Piranha: A GPU Platform for Secure Computation

***Abstract*—Secure multi-party computation (MPC) is an essential tool for privacy-preserving machine learning (ML). However, secure training of large-scale ML models currently requires a prohibitively long time (e.g. weeks) to complete. At the same time, large ML inference and training tasks in the plaintext setting are significantly accelerated by Graphical Processing Units (GPUs).**

We present Piranha, a modular platform for accelerating privacy-preserving protocols using GPUs. Piranha contributes three distinct layers: (1) a *device layer* that implements integer-based kernels for MPC operations and allows protocols to execute entirely on the GPU, (2) a modular *protocol layer* that support memory-efficient computation and reusable protocol components, and (3) an *application layer* where we implement a protocol-agnostic library for secure training and inference of neural networks. We show that our GPU kernels can improve the performance of MPC functionalities by *over two orders of magnitude* for common functionalities such as matrix multiplications, convolutions, and comparisons. We demonstrate Piranha by implementing 2-party (IEEE S&P '17), 3-party (PETS '21), and 4-party (USENIX Security '21) MPC protocols for secure NN training; we showcase a $16 - 48\times$ decrease in training time compared to a CPU-based implementation. Finally, secure training of large-scale neural networks has been unattainable due to prohibitive computational costs. For the first time, Piranha demonstrates the feasibility of *securely training a neural network with over 100 million parameters, in a little over one day.*

I. Introduction

Applications like machine learning (ML) have enjoyed tremendous success in automating tasks such as biometric authentication, personalized ad recommendation, or detecting fraudulent financial transactions [1–3]. However, these models come at a significant privacy cost, as the data underlying them can be highly sensitive, ranging from medical data to online behavior and financial records. This has incentivized the development of privacy-preserving approaches to ML [4–6].

Secure Multi-Party Computation (SMC/MPC) has emerged as a promising tool for privacy-preserving computation [4, 7–9]. MPC enables a group of entities to perform a joint computation without revealing their inputs to the computation. Thus, when data is sensitive, MPC can enable a the group of entities to generate insights from this data (such as training ML models or performing inference) without ever disclosing the data in plaintext to the other parties involved. MPC has shown tremendous progress in the past few years, making significant algorithmic improvements [10–13] as well as robust, efficient, and versatile implementations [14–17]. However, despite these advances, the overhead of MPC remains prohibitive when considering large computations. For instance, secure training of large machine learning models is over 4 orders of magnitude slower than plaintext training [18].

In the plaintext setting, large ML inference and training tasks are made practical by the use of GPUs – many-core hardware accelerators that support highly-parallelizable workloads. Individual operations, or kernels, are tiled across the many GPU processor threads to minimize execution time over large input data. For example, the use of GPUs can improve the training times of commonly used ML models by $10 - 30\times$ [19], making them an essential tool in today's ML infrastructure. This raises a natural question – can secure computation similarly benefit from GPU acceleration?

Improving the performance of MPC computation through the use of GPUs is not straightforward. The cryptographic nature of MPC computation and the presence of secret-shares make the protocols used for secure computation vastly different from their plaintext counterparts. For instance, comparing two plaintext values consists of a simple kernel operation whereas performing the same operation privately between two secret-shared values commonly requires multiple rounds of communication and a significant computation [20, 21]. While few recent works [13, 22] have looked into using GPUs to improve the performance of secure computation, they are limited to accelerating a *specific* MPC protocol. It is untenable for each new MPC protocol to re-implement their own GPU support, as it would require domain-specific knowledge of GPU programming, extensive low-level optimization and debugging, and result in the duplication of development efforts.

We present Piranha, a modular GPU-based framework for secure computation. Piranha supports a variety of linear secret-sharing schemes (LSSS), which encompasses a large number of state-of-the-art protocols for secure computation [18, 21, 23–27]. Piranha contributes three distinct, modular layers that provide a separation of concerns for GPU-accelerated private computation (Figure 1): a *device* layer that implements GPU kernels to accelerate various operations over local data buffers; a *protocol* layer that implements different MPC protocols, their secret-sharing schemes, and adversarial models; and an *application* layer that uses these protocols in an agnostic manner to perform a desired high-level computation. In sum, Piranha allows an MPC developer to benefit from GPU acceleration without developing expert knowledge or re-implementing GPU support from scratch.

We demonstrate the use of Piranha in implementing three different MPC protocols for secure neural network training – the 2-party SecureML [28], 3-party Falcon [18], and 4-party Fantastic Four [26] protocols. We plug these protocols into a high-level neural network library to provide GPU-assisted private training and inference of ML models. Compared to state-of-the-art CPU implementation [29] of computational building blocks such as matrix-multiplication, convolutions, and comparisons, Piranha improves runtime by 2 to 3 orders of magnitude. When considering end-to-end run times for private training of ML models, our GPU-accelerated implementation

is approximately 16-48× faster than CPU-only implementations [18,29]. Finally, *we are the first work to demonstrate the practicality of secure end-to-end training of neural networks, from scratch, with over 100 million parameters.*

A. Design Overview

The use of GPUs for secure computation poses a few challenges – the need to accelerate MPC operations in a protocol-independent manner, provide protocols with a simple, integer-based interface, and efficiently manage the limited availability of on-device memory offered by commodity GPUs. We design Piranha to address these issues while providing a broad scope for using the platform in various secure computation tasks.

GPU-based data interface. A natural, naïve approach to porting MPC protocols is to outsource only a few expensive operations to the GPU, or *device*, while maintaining the bulk of protocol execution on the CPU, or *host*. Unfortunately, this yields high overheads from data transfer, resulting in weak performance improvements, if any. Our approach is to provide MPC protocols with an interface to local vectors, or *shares*, of on-device data. This allows Piranha to handle data transmission between parties and transparently manage data allocation. This ensures that MPC protocol data remains on the GPU for the entirety of the computation. We require vector shares as the basic unit of computation because it ensures that any protocol or application implemented over the device layer can inherently take advantage of the GPU's parallelism. Finally, one of Piranha's key insights is that, if executing an MPC computation predominantly consists of computation over these shares, accelerating local operations yields significant performance benefits in a protocol-independent manner.

We specifically design the GPU-based shares for use by MPC. As arithmetic secret sharing-based protocols operate over rings or fields, MPC applications largely eschew floating-point data types in favor of values encoded into integer shares via fixed-point precision. Unfortunately, current state-of-the-art GPU kernels focus exclusively on floating point-based operations [30], targeting plaintext graphics and ML workloads. To address this issue, Piranha explicitly provides for integer-based shares and matching GPU integer kernels to accelerate common operations.

Memory-efficient computation. While modern CPUs boast terabytes of RAM for computation, present-day GPUs are constrained to a severely limited pool of available memory – 12 or 16 GB for commodity models. This is a salient issue for MPC, where protocols often maintain duplicated copies of data in separate secret shares, leading to a multiplicative increase in memory requirements. When paired with ML model parameters whose footprint can range in the gigabytes, even in plaintext, Piranha must make as efficient use of its limited device memory as possible. This can directly impact overall performance: in ML training, memory availability limits the total batch size – i.e. the number of data points processed in parallel – that can be supported on a single GPU.

To address this problem, we first promote in-place operations for local shares, performing additional memory allocation only when a protocol explicitly requests it. While applications like privacy-preserving ML training will always require a baseline allocation, encouraging protocols to reuse existing buffers minimizes temporary peaks in total memory usage. Second, we allow protocol implementations to define iterator-based views over local device data. These data views are critical in allowing protocols to directly perform complex operations over a local shares. In particular, this approach precludes the need to manually modify GPU data layouts, avoiding any temporary memory allocation or data transfer overhead that the computation would normally require.

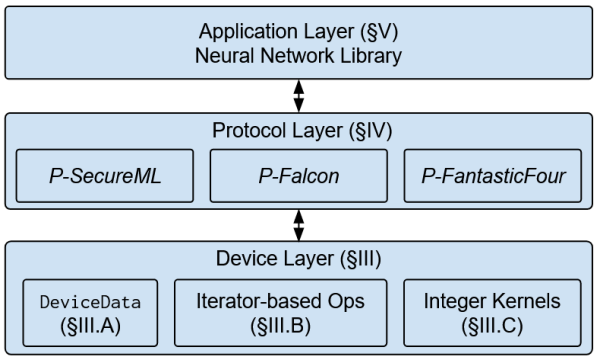

Fig. 1: Piranha's architecture. We contribute a low-level GPU device layer accelerating local, integer-based data shares. Using these shares, we implement functionality for three different MPC protocols: SecureML (2-party), Falcon (3-party), and FantasticFour (4-party). At the top-level, we provide a protocol-agnostic neural network library that can be executed by any of the implemented protocols. Piranhais modular in that it can support additional components above what we provide; in particular, MPC developers may choose to implement additional protocols in addition to those we demonstrate.

Protocol-agnostic applications. One of Piranha's primary goals is to support application execution over any of the platform's MPC protocols. We enable this by having each protocol implement the same high-level interface for applications: an abstraction for secret-shared data coupled with functionality to modify it. In this manner, the only modification applications must make to execute with a specific protocol is to simply "template" their data shares with the desired implementation. Piranha can use any of the 3 state-of-the-art MPC protocols we implement to execute our neural network training library implemented at the application layer.

A particular benefit of the platform's layer-based division is that it allows applications to focus on domain-specific MPC challenges independently of improvements to the underlying acceleration or protocol in use. In developing our training library, we investigate the ability of fixed-point encoding to both support large activation values and accurately maintain very small gradients. It is not obvious if a network will train when the inherent approximations within secure computation compound over many training iterations. Finally, we propose an MPC-friendly gradient approximation for softmax that avoids an expensive exponential calculation, while remaining stable throughout the training process.

B. Experimental Summary

Piranha demonstrates that the use of GPUs significantly improves the overall performance of MPC, enabling more efficient execution over larger data sizes in particular. Prior work estimates that the cost of training large scale neural networks such as VGG16 [31] on datasets like CIFAR10 [32] could range into the years [18]. We show that leveraging GPU acceleration can make these computations practical.

Section VI compares state-of-the-art CPU-based implementations against Piranha's protocols for core operations. Over large problem sizes, we show up to 200× runtime improvement for matrix multiplication, 300× improvement for convolutions, and 900× for ReLUs in a three-party setting. We demonstrate end-to-end training over four neural network architectures Section IV-D, completing each orders of magnitude faster than estimates in prior work. Finally, Piranha demonstrates computation over 4× larger input data by achieving the same magnitude reduction in memory overhead.

II. System Architecture

Piranha consists of three different components, shown in Figure 1, that form a framework to support GPU-accelerated secure computation. The first is a *device interface* that encapsulates the basic data types and operations MPC protocols require. We discuss how Piranha efficiently manages program data on the GPU and accelerates integer-based operations in Section III. Second, we define and implement a *protocol layer* (Section IV), whose interface is an abstraction of data secret-shared between parties, along with a set of functionalities to modify it. Finally, a protocol-agnostic application can be created on top of this interface. We demonstrate the generality of this architecture by implementing two-, three-, and four-party arithmetic secret-sharing protocols, described in Section IV-D, that independently satisfy the protocol interface.

To put Piranha in context, imagine implementing, as a privacy-preserving application, a simple fully-connected neural network layer. The core logic – management of layer parameters and implementation of its forward and backward passes – lies at the application level, implemented in terms of a protocol's opaque, secret-shared data vectors, and the corresponding functionality. In this case, our layer application would require that the protocol implement a *privacy-preserving* matrix multiplication, which it would use to compute its activations and resulting gradients. In turn, the MPC protocol used would implement its functionality as a series of *local* matrix multiplications between local device data share, accelerating each multiplication with a protocol-independent integer kernel in Piranha's device layer.

A. Device layer

The device layer accelerates local shares of GPU memory. It implements two specific components: an abstraction of a GPU-based integer vector (share), and kernels that accelerate common integer-based operations over these shares.

Section I discusses the need to accelerate integer computation over much more common floating-point kernels. For each MPC protocol Piranha implements, we analyzed the basic local operations their functionality requires, such as element-wise operators like addition or multiplication, convolution, comparisons, or matrix multiplication, and implement a set of general-purpose kernels that accelerate these operations and can be used by any protocol. We support 8-, 32- and 64-bit integer types for each kernel.

The device layer also performs communication management independently of the protocol implementations. Since data lives on the GPU throughout computation, when a local share must be sent to another party, Piranha copies the share to a temporary buffer on the CPU before transmitting over the network. The same process occurs in reverse when receiving data from a remote party.

B. Protocol layer

MPC implementations at the protocol layer leverage sets of local, device layer shares. Much like the lower-level interface, each protocol implements two components: a vector abstraction storing secret-shared data, along with a set of protocol functionalities that any application can use without requiring specialized knowledge of the MPC protocol itself. As a result, any protocol that provides an application's expected set of available functionality (e.g. matrix multiplication or comparison) can be used to perform the end-to-end computation without modifying the application itself.

Alongside individual protocol definitions, we implement some protocol functionality under the Arithmetic Black Box Model (discussed further in Section IV) that can be used to supplement any of the specific protocols, denoted as F_{ABB} in Figure 1. This demonstrates the benefit of Piranha's modular structure, in that functionality under this model can be reused in multiple protocols, and any improvement to their implementations has a wide effect on the performance of all applications using an affected protocol.

C. Application layer

Applications use the secret-shared vector abstraction exposed by implementations at the protocol layer and their associated sets of protocol functionality. As a result, applications can focus on solving domain-specific challenges such as secure training of neural networks, oblivious sorting, etc. In this work, we focus on the neural network application.

Efficiently developing and evaluating individual protocols can be challenging due to the obscured nature of the data being computed over. Piranha includes a robust set of profiling tools that can collect granular timing measurements and track memory usage to improve protocol performance. It tracks communication round-trips to identify inefficient implementations that may differ from the theoretical protocol round complexity, and can recover secret-shared values during development for analysis in plain text. We found that this capability allows significant introspection into protocol and application behavior, and speeds up the implementation process.

III. Device Layer

Effectively and easily interfacing with the GPU is a major barrier to MPC developers who wish to accelerate their protocols, but lack experience in programming optimized GPU

Listing 1 Sample DeviceData usage demonstrating its key capabilities: transparently accelerating element-wise operations (lines 7-8), using Piranha-implemented integer kernels for computation such as matrix multiplication (line 11), communicating share contents with other parties (lines 14-15), and using iterators to define views of existing data without performing a data copy (lines 18-21).

```
1  // Device share initialization
2  DeviceData<uint32_t> a = {1, 2, 3, 4, 5, 6};
3  DeviceData<uint32_t> b = {1, 0, 1};
4  DeviceData<uint32_t> c(2);
5
6  // Vectorized element-wise operations
7  a += 10;
8  a *= 2;
9
10 // GEMM call: a (2x3) * b (3x1) -> c (2x1)
11 c = gpu::gemm(a, b, 2, 1, 3);
12
13 // Communication with party id 1
14 a.send(1);
15 a.join();
16
17 // Even (offset=0) or odd (offset=1) values
18 DeviceData<uint32_t> d(
19   stride(c,2).begin()+offset,
20   stride(c,2).end()
21 );
```

kernels. Thus, a flexible abstraction is needed to support a wide array of MPC protocols while minimizing any domain-specific knowledge required. In this section, we discuss how Piranha addresses two primary challenges in providing extensible GPU support for MPC protocols: managing vectorized GPU data and supporting acceleration for integer-based computation.

A. Data management on the GPU

Piranha provides access to GPU memory through a single data abstraction we call a DeviceData buffer. A key property that DeviceDatas maintain is that their data resides only on the GPU; no buffers are maintained in CPU memory to avoid data transfer overhead when computing with GPU-based kernels. In the context of MPC protocols, these buffers often logically correspond to local copies of a secret share. A DeviceData can be templated by integral C++ data types such as uint32_t or uint64_t. Share *vectors*, not individual share values, are the basic unit of computation in Piranha, and so the abstraction is functionally equivalent to a std :: vector<> class, except that the data remains on-device. Listing 1, lines 2-4 show a few examples of how DeviceData vectors can be initialized.

Element-wise operations over collections of secret-shared values are common in secure computation. As a result, they are prime targets to accelerate in parallel, enabling the GPU to naturally improve protocol performance. As an added benefit, by using vectorized DeviceData shares, developers at the protocol layer inherently parallelize their protocol implementation. Piranha's device interface supports a variety of local operations on individual share vectors; as a simple example, lines 7 and 8 of Listing 1 perform an accelerated element-wise scalar addition and multiplication, with each value modified in parallel by a different GPU kernel thread.

A primary insight Piranha makes is that, independent of the specific protocol, MPC functionalities over secret-shared data decompose into a common set of local arithmetic operations. It is this narrow waist that the device layer targets for acceleration in a way that can benefit every MPC protocol. Consider a widely used primitive, secure matrix multiplication, that decomposes into simple matrix multiplications and additions over local data in a protocol-agnostic way. To this end, Piranha provides integer kernels for performing general matrix multiplication (GEMM) over the DeviceData class, which we use to build secure matrix multiplication protocols (cf. Section IV for an example). An individual GEMM call is shown in Listing 1, line 11. In Section VI, we evaluate how these kernels improve the performance of secure matrix multiplication by up to $200\times$ over a CPU-based implementation.

A note on communication. Currently, support for direct GPU-GPU communication over the network is nascent and not widely available. Thus, in Piranha, communication between GPUs is bridged via the CPU, incurring a data copy overhead for each round of communication. Given that GPU-CPU data transfer speeds are significantly faster than communication over the network, this overhead is not significant in the applications we consider. We manage communication by abstracting this complexity away from MPC developers by providing simple data transmission functions. A sample communication round to a different machine is shown at Listing 1, lines 14 and 15. In the background, Piranha copies the values in DeviceData a to a temporary CPU buffer, and transmits it over the network. The protocol execution can then wait until the buffer has been successfully sent by calling join() to synchronize protocol execution.

B. Iterator-based operations

Another key design criteria for Piranha's device share abstraction is memory efficiency. While CPU-based protocols have enjoyed "effectively" unlimited memory availability, realistic GPU-based MPC computation is restricted to commercially-available GPUs that generally have around 16 GBs of memory. Given the increase in memory consumption required by secret-shared protocols, the result of inefficient memory usage is to unnecessarily limit application problem sizes. Furthermore, the overhead of data allocation, particularly for vectors of large sizes, forms a significant portion of the total overhead of using GPUs. To address this issue, we seek to avoid any redundant temporary data allocation used to transform data into a specific layout for kernel execution. We achieve this using an iterator-based abstraction in our DeviceData class, as follows.

Piranha's iterators allow the developer to traverse data vectors in a program-defined order, applying operations over a "view" of GPU memory decoupled from the actual physical data layout. For instance, a common operation requires pairwise operation over elements of a vector (cf Section IV for details), i.e., operations over $\mathsf{vec}[2\mathsf{i}], \mathsf{vec}[2\mathsf{i}+1]$ for a given vector vec and over all indices i. A naïve approach would either require copying the odd and even components of the vector or to allocate new memory for storing the result. Our iterator-

based approach allows us to define odd and even views over the same vector that effectively allow the GPU to interpret the memory with a stride of 2. This abstraction enables memory efficient code design by allowing us to view a given memory allocation in different ways. Hence, this approach encourages limited additional memory allocation – performing in-place element-wise operations as well as storing the computation result in existing memory.

Lines 18-21 of Listing 1 demonstrate this concept. The two DeviceData vectors even and odd hold a view of all the values in c at a stride of 2, or put otherwise, skipping every other value (odd starts at index 1). Note that this is simply a "view", i.e., even, odd operate on the same physical memory held by the original DeviceData c. Creating this view for every other indexed value allows a pairwise computation to be performed *with no additional memory allocation required*.

C. Integer kernels

The MPC protocols we implement in Section IV-D operate on additive secret sharing over 32- or 64-bit ranges. As discussed in Section I, there is a lack of kernel implementations for these data types [33], because prior work has focused on improving the performance for floating point data types. Some integer kernels are implemented for 8-bit matrix multiplications into 32-bit accumulators, for example, but the lack of support for larger integer types can be attributed to concerns of overflow in the product. Thus, there are two ways to benefit from GPUs for large bit-width integer types.

The first is to decompose large integers into multiple values of smaller width, such as 16 bits, representing the original value $x = x_3 2^{48} + x_2 2^{32} + x_1 2^{16} + x_0$. Computation can then be performed over each 16-bit sub-value x_3, x_2, x_1, x_0 by embedding them into 64-bit floating point types. Note that a large slack is required, as the result of multiplying matrices of 16-bit values will often exceed 32 bits in size and floating point computation does not have the same modular overflow as for integers. The problem with this approach is that it requires multiple individual floating point kernel calls over 16-bit values to compute one 32- or 64-bit integer result.

The second approach, which we take, is to directly implement kernels over integer data types. Piranha directly adds support for full-size integer matrix multiplication and convolution kernels at the device layer. We use the general-purpose templated matrix multiplication and convolution kernels in CUTLASS [34] to support 32- and 64-bit integer types.

While we cannot use existing, highly optimized floating-point GPU kernels such as those provided by cuBLAS [33], there are two benefits to our approach: (1) Piranha's modular structure allows independent improvement of kernels, and thus future hardware support for large integer operations on GPUs can be easily integrated and benefit all pre-existing protocols, and (2) the ability to directly compute integer results in a single call to a GPU kernel yields a better performance overall than multiple calls to a more efficient floating point kernel. We demonstrate these gains in Section VI and Appendix A.

IV. Protocol layer

Piranha provides a framework for implementing various MPC protocols leveraging the benefits of GPU acceleration. In this section, we first describe how we use Piranha's DeviceData class to implement MPC protocols. We then highlight how complex protocols can be parallelized in a memory-efficient manner, and how Piranha allows for functionality reuse between protocols.

A. MPC protocol implementation

Any protocol implemented in Piranha specifies two things: the secret sharing base, including the adversarial model, and operations over this secret sharing base. For example, suppose an MPC developer seeks to implement a 3-party protocol using replicated secret sharing for an honest majority of corruptions (semi-honest corruptions). In this setting, a secret value x is composed of 3 shares $x \equiv x_0 + x_1 + x_2$, where each party holds only 2 of the 3 shares. Thus, the class for such a protocol will contain two DeviceData objects, one per share. Simple operations such as additions can be specified component-wise, leveraging the underlying GPU layer as shown in Listing 1.

For more complex operations, we need to implement an MPC functionality in this protocol, for example, as in the case of a secure matrix multiplication. To multiply two secret matrices x, y, if the first party holds shares (x_0, x_1), and (y_0, y_1), the protocol leverages the fact that

$$\begin{aligned} x \cdot y &= (x_0 + x_1 + x_2) \cdot (y_0 + y_1 + y_2) \\ &= (x_0 \cdot y_0 + x_0 \cdot y_1 + x_1 \cdot y_0) + (...) + (...) \end{aligned} \tag{1}$$

and thus the computation can be split such that the first term can be computed locally by the first party (and similarly for the other parties). Leveraging the device layer for each individual local GEMM computation (cf. Listing 1 line 11), the overall secure matrix multiplication protocol can be easily implemented as shown in Listing 2. This example shows the ease of implementing various MPC functionalities in Piranha's protocol layer by building over the local functionality at the device level. In Section VI, we directly evaluate the performance benefit of this implementation against a similar CPU-based protocol for secure matrix multiplication.

B. Memory-efficient protocol implementation

Section III demonstrates an iterator-based implementation for DeviceData buffers. In this section, we showcase how this abstraction can be used to perform efficient in-place memory computations. As an example, we consider a CarryOut protocol, that securely computes the carry bit for binary addition i.e., given the bitwise sharing $(a_{k-1}, \cdots, a_0)$ and $(b_{k-1}, ..., b_0)$ of two k-bit vales a, b, the goal is to compute the carry bit at the MSB c_k. This primitive forms the backbone of nearly every state-of-the-art comparison protocol [12, 20, 21, 35]. In the case of the neural network library we discuss in Section V, comparisons enable standard activation functions and pooling operations including ReLU and Maxpool.

The computation proceeds in $\log_2 k$ rounds by emulating a simple carry-lookahead adder [36, 37]. As part of the compu-

Listing 2 A replicated secret-sharing protocol class (3-party setting) implemented at the Piranha protocol layer. The protocol specifies the secret-sharing base: each party has two local DeviceData shares templated by type T. The matmul functionality is performed for this class by implementing a secure matrix multiplication based on Eq. 1.

```
// Replicated secret sharing class
class RSS<T> {
    DeviceData<T> shareA;
    DeviceData<T> shareB;
}

void RSS<T>::matmul(RSS<T> a, RSS<T> b,
                    RSS<T> c, ...) {
    DeviceData<T> localC;

    localC += gpu::gemm(a.shareA, b.shareA, ...);
    localC += gpu::gemm(a.shareA, b.shareB, ...);
    localC += gpu::gemm(a.shareB, b.shareA, ...);

    // Reshare and truncate localC to c
}
```

tation, at round $i \in \{1, 2, \cdots, \log_2 k\}$, the CarryOut computes the AND between adjacent propagating bits, i.e.,

$$p'_j = p_{2j} \wedge p_{2j+1} \tag{2}$$

where p_j are propagation bits at round i and p'_j are the propagation bits for the next round. At the end of $\log_2 k$ rounds, the final bit is the result of CarryOut.

A naïve implementation of the above will suffer from two major inefficiencies. First, bitwise expansion requires that each secret-shared bit be stored separately, increasing the memory footprint on the GPU. Second, using contiguous allocations to separate pairwise bits results in non-trivial overhead from additional memory use and data copies. Figure 2(a) shows 3 rounds of this CarryOut operation implementation where the propagating p bits are combined. Unfortunately, due to the vectorized nature of data computation on the GPU, half of p is copied at each step to a different memory allocation before the next round can be evaluated. During one execution of this particular CarryOut implementation, $\log(n)$ additional data copies are performed.

Piranha allows us to circumvent these issues by implementing a functionality that executes purely in-place. The iterator-based device layer allows this protocol layer to define views over pre-allocated data. Figure 2(b) demonstrates an memory-optimized version of CarryOut leveraging this ability. For each round, the protocol defines two iterators, one for every even element (yellow values), and one for every odd element (blue values), and uses those as the basis for executing a kernel computing the next values of the propagation bit. Furthermore, we can reuse the first iterator to store the results in the original allocated buffer, resulting in no additional data copies or memory allocation. Since the entire bitwise vector is allocated until the end of the protocol, we continue to use (increasingly less of) it to store intermediate results until the final carry bit is calculated. Note that depending on the specific protocol, each blue or yellow value is duplicated across multiple local shares and the computation is done in

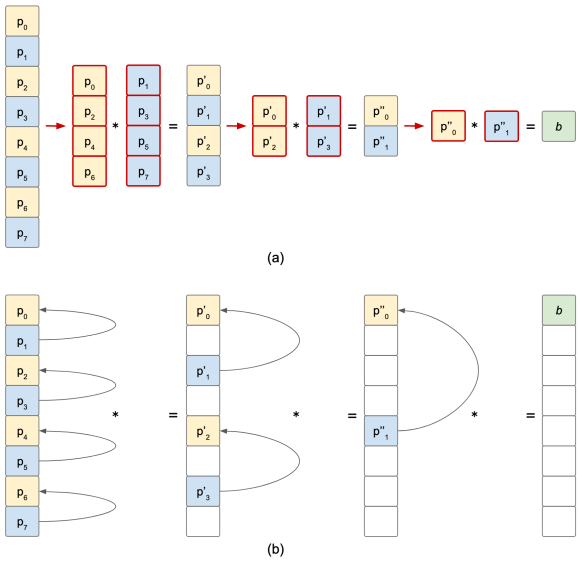

Fig. 2: Comparison of a memory-inefficient naive carryout implementation Figure 2a and our iterator-based in-place computation Figure 2b. In the former approach, new memory allocations and data copy – highlighted in red – are done to split pairwise elements into contiguous vectors for parallel GPU processing. The ability to define iterators and execute kernels over non-contiguous memory allows Piranha to avoid any additional memory allocation.

parallel to each. As a consequence, the *same functionality can be implemented efficiently with no additional memory usage*. Furthermore, templating allows the bit-vectors to use smaller datatypes (say uint8_t) compared to the datatypes used in the secure computation (say uint64_t), thus optimally allocating memory on the GPU.

C. Reusable protocol components

The structure of Piranha supports reusing protocol implementations, so that protocols can build on other implementations in a number of ways. For instance, a new protocol for secure comparison that operates in the same setting as another implemented protocol in Piranha can focus solely on implementing the secure comparison functionality and inherit the rest from the existing share type in Piranha. This also helps in maintaining the compatibility at the application layer.

Another reusable component of Piranha is the implementation of share agnostic functionalities. For example, this includes protocols that have been proven secure in the arithmetic black box model $\mathcal{F}_{\mathsf{ABB}}$ [38]. Such protocols are specified agnostic to the specific adversarial model and remain the same as long as the basic operations are performed securely in the specific adversarial model. A number of cryptographic primitives and functionalities are proven secure in this model [20,21,39]. Piranha allows such methods to be generically implemented once at the protocol level, alongside protocol-specific functionality, and can then be inherited by any other implementation. As two examples, we implement a state-of-the-art comparison protocol by Makri *et. al.* [21] and a protocol for approximate square-root and inverse computation based on [18].

Secure comparison. We use secure comparison as an example of implementing a method in the arithmetic black box model.

The comparison protocol uses edaBits [20] as preprocessing material to efficiently compute a comparison of secret values. An edaBit is an secret sharing of a random value and the bit decomposition of the same value as boolean shares i.e.,

$$\mathsf{edaBit} : [r]_M, [r_0]_2, [r_1]_2, \cdots, [r_m]_2 \text{ where } r \xleftarrow{\$} \mathbb{Z}_M \quad (3)$$

where $m + 1 = \log_2 M$. The protocol for generating this can be found in [20]. The problem of secure comparison over arithmetic secret-sharing can them be converted to a secure comparison over boolean secret-sharing using the edaBit. The latter can then be implemented efficiently using bitwise operations such as CarryOut [36,37]. Details of this operation are presented in Section IV-B.

Approximate computations. The privacy-preserving neural network application we implement requires a pair of specific protocols for the normalization layers: secure integer division and secure computation of a square root. MPC protocols for these primitives typically require approximate computation using Newton's methods. We write a generic functionality based on the protocols from [18,40] where we find the nearest power of two for each input value and then evaluate a fixed-point Taylor series polynomial approximation. We use a simple Python script to compute polynomials of a given degree that approximate each target function, in this case, sqrt and inverse. These functionalities are then implemented and used across different protocols. Specifically, Piranha uses the following approximations:

$$\begin{aligned} \mathsf{sqrt}(x) &= 0.424 + 0.584(x) \\ 1/x &= 4.245 - 5.857(x) + 2.630(x^2) \end{aligned} \quad (4)$$

These approximations achieve an L1 error of 0.00676 and 0.02029, respectively.

D. MPC protocols

We implement three different MPC protocols to showcase Piranha's generality at the protocol layer: a 2-party implementation based on SecureML [28], a 3-party implementation built upon Falcon [18], and a 4-party protocol [26]. We briefly describe each of these protocols below.

Two-party protocol (P-SecureML). In 2017, Mohassel and Zhang [28] proposed a 2-party (and a trusted third party variant) protocol for privacy-preserving machine learning, using a 2-out-of-2 arithmetic secret sharing as the basis for its functionality. The linear layers are computed using Beaver triples and the non-linear layers are evaluated with garbled circuits. In our implementation, we replace the expensive GC-based evaluation of ReLUs with a more recent and efficient comparison protocol using edaBits [20, 21].

Three-party protocol (P-Falcon). We build a 3-party protocol using the work of Wagh *et. al.* [18]. It uses a 2-out-of-3 replicated secret-sharing as the basis for its functionality. The linear layers are performed using local multiplications and resharing, a technique used in many other 3PC frameworks [12, 41, 42]. The non-linear layers are computed using a specialized comparison protocol building upon [27]. Once again, we replace the comparison protocol using the more efficient work by Makri *et. al.* [21].

matmul(...)	Matrix multiplication of two matrices.
convolution(...)	Convolution of two tensors.
maxpool(...)	Compute the maximum of set of values.
truncate(...)	Truncate i.e., divide shares by power of 2.
reconstruct(...)	Opening of secret shares.
selectShare(...)	Select one out of two shares given a boolean secret shared value.
comparison(...)	Compare two shares.
sqrt(...)	Compute an approximate square root.
inverse(...)	Compute an approximate fixed-point inverse.

TABLE I: Functionalities required by the neural network training application, implemented by each class in Piranha's protocol layer.

Four-party protocol (P-FantasticFour). Our 4-party implementation follows the work of Dalskov *et. al.* [26]. It uses 3-out-of-4 replicated secret sharing: linear layers are performed using a generalization of the replicated secret sharing approach, thus using a combination of local multiplications and resharing (known as joint message passing and INP in the work and similar to [43]). For comparison (probabilistic truncation), the protocol uses a combination of [44] and [43].

V. Application Layer

Our final layer of abstraction is the neural network layer. This interface is guided by the types of the deep learning architectures we wish to support. Currently, Piranha implements protocol-agnostic versions of the following layers in full generality:

(1) Linear layers: Convolution and fully-connected layers
(2) Pooling operations: Maxpool and averagepool
(3) Activation functions: ReLU
(4) Normalization: Layer normalization

Layers use the popular Kaiming weight initialization [45].

As a result, any neural network architecture that is composed of these layers can be run using Piranha. This covers a large class of popular networks used in computer vision - from simple multi-layer perceptrons like SecureML [28] to more complex convolutional neural networks such as AlexNet [46] and VGG16 [31]. In our evaluation in Section VI, we compare Piranha to the networks used in prior works [18, 22].

A. Interfacing the neural network library

As discussed in Section V, we focus on the secure evaluation of neural network models as our target application. To support the neural network library over multiple MPC protocols, we require each MPC protocol to implement a common set of functionalities. Once this set is implemented, the protocol can support training and inference over any neural network architecture constructed with the supported layers. This required set of MPC functionalities is given in Table I.

Listing 3 shows a simplified look at the forward pass of a fully connected layer. The functionality simply takes a batch of inputs, multiplies them with the layer weights and adds the layer's bias to compute the activations. The forward pass implementation is protocol-agnostic in that it can be templated

Listing 3 Protocol-agnostic implementation of a fully-connected neural network layer. Any protocol class, such as the RSS class in Listing 2, that implements the desired matmul functionality can be used to compute the forward pass.

```
// Fully connected layer forward pass
template<typename Share>
void FCLayer<Share>forward(Share input) {

    matmul(input, this->weights,
           this->activations, ...);

    this->activations += this->bias;
}
```

with any given Share type (e.g. from Listing 2's RSS share) and requires only that the required functionality matmul be implemented by that protocol.

B. Secure training of neural networks

Training neural networks, especially larger and deeper networks presents a number of challenges. In order to demonstrate learning, we face three major challenges:

(1) Back propagation gradients are frequently much smaller than the remaining activations and must be preserved by the finite precision available in fixed-point integers.
(2) The quality of the gradients can also significantly affect the training process. Ensuring that the final layer gradient computation is accurate has a significant impact on how well the network trains. Inaccuracies are compounded by linear layers, which yield approximate values due to each multiplication performed with finite precision arithmetic.
(3) Closely related to the previous issue is the stability of the final layer gradients. As the network trains, the magnitudes of the final layer activations grow in size. Softmax computations to generate the needed gradients (which involve an exponentiation) can quickly exceed the size of the data type, yielding an overflow and destabilizing the learning process.

We showcase in Section VI that privately training neural networks is indeed possible for large networks with over 100 million parameters. We use fixed-point arithmetic to encode real numbers for neural network experiments. For private inference, we observe that the neural network can be run over 32-bit data-types with a fixed-point precision of 13 bits. However, for private training, to retain the gradients with sufficient precision, we use 64-bit data types with 20 or more bits of fixed-point precision. Finally, to address latter challenges, we propose a new gradient computation function. Our gradient computation has two main advantages: it is more stable to large activations, and it is MPC-friendly. The first is achieved because we approximate the exponential with a function that does not increase the magnitude of the secret-shared values. The second is achieved by using only comparisons, which significantly reduces the round complexity of the computation.

Gradient Computations. In order to compute the gradients for the backward propagation, we apply a softmax coupled with the cross-entropy loss function. Suppose the output of the last layer is $\mathbf{x} = (x_0, \cdots, x_9)$, and $\mathbf{y} = (y_0, \cdots, y_9)$ is a one hot encoding of the true label, then the loss function (per image) is given by:

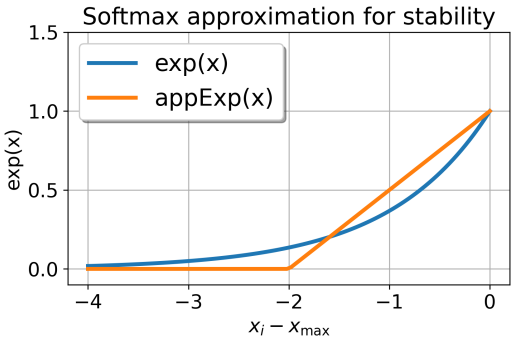

Fig. 3: Our new approximate computation of last layer gradients that stabilize the learning process.

$$\ell = -\sum_i y_i \log p_i \quad \text{where} \quad p_i = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{5}$$

The gradient is then given by:

$$\nabla_i = \frac{\partial \ell}{\partial x_i} = p_i - y_i \tag{6}$$

While there are a few different ways to compute this gradient [47], they do not solve the challenges mentioned above, which are critical when training is performed on larger networks and datasets. Note that the softmax function remains the same if the logits p_i are computed using the activations $x_i - x_{\max}$ where $x_{\max} = \max(x_1, \cdots, x_k)$ if k is the number of classes. In other words,

$$p_i = \frac{e^{x_i - x_{\max}}}{\sum_{j=1}^{k} e^{x_j - x_{\max}}} \tag{7}$$

We propose a new function computation to approximate the above computation (Eq. 7):

$$p_i \approx \mathsf{appExp}(x_i - x_{\max}) / \sum_{j=1}^{k} \mathsf{appExp}(x_j - x_{\max}) \tag{8}$$

where $\mathsf{appExp}(\cdot)$ is the approximate exponential function as shown in Fig. 3. We compute the inverse in plaintext using a functionality similar to FALCON. To preserve the long tail of the exponential, we add a small bias of 10^{-3} to each component of $\mathsf{appExp}(\cdot)$. Note that this function is (1) relatively easy to compute within MPC, and (2) preserves (i.e., does not increase) the magnitude of the activations. These factors make the gradient computations using this function stable from the machine learning perspective.

VI. EVALUATION

In our evaluation, we answer the following questions:

(1) *In comparison to state-of-the-art, CPU-based prior work, how well does* Piranha *accelerate the same computation tasks?* (Section VI-B)
(2) *Can* Piranha *be used to securely train large neural networks (over 100 million parameters) in a reasonable amount of time?* (Section VI-C)

(3) *How well does* Piranha *manage constrained GPU memory and how well does its memory-conscious design improve scalability at the application layer?* (Section VI-D)
(4) *How does the performance of privacy-preserving inference and training implemented on* Piranha *compare with prior work?* (Section VI-E)

A. Evaluation set-up

We run our experiments over similar hardware and networking environments as prior works [18, 26, 28]. For CPU-based implementations, we use Azure F32s_v2 instances with Intel Xeon Platinum 8272CL @ 3.4GHz processors, 32 cores, and 64 GB of RAM. Networked experiments are executed in a LAN setting with a bandwidth of 10 Gbps and ping time of 0.2 ms. GPU-based experiments are run on Azure NC6s_v3 instances with 6-core Intel Xeon E5-2690 v4 CPUs with 112 GB RAM and Nvidia Tesla V100 GPUs with 16 GB RAM.

We add matrix multiplication and convolution kernels for large integer types by building on CUTLASS [34], at commit 0f10563, to which we add support for 32- and 64-bit integer matrix multiplication and convolution. We use the default tiling parameters, while element-wise kernels are parallelized using Thrust [48].

Baseline. As a baseline, we compare against protocol implementations from MP-SPDZ [14, 29] at commit e6dbb4. MP-SPDZ is a state-of-the-art open-source secure computation platform with over 34 protocols and represents a CPU-based analog to Piranha. For each MPC protocol that we implement, we choose a state-of-the-art protocol implemented by MP-SPDZ in the same setting: individual operations are benchmarked in Section VI-B with the 2-party semi2k, 3-party replicated-ring, and 4-party rep4-ring protocols.

Models and Datasets. We evaluate our high-level neural network library with four neural network architectures: SecureML [28], a simple 3-layer network, and LeNet [49], a 5-layer convolutional network, over MNIST [50], and AlexNet [46], an 8-layer convolutional network, and VGG16 [31], a 16-layer convolutional network, over the CIFAR10 dataset [32]. While Piranha fully supports the use of maxpool layers in these architectures, as in CryptGPU [22], we substitute them with averagepool layers to maintain comparative accuracy. Notably, averaging operations are much less computationally expensive in each Piranha framework than max operations, avoiding a significant number of calls to secure comparisons.

B. Performance Comparison of Piranha vs. CPU

In this section, we compare the performance of Piranha with state-of-the-art CPU-based protocols over a set of MPC workloads. For each protocol discussed in Section IV-D, we execute individual operations commonly used by a secure neural network application – matrix multiplications, convolutions, and ReLU comparisons – and compare against the same operations computed using MP-SPDZ [29] with protocols in the same setting, as described in Section VI-A. In general, our results find that Piranha's acceleration can improve performance by *2-3 orders of magnitude* for these important MPC functionalities. Figure 4 summarizes the results for each these operations as a function of various problem sizes.

We evaluate matrix multiplication performance by multiplying two $N \times N$ matrices for logarithmically-increasing values of N. Considering small matrices of dimension $N = 10$, where platform overhead such as data transfer to the GPU is most likely to have an out-sized impact on overall performance, we find that using Piranha results in a performance benefit of 6 to 60× in the four- or two-party settings, respectively. Likewise, as the problem size increases, so does the impact of GPU acceleration on runtime. For the largest matrix multiplication benchmarks with $N = 300$, Piranha's 3- and 4-party protocols improve on the CPU-based MP-SPDZ implementations by 2 orders of magnitude, while P-SecureML shows a 4 order of magnitude improvement over MP-SPDZ's semi2k implementation.

For the convolutions, we benchmark problems in order of increasing complexity. Each convolution layer is parameterized by a $[iw, c_{\mathsf{in}}, c_{\mathsf{out}}, f]$ tuple, where iw is the input image dimension, c_{in} and c_{out} are the number of input and output channels, respectively, and f is the filter size. We use the total number of multiplications as a proxy for layer complexity (the complexity of the resulting unrolled matrix multiplication). The specific convolutions we compute are listed in Figure 4, ranging in complexity from 1.47×10^7 to 1.86×10^9 multiplications. Similar to the matrix multiplication benchmarks, Piranha shows a significant improvement in performance, performing on average 175 and 73× better in the 3- and 4-party setting, respectively. Piranha is much faster than the MP-SPDZ 2-party semi2k implementation, achieving a speed up of 3 orders of magnitude, on average.

Finally, ReLU operations are benchmarked over N-element vectors of logarithmically increasing size. For small vectors of $N = 10$ vectors, Piranha improves on each CPU-based protocol by between 1.3 and 5.5×, again seeing modest gains due to overhead dominating the relatively simple computation. For large vector sizes, we show extensive gains by applying GPU acceleration. Figure 4 shows between a 300 and 1380× speedup across MPC protocols over large ReLU inputs, completing 90 second CPU-based operations in less than a second.

C. Secure Training of Neural Networks

No prior work has successfully trained, within secure computation, a network such as VGG16, which over CIFAR10 has over *100 million learnable parameters*. While existing work has estimated the time to train such a network, the training times are prohibitively large – over 14 days [18] to complete 10 training epochs. This work is the first to securely train such a neural network, in less than a day and a half: our results are detailed in Table II.

We train each network with each protocol Piranha currently supports for 10 epochs with 128-image batches. For each training run, we report the total training time and per-party communication. Every training pass used the MPC-friendly softmax replacement we propose in Section V; over every network architecture we evaluate, our approximation remains

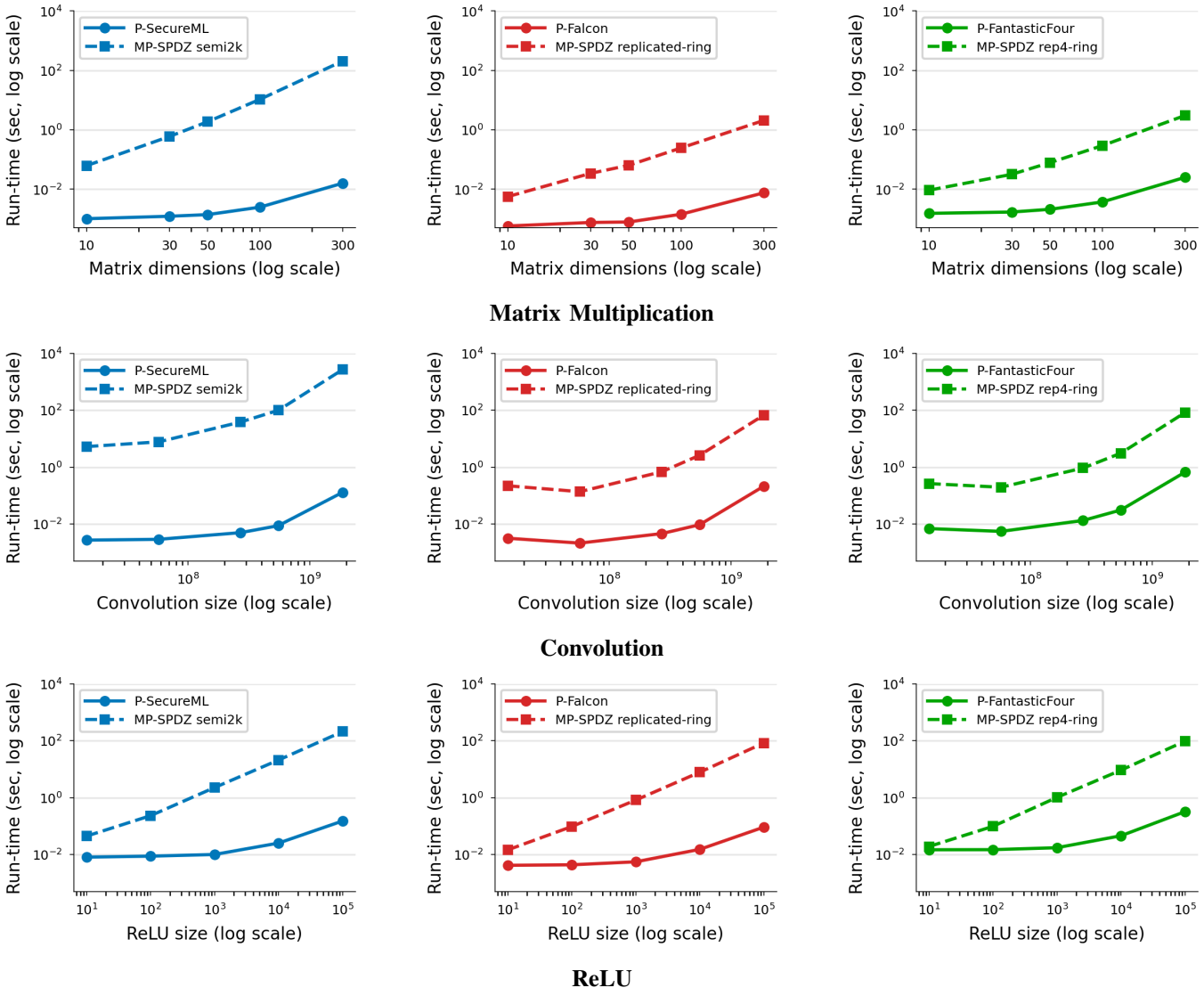

Fig. 4: The figures benchmark secure protocols for matrix multiplication, convolutions, and ReLU across 2-, 3-, and 4-party protocols for various sizes of these computations. Piranha consistently improves the run-time of these computations, with improvements as large as 2-4 orders of magnitude for larger computation sizes.

Network (Dataset)	Protocol	Time (min)	Comm. (GB)
SecureML (MNIST)	P-SecureML	12.99	49.55
	P-Falcon	7.51	22.84
	P-FantasticFour	23.39	33.01
LeNet (MNIST)	P-SecureML	87.55	683.18
	P-Falcon	71.56	485.90
	P-FantasticFour	219.20	676.13
AlexNet (CIFAR10)	P-SecureML	153.63	740.50
	P-Falcon	124.33	382.18
	P-FantasticFour	296.57	533.74
VGG16 (CIFAR10)	P-SecureML	3336.95	31053.66
	P-Falcon	1979.92	17235.35
	P-FantasticFour	7451.25	27001.08

TABLE II: Time and communication costs for completing 10 training iterations over four neural network architectures, for each of Piranha's MPC protocol implementations. We are the first work to demonstrate secure training of VGG16, a network with over 100 million parameters, in less than a day and a half for P-Falcon.

stable and allows the networks to train successfully. For deeper networks, we observe that gradients reaching the initial layers routinely approach 2^{-20} and are further reduced by the current learning rate. If the fixed-point precision used by the network is not selected carefully, parameter update gradients will approach the minimum value Piranha can represent, yielding imprecise results and barring the model from training correctly. As a result, we train the shallow SecureML with 20 bits of fixed-point precision, LeNet and AlexNet with 23 bits, and VGG16 with 26 bits of precision.

On a small dataset like MNIST, Table II shows that Piranha's neural network training library can quickly train SecureML and LeNet, achieving greater than 96% test accuracy in no more than 2 hours with P-Falcon and P-SecureML. As is expected from the more complex P-FantasticFour, it performs the same training pass approximately 2 to 3× slower. This is a result of its 4-party setting that requires 10 local matrix

multiplication operations for every privacy-preserving matrix multiplication, compared to only 3 local multiplications in Piranha's 3-party P-Falcon implementation.

On the larger CIFAR10 dataset, training times increase significantly but remain feasible. Over AlexNet, all protocols can successfully complete their training runs in under 5 hours, achieving 40% test accuracy over that time. Given that the baseline accuracy for the dataset is 10%, this indicates that Piranha trains over even large networks. Finally, considering VGG16, the most complex network Piranha trains over, training times are considerable: P-SecureML and P-FantasticFour require 2 and 5 days, respectively, to complete. Importantly, however, we can complete 3-party VGG16 training in only 33 hours with a 54% test accuracy, which prior work estimated to take *336* hours but did not actually execute the training [18].

These training times are only possible due to two main factors. First, improved computation times (through the use of GPU-accelerated kernels) reduces the overhead of matrix multiplication and convolution, whose costs grow superquadratically with their dimensions, and are a significant part of the total runtime. The second is the ability to train over large batch sizes. Large batch sizes improve the efficiency of the stochastic gradient descent algorithm, and runtime scales better with batch sizes. Thus, a batch size of 128 has a lower run-time than computing over two 64-image batches.

D. Memory Efficiency

Commodity GPUs, including those we use to evaluate Piranha, are commonly constrained to 16GB of memory. We evaluate how effectively Piranha manages this memory constraint by tracking peak memory usage over training passes. When all other parameters are the same (protocol, computational task, and GPU hardware), prior work can only execute over batch sizes of 32 [22]. This section shows how careful memory management directly translates to executing neural network training over significantly larger batch sizes on a single GPU than has been previously possible.

First, we illustrate the benefits of two main capabilities we discussed in Section IV-B – iterator-based protocol implementations and type-based optimizations – in reducing our memory footprint and enabling realistically-sized computations. Figure 5 tracks on-GPU memory usage for P-Falcon, updated after every (de)allocation, during a VGG16 forward pass with an input batch size of 4, for three Piranha versions. Note that storing the network parameters themselves requires a static base allocation amount, in this case, approximately 345 MB, while calculating each layer's activations requires additional temporary allocations. These additional allocations can significantly strain the GPU's available memory and preclude larger batch sizes.

Figure 5(a) shows the memory allocation trace for the naive P-Falcon implementation described in Figure 2(a), which requires a significant amount of data copies while executing ReLU comparisons, where secret-shared values are expanded into bitwise format. As a result, driven by the initial network layers with larger inputs, the peak GPU memory load is 2.28 GB, an $7\times$ increase over the allocation required for the network itself. The total number of memory operations is also extremely high: almost 16,000 such allocations and frees are performed over the course of the computation.

In contrast, Figure 5(b) shows the results of an improved iterator-based implementation, that operates over views of already-allocated shares, without incurring additional memory load. In-place computation yields significant memory savings: for batches of 4 images, the iterator-based Piranha version requires only 1.38 GB at its peak compared to the base implementation of Figure 5(a). The number of GPU memory operations also drops as a result, resulting in almost $4\times$ less allocations and frees during the network's inference pass.

However, even with these optimizations, the measured peak memory usage of over 1 GB in Figure 5(b) would not support training runs over 128-image batches. In Figure 5(c), we evaluate the impact of sizing memory appropriately for data at the protocol layer. In particular, the bitwise expansion used in our ReLU comparison protocol remained a major source of memory blowup, as bit values were each stored into a full 32- or 64-bit values. Modifying Piranha protocols to closely match the size of allocated values with their logical sizes, i.e. allocating 8-bit integers for any bitwise vectors, significantly cut the peak memory usage by a factor of 2, to 581 MB, or only 250 MB above the baseline model memory requirements. Compared to the initial naïve implementation, these optimizations resulted in $4\times$ better memory performance.

Finally, Table III shows peak GPU memory usage for Piranha as it performs training passes over each evaluated network architecture. For SecureML in particular, the baseline memory requirement for the network parameters dominates any temporary memory requirements, as the peak memory use only grows by 6 MB between runs over batches of 1 image and 128 images. As expected, P-FantasticFour exhibits larger increases in peak memory use as batch size increases. This is due to the increased number of local shares it must maintain for each secret-shared value, proportionally increasing memory load. Critically, Piranha's evaluation of VGG16 over 128-image batches approaches the limit of what single GPUs can support: P-FantasticFour requires a peak memory load of just under 10 GB, so an increased batch size of 256, for example, would deplete the 16 GB of GPU memory available.

E. Comparison with Prior Work

Finally, we compare the runtime and communication overhead of Piranha relative to a state-of-the-art protocols for neural network training: a CPU-based implementation, Falcon [18], and a GPU-based implementation, CryptGPU [22]. Both protocols are fixed to a 3-party setting, while Piranha is designed to support a general class of LSSS protocols. In this section, we compare the performance of existing protocols with Piranha's equivalent 3-party P-Falcon implementation, to evaluate whether the generality of Piranha's design comes at a performance cost.

We benchmark the run-time for a *single* training and inference pass over 3 different networks – LeNet, AlexNet, and

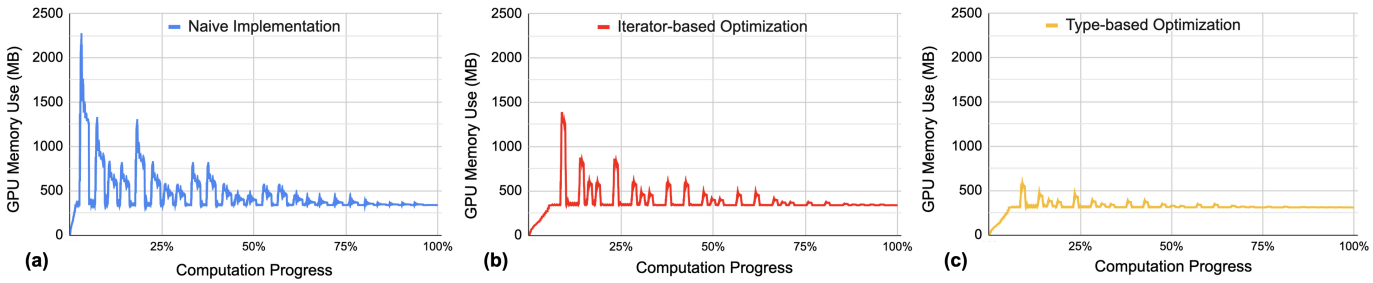

Fig. 5: GPU memory footprint over the course of a VGG16 forward pass. Each point represents a snapshot measurement of total GPU memory allocation (in MB) at each memory operation (allocations or de-allocations). Figure 5(a) shows the footprint of a naive GPU implementation, Figure 5(b) the footprint of the same computation after iterator-based optimizations, and Figure 5(c) the footprint after efficiently sizing bit-containing data structures.

	Protocol	SecureML (MB) (MNIST)			LeNet (MB) (MNIST)			AlexNet (MB) (CIFAR10)			VGG16 (MB) (CIFAR10)		
		k=1	k=64	k=128	k=1	k=64	k=128	k=1	k=64	k=128	k=1	k=64	k=128
Private Training	P-SecureML	319	321	325	437	535	661	507	531	585	629	3017	5505
	P-Falcon	325	327	331	461	577	749	603	649	689	847	3927	7207
	P-FantasticFour	331	335	339	481	651	897	675	743	805	1027	5481	10197

TABLE III: The maximum memory usage of a secure training pass (forward and backward pass) for various MPC protocols and network architectures. Piranha's memory efficient design enables running large networks such as VGG16 with a batch size of 128 where prior works have been able to only run batch sizes of 32 [22].

Computation	Framework	Time (s)			Communication (MB)		
		LeNet (MNIST)	AlexNet (CIFAR10)	VGG16 (CIFAR10)	LeNet (MNIST)	AlexNet (CIFAR10)	VGG16 (CIFAR10)
Private Inference	Falcon	0.038	0.110	1.440	2.29	4.02	40.05
	CryptGPU	0.380	0.910	2.140	3	2.43	56.2
	P-Falcon	**0.031**	**0.131**	**0.469**	**2.492**	**1.960**	**88.39**
Private Training	Falcon	14.9	62.37	360.83	0.346	0.621	1.78
	CryptGPU	2.21	2.910	12.140	1.14	1.37	7.55
	P-Falcon	**0.888**	**1.419**	**7.473**	**0.417**	**0.581**	**4.261**

TABLE IV: We compare the run-times for private training and inference of various network architectures with prior state-of-the-art works over CPU and GPU. Falcon and CryptGPU values are sourced from [22] Table I. Private inference uses batch size of 1, training uses 128 for LeNet, AlexNet and 32 for VGG16. For smaller computations (private inference), Piranha provides comparable performance to CPU-based protocols. However, for larger computations (private training), Piranha shows consistent improvement between $16 - 48\times$, a factor that improves with scale.

VGG16. While we can support batch sizes of up to 128 on each of these networks, we scale down our computation to provide an apples to apples comparison with prior work. The results are presented in Table IV.

For private inference, where the forward passes use a single input image (batch size of 1), the computation is not large enough to fully benefit from GPU acceleration. Table IV shows that Piranha achieves comparable performance to the CPU-based FALCON for private inference over small networks, but over the much larger VGG16 architecture, Piranha already yields a $3\times$ performance improvement.

GPU acceleration has a much stronger impact on private training iterations, where the computation sizes are much larger due to the increased batch size and the addition of a backward pass over the network. Even on the smallest architecture, LeNet, Piranha performs training iterations $16\times$ faster by leveraging a GPU, while on the larger architectures we benchmark, we show between a 44-48$\times$ speedup.

In addition to evaluating the benefits of GPU acceleration, Table IV also quantifies whether Piranha incurs additional overhead from supporting multiple protocol implementations, compared to tools that integrate a specific MPC protocol end-to-end like CryptGPU [17]'s 3-party implementation. Considering private inference, Piranha is significantly faster, showing approximately 12, 7, and $4\times$ speedup on each of LeNet, AlexNet, and VGG16, respectively. We also show a performance advantage in computing training iterations, with performance gains ranging from approximately $2.5\times$ on LeNet to $1.6\times$ on VGG16. We attribute these constant improvements to a few factors. First, Piranha's direct use of 64-bit integer kernels avoids the repeated 16-bit floating point multiplications that CryptGPU incurs. We do this at the cost of using less powerful GPU integer cores and kernel implementations that must be emulated with 32-bit integer instructions. Second, even though Piranha supports many different protocol implementations, Table IV shows that the negligible overhead of our approach can yield the same or better performance than single-protocol designs. Third, some portion of these performance

difference may be attributable to different programming environments – Piranha is implemented in C++ while CryptGPU is implemented over PyTorch.

VII. Discussion, Limitations, and Future Work

Section VI shows that GPUs provide much-needed performance acceleration for secure computation. Piranha's modular platform structure means that functional enhancements made at any layer of the platform – from future performance improvements in the GPU kernels to additional MPC protocols or new privacy-preserving applications – can immediately benefit other system components.

Device layer. The device layer separates protocols from the GPU interface. Thus, acceleration of local operations, optimizations, or entirely different methods of performing integer- and fixed point-based calculations can be independently developed. Even in its current state, Piranha's integer kernels are slower than their floating-point equivalents implemented by popular libraries like cuBLAS [33], as they can take advantage of features like tensor cores that focus exclusively on floating-point. Future efforts can focus on supporting better kernels, enabling multi-GPU usage, and supporting custom accelerators on platforms such as FPGAs [51].

Protocol layer. Piranha can be used for development of newer multi-party protocols, expanding support for different number of parties, innovative protocols, and adversarial models. As noted in Section I, we focus on LSSS protocols in a semi-honest security model, and the protocols we implement operate over 32- and 64-bit integer rings, such that the existing hardware support for modular arithmetic simplifies computational overhead. However, support for other protocol types can be expanded, in supporting field operations, accelerating garbled circuit evaluation [52], or adding homomorphic encryption support [53] to enable dishonest-majority protocols.

Application layer. We showcase the use of Piranha for making meaningful progress on private neural networks training. Piranha's modular approach provides a rich environment for innovation in MPC-friendly neural network design, such as private training of newer architectures like residual networks, transformers, or LSTMs. While we only evaluate Piranha over a neural network training application, the platform allows development of arbitrary, protocol-agnostic secure computation. Future work can focus on demonstrating the ability of the platform to support applications in other areas, such as oblivious sorting or oblivious RAMs.

VIII. Related Work

In recent years, a number of new frameworks have been proposed for privacy-preserving approaches to machine learning. While most frameworks demonstrate a CPU-only implementation, there are a few works that explore GPU assisted computation. The two earliest works by Husted *et. al.* [54] and Frederiksen and Nielsen [55] explore the use of GPUs for improving secure computation using garbled circuits and OT extensions. Delphi [13] uses GPUs to improve the performance of linear components of the computation. In a more recent work, CryptGPU [22] building on top of the CrypTen framework [17] uses GPUs for the entire computation. Recently, GForce [56] shows the benefits of GPU acceleration for secure inference. In a somewhat related effort, cuHE [53] and PixelVault [57] use GPUs for homomorphic encryption, securing keys, and encryption operations. Visor [58] has looked at using GPUs for secure computation over enclaves.

A number of general purpose frameworks have improved the practical performance of MPC. In the dishonest majority setting, a number of works [20, 59–63] improve the performance of the original SPDZ protocols [64, 65]. Helen [66] proposes a system to train a linear model in a dishonest majority setting. Poseidon [67] explores the use of MPC techniques for federated learning in a similar corruption model. A lot more frameworks propose new specialized protocols and implementations in the semi-honest and honest majority adversarial settings. Recent 2-party computation frameworks include [13, 28, 68–73] that typically look at protocols in the semi-honest setting. A number of frameworks explore a 3-party setup with an honest majority corruption. This includes [12, 18, 22–24, 26, 27, 43]. Similarly, 4-party computation frameworks include [25, 26, 43, 74]. Other proposed frameworks include [75, 76]. An entire line of work improves the performance of garbled circuit based approaches to secure computation. Recent advances include as well as silent OT extension protocols such as [10, 77–79].

A number of libraries with varying infrastructures are open sourced. MP-SPDZ and SCALE-MAMBA [14, 16] implement a number of protocols, including most of the dishonest majority protocols. CrypTen [17] implements a few protocols over PyTorch. Other popular libraries providing a number of useful secure computation tools include [15, 80]. There also exist open-source libraries for privacy-preserving machine learning such as Rosetta and PySyft [81, 82]. To the best of our knowledge, there is no open source library that enables general secure computation applications to benefit from the use of GPUs. Piranha can not only fill this gap, but reduce the gap between plaintext and privacy-preserving computations.

IX. Conclusion

In this work, we propose Piranha, a platform for GPU-accelerated MPC protocol development. Piranha contributes three modular components: a device layer that manages protocol memory on the GPU and accelerates MPC-specific integer operations, a protocol layer where memory-efficient in-place operations can be leveraged to fit the constrained GPU environment, and an application layer for privacy-preserving computation on any underlying protocol. The modular structure of Piranha provides wide applicability for other projects to use GPU acceleration without requiring expert knowledge. To demonstrate that Piranha provides significant improvements in run-time through GPU-based acceleration, we implement 3 different MPC protocols for secure training of neural networks on top of Piranha, resulting in a 16-48$\times$ performance improvement. Finally, using Piranha, we are able to securely train, for the first time, a neural network, from scratch, with over 100 million parameters.

REFERENCES

[1] K. W. Bowyer, K. Hollingsworth, and P. J. Flynn, "Image understanding for iris biometrics: A survey," *Computer vision and image understanding*, vol. 110, no. 2, pp. 281–307, 2008.

[2] C. Perlich, B. Dalessandro, T. Raeder, O. Stitelman, and F. Provost, "Machine learning for targeted display advertising: Transfer learning in action," *Machine learning*, vol. 95, no. 1, pp. 103–127, 2014.

[3] J. Perols, "Financial statement fraud detection: An analysis of statistical and machine learning algorithms," *Auditing: A Journal of Practice & Theory*, vol. 30, no. 2, pp. 19–50, 2011.

[4] A. C. Yao, "Protocols for Secure Computations," in *23rd annual symposium on foundations of computer science (sfcs 1982)*. IEEE, 1982, pp. 160–164.

[5] C. Gentry, "A fully homomorphic encryption scheme," Ph.D. dissertation, Stanford University, 2009, crypto.stanford.edu/craig.

[6] C. Dwork, A. Roth *et al.*, "The algorithmic foundations of differential privacy," in *Foundations and Trends in Theoretical Computer Science*, 2014.

[7] A. C. Yao, "How to generate and exchange secrets (extended abstract)," in *IEEE Symposium on Foundations of Computer Science (FOCS)*, 1986.

[8] O. Goldreich, S. Micali, and A. Wigderson, "How to play any mental game or a completeness theorem for protocols with honest majority," in *ACM Symposium on Theory of Computing (STOC)*, 1987.

[9] M. Ben-Or, S. Goldwasser, and A. Wigderson, "Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract)," in *ACM Symposium on Theory of Computing (STOC)*, 1988.

[10] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, and P. Scholl, "Efficient pseudorandom correlation generators: Silent ot extension and more," in *Advances in Cryptology—CRYPTO*, 2019, pp. 489–518.

[11] E. Boyle, N. Gilboa, and Y. Ishai, "Function secret sharing," in *Advances in Cryptology—EUROCRYPT*, 2015.

[12] P. Mohassel and P. Rindal, "ABY^3: A mixed protocol framework for machine learning," in *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[13] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. A. Popa, "Delphi: A cryptographic inference service for neural networks," in *USENIX Security Symposium (USENIX)*, 2020.

[14] M. Keller, "MP-SPDZ: A Versatile Framework for Multi-Party Computation," in *ACM Conference on Computer and Communications Security (CCS)*, 2020.

[15] P. Rindal, "libOTe: an efficient, portable, and easy to use Oblivious Transfer Library," https://github.com/osu-crypto/libOTe.

[16] A. Aly, M. Keller, E. Orsini, D. Rotaru, P. Scholl, N. P. Smart, and T. Wood, "SCALE-MAMBA v1.2: Documentation," https://homes.esat.kuleuven.be/~nsmart/SCALE/Documentation.pdf, 2018.

[17] "Crypten: Privacy-preserving machine learning built on pytorch," https://github.com/facebookresearch/CrypTen, 2019.

[18] S. Wagh, S. Tople, F. Benhamouda, E. Kushilevitz, P. Mittal, and T. Rabin, "FALCON: Honest-Majority Maliciously Secure Framework for Private Deep Learning," in *Privacy Enhancing Technologies Symposium (PETS)*, 2021.

[19] S. Shi, Q. Wang, P. Xu, and X. Chu, "Benchmarking state-of-the-art deep learning software tools," in *2016 7th International Conference on Cloud Computing and Big Data (CCBD)*. IEEE, 2016, pp. 99–104.

[20] D. Escudero, S. Ghosh, M. Keller, R. Rachuri, and P. Scholl, "Improved Primitives for MPC over Mixed Arithmetic-Binary Circuits," in *Advances in Cryptology—CRYPTO*, 2020.

[21] E. Makri, D. Rotaru, F. Vercauteren, and S. Wagh, "Rabbit: Efficient Comparison for Secure Multi-Party Computation," in *Financial Cryptography and Data Security (FC)*, 2021.

[22] S. Tan, B. Knott, Y. Tian, and D. J. Wu, "Cryptgpu: Fast privacy-preserving machine learning on the GPU," in *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[23] H. Chaudhari, A. Choudhury, A. Patra, and A. Suresh, "Astra: High throughput 3pc over rings with application to secure prediction," in *ACM SIGSAC Conference on Cloud Computing Security Workshop*, 2019.

[24] A. Patra and A. Suresh, "Blaze: Blazing fast privacy-preserving machine learning," in *Symposium on Network and Distributed System Security (NDSS)*, 2020.

[25] R. Rachuri and A. Suresh, "Trident: Efficient 4pc framework for privacy preserving machine learning," in *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[26] A. Dalskov, D. Escudero, and M. Keller, "Fantastic four: Honest-majority four-party secure computation with malicious security," in *USENIX Security Symposium (USENIX)*, 2021.

[27] S. Wagh, D. Gupta, and N. Chandran, "SecureNN: 3-Party Secure Computation for Neural Network Training," in *Privacy Enhancing Technologies Symposium (PETS)*, 2019.

[28] P. Mohassel and Y. Zhang, "SecureML: A System for Scalable Privacy-Preserving Machine Learning," in *IEEE Symposium on Security and Privacy (S&P)*, 2017.

[29] Data61, "MP-SPDZ: Versatile Framework for Multi-party Computation," https://github.com/data61/MP-SPDZ, 2019.

[30] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry *et al.*, "An updated set of basic linear algebra subprograms (blas)," *ACM Transactions on Mathematical Software*, 2002.

[31] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," https://arxiv.org/abs/1409.1556, 2014.

[32] A. Krizhevsky, V. Nair, and G. Hinton, "The CIFAR-10 dataset," http://www.cs.toronto.edu/kriz/cifar.html, 2014.

[33] "cublas," https://developer.nvidia.com/cublas, accessed: 2021-08-01.

[34] "Cutlass: Cuda templates for linear algebra subroutines," accessed: 2021-08-01.

[35] I. Damgård, D. Escudero, T. Frederiksen, M. Keller, P. Scholl, and N. Volgushev, "New primitives for actively-secure mpc over rings with applications to private machine learning," in *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[36] "Secure supply chain management," https://faui1-files.cs.fau.de/filepool/publications/octavian_securescm/SecureSCM-D.9.2.pdf, 2009.

[37] O. Catrina and S. De Hoogh, "Improved primitives for secure multiparty integer computation," in *Security and Cryptography for Networks*, 2010.

[38] I. Damgård and J. B. Nielsen, "Universally composable efficient multiparty computation from threshold homomorphic encryption," in *Advances in Cryptology—CRYPTO*. Springer, 2003, pp. 247–264.

[39] P. Laud, A. Pankova, M. Pettai, and J. Randmets, "Specifying sharemind's arithmetic black box," in *ACM workshop on Language support for privacy-enhancing technologies*, 2013.

[40] T. Ryffel, P. Tholoniat, D. Pointcheval, and F. Bach, "Ariann: Low-interaction privacy-preserving deep learning via function secret sharing," in *Privacy Enhancing Technologies Symposium (PETS)*, 2022.

[41] T. Araki, J. Furukawa, Y. Lindell, A. Nof, and K. Ohara, "High-throughput semi-honest secure three-party computation with an honest majority," in *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[42] J. Furukawa, Y. Lindell, A. Nof, and O. Weinstein, "High-throughput secure three-party computation for malicious adversaries and an honest majority," in *Advances in Cryptology—EUROCRYPT*, 2017.

[43] N. Koti, M. Pancholi, A. Patra, and A. Suresh, "Swift: Super-fast and robust privacy-preserving machine learning," in *USENIX Security Symposium (USENIX)*, 2021.

[44] D. Escudero, A. Dalskov, and M. Keller, "Secure evaluation of quantized neural networks," in *Privacy Enhancing Technologies Symposium (PETS)*, 2020.

[45] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *IEEE international conference on computer vision*, 2015.

[46] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2012.

[47] M. Keller and K. Sun, "Effectiveness of mpc-friendly softmax replacement," 2020, https://arxiv.org/pdf/2011.11202.pdf.

[48] Nvidia, "Thrust, the cuda c++ template library," https://docs.nvidia.com/cuda/thrust/index.html.

[49] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[50] "MNIST database," http://yann.lecun.com/exdb/mnist/, accessed: 2017-09-24.

[51] Y. Dou, S. Vassiliadis, G. K. Kuzmanov, and G. N. Gaydadjiev, "64-bit floating-point fpga matrix multiplication," in *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, 2005, pp. 86–95.

[52] A. C. Yao, "Protocols for secure computations," in *IEEE Symposium on Foundations of Computer Science (FOCS)*, 1982.

[53] W. Dai and B. Sunar, "cuhe: A homomorphic encryption accelerator library," in *International Conference on Cryptography and Information Security in the Balkans*, 2015.

[54] N. Husted, S. Myers, A. Shelat, and P. Grubbs, "Gpu and cpu parallelization of honest-but-curious secure two-party computation," in *Annual Computer Security Applications Conference (ACSAC)*, 2013.

[55] T. K. Frederiksen, T. P. Jakobsen, and J. B. Nielsen, "Faster maliciously secure two-party computation using the gpu," in *Conference on Security and Cryptography for Networks*, 2014.

[56] L. K. Ng and S. S. Chow, "Gforce: Gpu-friendly oblivious and rapid neural network inference," in *USENIX Security Symposium (USENIX)*, 2021.

[57] G. Vasiliadis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis, "Pixelvault: Using gpus for securing cryptographic operations," in *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[58] R. Poddar, G. Ananthanarayanan, S. Setty, S. Volos, and R. A. Popa, "Visor: Privacy-preserving video analytics as a cloud service," in *USENIX Security Symposium (USENIX)*, 2020.

[59] M. Keller, V. Pastro, and D. Rotaru, "Overdrive: making SPDZ great again," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2018, pp. 158–189.

[60] H. Chen, M. Kim, I. Razenshteyn, D. Rotaru, Y. Song, and S. Wagh, "Maliciously Secure Matrix Multiplication with Applications to Private Deep Learning," in *Advances in Cryptology—ASIACRYPT*, 2020.

[61] A. Aly, E. Orsini, D. Rotaru, N. P. Smart, and T. Wood, "Zaphod: Efficiently combining lsss and garbled circuits in scale," in *ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, 2019.

[62] D. Rotaru and T. Wood, "Marbled Circuits: Mixing Arithmetic and Boolean Circuits with Active Security," in *International Conference on Cryptology in India*. Springer, 2019, pp. 227–249.

[63] M. Keller, E. Orsini, and P. Scholl, "MASCOT: Faster malicious arithmetic secure computation with oblivious transfer," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 830–842.

[64] I. Damgård, V. Pastro, N. Smart, and S. Zakarias, "Multiparty Computation from Somewhat Homomorphic Encryption," in *Annual Cryptology Conference*. Springer, 2012, pp. 643–662.

[65] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart, "Practical Covertly Secure MPC for Dishonest Majority–or: Breaking the SPDZ Limits," in *European Symposium on Research in Computer Security*. Springer, 2013, pp. 1–18.

[66] W. Zheng, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Helen: Maliciously secure coopetitive learning for linear models," in *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[67] S. Sav, A. Pyrgelis, J. R. Troncoso-Pastoriza, D. Froelicher, J.-P. Bossuat, J. S. Sousa, and J.-P. Hubaux, "Poseidon: Privacy-preserving federated neural network learning," in *Symposium on Network and Distributed System Security (NDSS)*, 2021.

[68] D. Rathee, M. Rathee, N. Kumar, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma, "Cryptflow2: Practical 2-party secure inference," in *ACM Conference on Computer and Communications Security (CCS)*, 2020.

[69] R. Lehmkuhl, P. Mishra, A. Srinivasan, and R. A. Popa, "Muse: Secure inference resilient to malicious clients," in *USENIX Security Symposium (USENIX)*, 2021.

[70] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan, "GAZELLE: A Low Latency Framework for Secure Neural Network Inference," in *USENIX Security Symposium (USENIX)*, 2018, pp. 1651–1669.

[71] M. S. Riazi, C. Weinert, O. Tkachenko, E. M. Songhori, T. Schneider, and F. Koushanfar, "Chameleon: A hybrid secure computation framework for machine learning applications," in *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2018.

[72] A. Patra, T. Schneider, A. Suresh, and H. Yalame, "ABY2.0: Improved Mixed-Protocol Secure Two-Party Computation," in *USENIX Security Symposium (USENIX)*, 2021.

[73] R. Kanagavelu, Z. Li, J. Samsudin, Y. Yang, F. Yang, R. S. M. Goh, M. Cheah, P. Wiwatphonthana, K. Akkarajitsakul, and S. Wang, "Two-phase multi-party computation enabled privacy-preserving federated learning," in *IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2020.

[74] M. Byali, H. Chaudhari, A. Patra, and A. Suresh, "FLASH: Fast and robust framework for privacy-preserving machine learning," in *Privacy Enhancing Technologies Symposium (PETS)*, 2020.

Kernel		Time (ms)			
Library	Datatype	784x9x20	1024x27x64	784x147x64	10000x1000x10000
cuBLAS	float-32	0.014	4.16	4.45	54.19
Piranha	float-32	0.981	4.51	4.56	65.16
Piranha	int-32	3.61	4.38	4.52	78.35
cuBLAS	float-64	4.58	6.37	4.70	126.5
Piranha	float-64	4.60	5.92	4.69	114.95
Piranha	int-64	4.76	4.66	4.90	2482.17

TABLE V: Runtimes for the matrix multiplication kernels used in Piranha vs. the cuBLAS implementation on various sizes of matrices.

[75] W. Zheng, R. Deng, W. Chen, R. A. Popa, A. Panda, and I. Stoica, "Cerebro: A platform for multi-party cryptographic collaborative learning," in *USENIX Security Symposium (USENIX)*, 2021.

[76] R. Poddar, S. Kalra, A. Yanai, R. Deng, R. A. Popa, and J. M. Hellerstein, "Senate: A maliciously-secure mpc platform for collaborative analytics," in *USENIX Security Symposium (USENIX)*, 2021.

[77] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, P. Rindal, and P. Scholl, "Efficient two-round ot extension and silent non-interactive secure computation," in *ACM Conference on Computer and Communications Security (CCS)*, 2019, pp. 291–308.

[78] P. Schoppmann, A. Gascón, L. Reichert, and M. Raykova, "Distributed vector-ole: improved constructions and implementation," in *ACM Conference on Computer and Communications Security (CCS)*, 2019, pp. 1055–1072.

[79] K. Yang, C. Weng, X. Lan, J. Zhang, and X. Wang, "Ferret: Fast Extension for coRRElated oT with small communication," in *ACM Conference on Computer and Communications Security (CCS)*, 2020, pp. 1607–1626.

[80] D. Demmler, T. Schneider, and M. Zohner, "ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation," in *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[81] "A Privacy-Preserving Framework Based on TensorFlow," https://github.com/LatticeX-Foundation/Rosetta/.

[82] "A Python library for secure and private Deep Learning." https://github.com/OpenMined/PySyft.

APPENDIX A
COMPARISON WITH FLOATING POINT KERNELS

We mention in Section III the tradeoff in performance when computing directly over integer buffers on the GPU, as opposed to decomposing large bit-width values into smaller chunks for use in floating point-based kernels. In Table V, we compare the 32- and 64-bit kernels that Piranha uses, implemented with CUTLASS [34], against state-of-the-art 32- and 64-bit floating point kernels from cuBLAS [33]. While we can directly compare floating point performance between the systems, cuBLAS does not support large integer matrix multiplication, so we only present Piranha-based results for comparison.

We benchmark the runtimes for the matrix multiplication kernels used in Piranha vs. the cuBLAS implementation on various sizes of matrices in Table V. We observer that Piranha kernels, when executed with floating point datatypes result in comparable overhead to cuBLAS implementations. However, executing 32-bit integer multiplications is much more expensive in Piranha compared to the floating point case. 64-bit integer multiplications are relatively comparable to cuBLAS 64-bit floating point, but at very large matrix sizes, there is a significant difference between the two. This is likely due to the fact that 64-bit integer operations are emulated using 32-bit integer instructions that target the GPU integer cores used.