997. doi: 10.1109/72.554195.
[34] S. K. Pal and S. Mitra, "Multilayer perceptron, fuzzy sets, classifiaction," 1992. Accessed: Dec. 31, 2023. [Online]. Available: http://library.isical.ac.in:8080/jspui/bitstream/10263/4569/1/308.pdf
[35] D. C. Montgomery, E. A. Peck, and G. G. Vining, "Introduction to linear regression analysis," in *Introduction to Linear Regression Analysis*, 6th ed. Hoboken, NJ, USA: Wiley, 2021.
[36] F. C. Pampel, "Logistic regression: A primer," in *Logistic Regression: A Primer*, 1st ed. Thousand Oaks, CA, USA: Sage Publications, 2020.
[37] P. Mohassel and Y. Zhang, "SecureML: A system for scalable privacy-preserving machine learning," in *2017 IEEE Symp. Secur. Priv. (SP)*, San Jose, CA, USA, IEEE, 2017, pp. 19–38.