Piranha: A GPU Platform for Secure Computation

Jean-Luc Watson, Sameer Wagh, and Raluca Ada Popa
University of California, Berkeley

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 to complete. Given that large ML inference and training tasks in the plaintext setting are significantly accelerated by Graphical Processing Units (GPUs), this raises the natural question: can secure MPC leverage GPU acceleration? A few recent works have studied this question in the context of accelerating specific components or protocols, but do not provide a general-purpose solution. Consequently, MPC developers must be both experts in cryptographic protocol design and proficient at low-level GPU kernel development to achieve good performance on any new protocol implementation.

We present Piranha, a general-purpose, modular platform for accelerating secret sharing-based MPC protocols using GPUs. Piranha allows the MPC community to easily leverage the benefits of a GPU without requiring GPU expertise. Piranha contributes a three-layer architecture: (1) a *device layer* that can independently accelerate secret-sharing protocols by providing integer-based kernels absent in current general-purpose GPU libraries, (2) a modular *protocol layer* that allows developers to maximize utility of limited GPU memory with in-place computation and iterator-based support for non-standard memory access patterns, and (3) an *application layer* that allows applications to remain completely agnostic to the underlying protocols they use.

To demonstrate the benefits of Piranha, we implement 3 state-of-the-art linear secret sharing MPC protocols for secure NN training: 2-party SecureML (IEEE S&P '17), 3-party Falcon (PETS '21), and 4-party FantasticFour (USENIX Security '21). Compared to their CPU-based implementations, the same protocols implemented on top of Piranha's protocol-agnostic acceleration exhibit a $16-48\times$ decrease in training time. For the first time, Piranha demonstrates the feasibility of training a *realistic* neural network (e.g. VGG), end-to-end, using MPC *in a little over one day*. Piranha is open source and available at https://github.com/ucbrise/piranha.

1 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 [13,65,66]. 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 [32,37,91].

Secure Multi-Party Computation (SMC/MPC) has emerged as a promising tool for privacy-preserving computation [12,39,91]. 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 [15,16,57,59] as well as robust, efficient, and versatile implementations [7,21,45,72]. 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 [88].

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 $10-30\times$ [77], making them an essential tool in today's ML infrastructure. A few recent works [35, 42, 57, 79] have utilized GPUs to accelerate MPC computation. However, their GPU usage is limited to accelerating individual operations or a specific MPC protocol. Delphi [57], for example, only accelerates convolution operations before continuing computation on the CPU, while CryptGPU [79] designs a specific 3-party protocol for its application. As a result, any new protocol must re-implement the same basic GPU support, a difficult task in general. Requiring MPC developers to develop domain-specific knowledge of GPU task scheduling and memory hierarchy to implement efficient kernels raises the barrier to entry and impedes the development of practical MPC-based systems.

1.1 Challenges and Insights

Supporting efficient secure computation on the GPU faces a few core problems. Plain-text ML computation is straightforward and can be done directly in floating point with reasonable memory constraints, while the equivalent multi-party computation can be accomplished using any number of different protocols, operating over integer types, with significantly higher available memory requirements. We

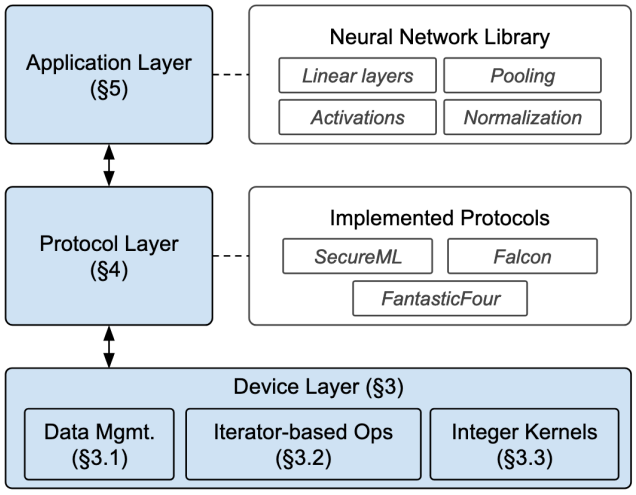


Figure 1: Piranha's three-layer architecture in blue, with components implemented on top in white. On the device layer, we contribute low-level GPU kernels accelerating local, integer-based data shares. At the protocol layer, we implement functionality for three different linear secret-sharing (LSSS) MPC protocols at the protocol layer: SecureML [60] (2-party), Falcon [88] (3-party), and FantasticFour [23] (4-party). At Piranha's application layer, we provide a protocol-agnostic neural network library that can be executed by any of the protocols. Piranha is modular in that it can support additional components beyond what we provide.

design Piranha to address these challenges while providing a general-purpose platform for MPC development, with support for linear secret-sharing schemes (LSSS), encompassing a large (and growing) number of state-of-the-art protocols for secure computation [18, 23, 56, 64, 69, 86, 88].

Challenge: Protocol-independent acceleration. As even simple multi-party operations such as multiplications may be computed using a wide variety of approaches based on the protocol used, how can a platform efficiently provide acceleration support to each of them? While entire MPC computations are quite different, they are almost always decomposed into individual operations over local data shares mixed with communication between the parties to obtain the final result. Thus, accelerating local operations over local shares can yield significant performance benefits while remaining entirely protocol-independent. Piranha uses vector shares as the basic unit of computation over individual values, as it ensures that any protocol or application implemented using them will inherently take advantage of the GPU's parallelism. With a shared abstraction for local data, Piranha can transparently manage data transmission and memory allocation, keeping data on the GPU for the entirety of the computation while minimizing data transfer from the CPU.

Challenge: Enabling integer-based GPU computation. Data representation is an important consideration for secure computation libraries. There is a tension between supporting high-precision real values required by applications such as NN training (e.g. float datatypes) and structured algebraic properties required by the secret sharing schemes (e.g. int datatypes). State-of-the-art secure computation libraries and frameworks resolve this tension by using fixed-point datatypes, encoding real values with a fixed precision into a large integral datatype (typically 64-bits). Unfortunately, GPUs primarily focus on accelerating floating-point computation with extremely efficient kernel implementations, targeting plaintext graphics and ML workloads. This has resulted in a dearth of GPU kernels for large bitsize (32- and 64-bit) integer computations [3]. We argue that the lack of integer kernels in existing GPU libraries significantly hampers simple acceleration for MPC protocols; Piranha explicitly provides for integer-based shares and matching GPU integer kernels to accelerate common operations.

Challenge: Supporting large MPC problems in limited GPU memory. 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 primarily support 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, MPC protocols may exhibit non-standard memory access patterns incompatible with the integer kernels available. Naively copying data into the desired layout before performing the computation unnecessarily limits the problem sizes we can support, so to efficiently parallelize some operations, Piranha's insight is that memory-efficient computation can be achieved with views over GPU memory, allowing for in-place computation. 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.

1.2 Evaluation Summary

Piranha addresses these issues with a modular, three-layer GPU-based framework for secure computation (Figure 1) whose structure we discuss in Section 2. We demonstrate the practical use of Piranha in implementing three different LSSS protocols for secure neural network training – the 2-party SecureML [60], 3-party Falcon [88], and 4-party Fantastic

Four [23] protocols. Piranha does not propose a new secure multi-party protocol, rather, we focus on demonstrating how the platform accelerates existing 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 [28] of computational building blocks such as matrix-multiplication, convolutions, and comparisons, Piranha improves runtime by 2 to 3 orders of magnitude. Thus, Piranha makes a big step forward towards practical MPC training. For example, prior work such as Falcon estimates that training a realistic neural network like VGG16 using its 3-party protocol would require 14 days [88]. In comparison, Piranha can perform the same training process in 33 hours, a 10× improvement.

One would expect that since Piranha accelerates general LSSS-based MPC, Piranha would thus be slower than a system like CryptGPU [79] that is tailored for a *specific* MPC protocol. We show that in fact, we achieve generality while demonstrating a 2-12× improvement in runtime and supporting up to a 4× greater problem size on the same GPU hardware. CryptGPU [79] only demonstrates full end-to-end training on simple networks such as AlexNet [51], while only micro-benchmarking single-layer training passes for larger networks like VGG16 [78] which has twice as many parameters. In contrast, for the first time, Piranha demonstrates the feasibility of training a *realistic* neural network like VGG [78], *end-to-end*, using MPC *in a little over one day*.

2 System Architecture

Piranha contributes three distinct, modular layers that provide a separation of concerns for GPU-accelerated secure computation (Figure 1): a *device* layer that abstracts GPU-specific code from MPC developers; 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 for high-level computation.

The *device layer* consists of two components. First, it provides an abstraction of a GPU-based integer vector, which represents a locally-held share of a vector whose values are secret-shared among multiple parties. These shares live on the GPU throughout the computation, minimizing data transfer overhead. Communication is handled in a protocol-independent manner: when necessary, the device layer copies a share to the CPU before transmitting it over the network. Second, the device layer maintains a set integer kernels that implement commonly-needed functionality (e.g. elementwise addition or matrix multiplication) over local share vectors. We discuss how MPC operations are accelerated in Section 3.

The *protocol layer* allows MPC developers to compose operations on local shares into a full protocol, benefiting from GPU acceleration without developing expert knowledge or re-implementing GPU support from scratch. Applications rely on each protocol to provide an interface in the form of a secret-shared vector and a set of functionalities that can operate on them. Alongside individual protocol definitions, we implement protocol functionality under the Arithmetic Black Box Model that can be used to supplement any of the specific protocols, demonstrating the benefit of Piranha's modular structure. In addition, MPC protocols can require intricate computation that cannot be foreseen at the device layer; Section 4 details how iterator-based views over local shares on the GPU can be used to enable these operations while remaining within the GPU's limited memory constraints.

Finally, at the *application layer*, computation can focus on solving domain-specific challenges such as secure neural network training, without a dependency on any specific protocol. The functionality set provided by each protocol determines which applications can use a given protocol without requiring modification.

To put Piranha in context, imagine implementing a simple privacy-preserving neural network layer. Its core logic (e.g. updating layer parameters during forward and backward passes) remains untouched at the application layer. Instead of using plaintext vectors, however, the layer makes use of a vector secret-shared by an implementation at the protocol layer, and operates on these secret shares using the corresponding protocol functionality, for example, a privacy-preserving matrix multiplication. In turn, the protocol decomposes its multiplication into a series of local matrix multiplications, which are accelerated by a protocol-independent integer kernel in Piranha's device layer.

Threat model. Piranha assumes that parties participating in a protocol execution operate in separate trust domains, using their dedicated GPUs (e.g. in a cloud provider of choice). A GPU in Piranha communicates with another parties' GPU through their associated CPUs and across a normal Internet connection. As such, Piranha can be used in both LAN and WAN environments. Due to Piranha acting as a platform for existing MPC protocols, parties executing an application with Piranha inherit the security guarantees of the underlying MPC protocol. For example, a protocol with semi-honest security retains those guarantees while being executed by Piranha. We implement and evaluate three such semi-honest protocols on top of Piranha in Section 6.

3 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 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.

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 );
```

3.1 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 4 for an example). An individual GEMM call is shown in Listing 1, line 11. In Section 6, we evaluate how these kernels improve the performance of secure matrix multiplication by up to 200× 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.

3.2 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 4 for details), i.e., operations over vec[2i],vec[2i+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*.

3.3 Integer kernels

The MPC protocols we implement in Section 4.4 operate on additive secret sharing over 32- or 64-bit ranges. As discussed in Section 1, there is a lack of kernel implementations for these data types [3], 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 [4] 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 [3], 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 6 and Appendix B.

4 Protocol layer

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

4.1 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 a MPC developer seeks to implement a 3-party protocol using replicated secret sharing for an honest majority of semi-honest corruptions (for instance [9, 88]). 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.

To multiply two secret matrices x, y, if the first party holds shares (x_0, x_1), and (y_0, y_1), the output can be computed by regrouping the terms of the product as [9]:

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

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 6, we directly evaluate the performance benefit of this implementation against a similar CPU-based protocol for secure matrix multiplication.

Randomness generation. We assume that the parties maintain secure point-to-point communication channels and share pairwise AES keys to generate common randomness. Recent works have looked at efficiently generating randomness on the GPUs [10, 54, 89]. While CPU cores outperform GPU cores for smaller amount of randomness, it becomes desirable to generate randomness using GPUs for large scale random number requirements [10]. Such random number generation can be easily added to the protocol layer in Piranha.

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.

```
1  // Replicated secret sharing class
2  class RSS<T> {
3    DeviceData<T> shareA, shareB;
4  }
5
6  void RSS<T>::matmul(RSS<T> a, RSS<T> b,
7              RSS<T> c, ...) {
8    DeviceData<T> localC;
9
10   localC += gpu::gemm(a.shareA, b.shareA, ...);
11   localC += gpu::gemm(a.shareA, b.shareB, ...);
12   localC += gpu::gemm(a.shareB, b.shareA, ...);
13   // Reshare and truncate localC to c
14 }
```

4.2 Memory-efficient protocols

Section 3 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 $(b_{k-1},\ldots,b_0)$ 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 [24, 34, 56, 59]. In the case of the neural network library we discuss in Section 5, 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 [5]. As part of the computation, at round $i \in \{1,2,\cdots,\log_2 k\}$, the CarryOut computes the AND between adjacent propagating bits, i.e., $p'_j = p_{2j} \wedge p_{2j+1}$ 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 2a 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 must be copied at each step to a different memory allocation before the next round can be evaluated (red-outlined in Figure 2a). During one execution of this particular CarryOut implementation, $\log(n)$ additional data copies are performed.

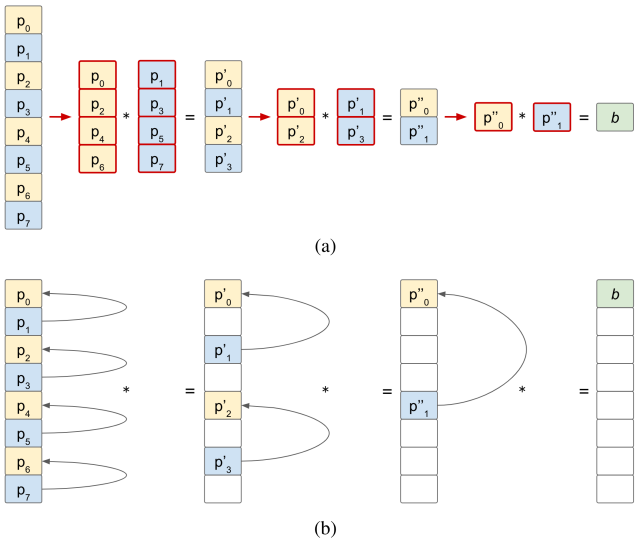


Figure 2: Comparison of a memory-inefficient naive carryout implementation Figure 2a and our iterator-based in-place computation Figure 2b. In the former approach, new memory allocations and data copy – highlighted in red – are done to split pairwise elements into contiguous vectors for parallel GPU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

In contrast, Piranha uses iterator-based views to allow access to non-contiguous data elements in strides. Figure 2b 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. The iterators are input to a pairwise comparison kernel that would otherwise expect data marshalled into a specific contiguous layout, allowing for efficient computation entirely without data movement.

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. Finally, 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 minimizing the memory footprint they require on the GPU.

4.3 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 $\mathcal{F}_{\mathsf{ABB}}$ [26]. 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 [34, 52, 56]. 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.* [56] and a protocol for approximate square-root and inverse computation based on [88].

Secure comparison. We use secure comparison as an example of implementing a method in the arithmetic black box model. The comparison protocol uses edaBits [34] as preprocessing material to efficiently compute a comparison of secret values. An edaBit is a secret sharing of a random value and the bit decomposition of the same value as boolean shares i.e.,

$$\mathsf{edaBit}: [r]_M, [r_0]_2, [r_1]_2, \cdots, [r_m]_2 \text{ where } r \xleftarrow{\$} \mathbb{Z}_M \qquad (2)$$

where $m+1 = \log_2 M$. The protocol for generating this can be found in [34]. The problem of secure comparison over arithmetic secret-sharing can then 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 [5]. Details of this operation are presented in Section 4.2.

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 [74, 88] 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:

$$\begin{aligned} \mathsf{sqrt}(x) &= 0.424 + 0.584(x) \\ 1/x &= 4.245 - 5.857(x) + 2.630(x^2) \end{aligned} \qquad (3)$$

These approximations achieve an L1 error of 0.00676 and 0.02029, respectively, for x between 0.5 and 1.

4.4 MPC protocols

We implement three different MPC protocols to demonstrate Piranha's generality at the protocol layer: a 2-party implementation based on SecureML [60], a 3-party implementation built upon Falcon [88], and a 4-party protocol [23]. We briefly describe each of these protocols below, and prefix them with "P-" to indicate they are implementations accelerated by Piranha.

Two-party protocol (P-SecureML). In 2017, Mohassel and Zhang [60] 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 [34, 56].

Three-party protocol (P-Falcon). We build a 3-party protocol using the work of Wagh *et. al.* [88]. 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 [9, 36, 59]. The non-linear layers are computed using a specialized comparison protocol building upon [86]. Once again, we replace the comparison protocol using the more efficient work by Makri *et. al.* [56].

Four-party protocol (P-FantasticFour). Our 4-party implementation follows the work of Dalskov *et. al.* [23]. 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 [49]). For comparison (probabilistic truncation), the protocol uses a combination of [33] and [49].

5 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 [6, 11]:

(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 [40]. 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 [60] to more complex convolutional neural networks such as AlexNet [51] and

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 1: Functionalities required by the NN training application, implemented by each class in Piranha's protocol layer.

VGG16 [78]. In our evaluation in Section 6, we compare Piranha to the networks used in prior works [79, 88].

5.1 Interfacing the neural network library

As discussed in Section 5, 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 1.

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 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.

5.2 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.

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;
}
```

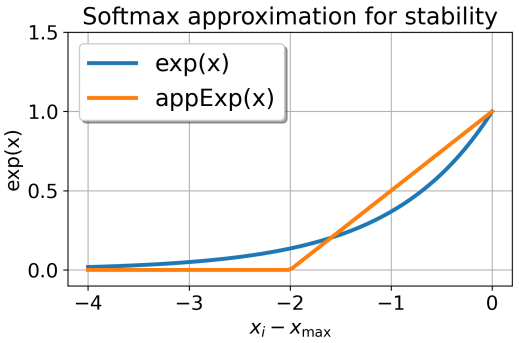


Figure 3: Our new approximate computation of last layer gradients that stabilize the learning process.

Softmax [31] 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 6 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, with deeper network depths requiring higher precision (Section 6.4). 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 [41], 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:

$$\ell = -\sum_i y_i \log p_i \quad \text{where} \quad p_i = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{4}$$

The gradient is then given by:

$$\nabla_i = \frac{\partial \ell}{\partial x_i} = p_i - y_i \tag{5}$$

While there are a few different ways to compute this gradient [48], 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,

$$p_i = \frac{e^{x_i - x_{\max}}}{\sum_{j=1}^{k} e^{x_j - x_{\max}}} \tag{6}$$

We propose a new function computation to approximate the above computation (Eq. 6):

$$p_i \approx \mathsf{appExp}(x_i - x_{\max}) / \sum_{j=1}^{k} \mathsf{appExp}(x_j - x_{\max}) \tag{7}$$

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.

6 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 6.2)

(2) *Can* Piranha *be used to successfully and securely train large neural networks (e.g. over 100 million parameters) in a reasonable amount of time? (Sections 6.3 and 6.4)*

(3) *What are* Piranha*'s computation and communication costs in LAN and WAN environments?* (Section 6.5)

(4) *How well does* Piranha *manage constrained GPU memory and how well does its memory-conscious design improve scalability at the application layer?* (Section 6.6)

(5) *How does the runtime performance of privacy-preserving inference and training, supported by* Piranha*'s protocol-agnostic acceleration, compare with prior work on targeted protocols?* (Section 6.7)

6.1 Evaluation set-up

We run our experiments over similar hardware and networking environments as prior works [23, 60, 88]. For CPU-based implementations, we use Azure F32s_v2 instances with Intel Xeon Platinum 8272CL @ 3.4GHz processors 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 [4], 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 [62].

Baseline. As a baseline, we compare against protocol implementations from MP-SPDZ [28, 45] 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 6.2 with the 2-party semi2k, 3-party replicated-ring, and 4-party rep4-ring protocols. Each of these implementations operate on a single CPU core. We focus evaluating Piranha's performance in the data-dependent "online" phase, as offline generation of data-independent components such as Beaver triples [60] or edaBits [34] can be easily parallelized independently from a particular computation.

Models and Datasets. We evaluate our high-level neural network library with four neural network architectures: SecureML [60], a simple 3-layer network, and LeNet [53], a 5-layer convolutional network, over MNIST [58], and AlexNet [51], an 8-layer convolutional network, and VGG16 [78], a 16-layer convolutional network, over the CIFAR10 dataset [50]. While Piranha fully supports the use of maxpool layers in these architectures, as in CryptGPU [79], we substitute them with averagepool layers to maintain comparative accuracy. Notably, averaging operations are significantly less expensive than max operations in each Piranha-accelerated protocol, as summation requires only a locally-computed linear combination of secret shares while oblivious comparison incurs a logarithmic number of communication rounds among the parties.

6.2 Comparison vs. CPU Implementations

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 4.4, we execute individual operations commonly used by a secure neural network application – matrix multiplications, convolutions, and ReLU comparisons – and compare against the same

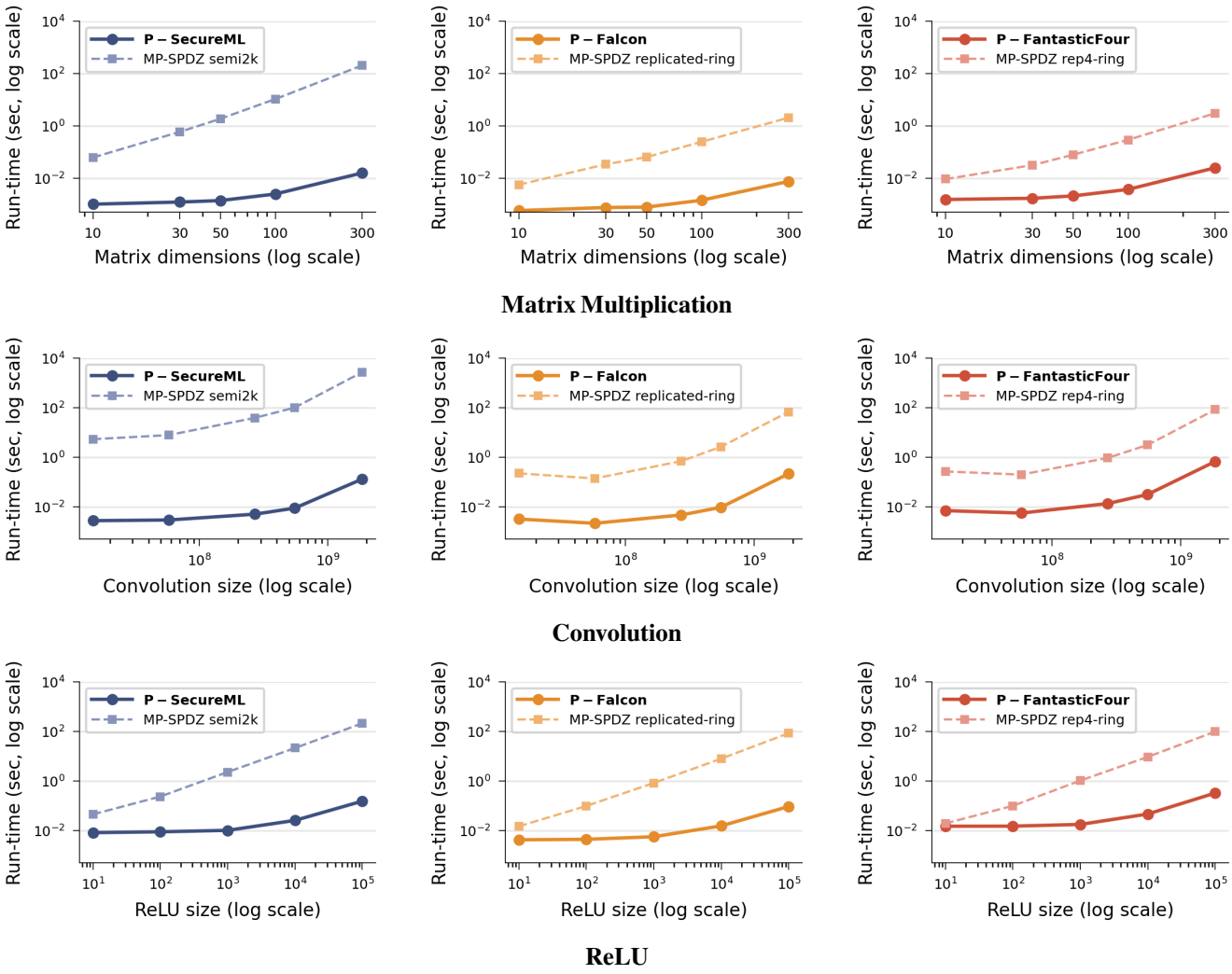


Figure 4: The figures benchmark secure protocols for matrix multiplication, convolutions, and ReLU across 2-, 3-, and 4-party protocols for various sizes of these computations. Piranha consistently improves the run-time of these computations, with improvements as large as 2-4 orders of magnitude for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

operations computed using MP-SPDZ [28] with protocols in the same setting, as described in Section 6.1. 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 $[iw, c_{\text{in}}, c_{\text{out}}, f]$ 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

Network (Dataset)	Protocol	Time (min)	Comm. (GB)	Accuracy	
				Train (%)	Test (%)
SecureML (MNIST)	P-SecureML	12.99	49.55	97.37	96.56
	P-Falcon	7.51	22.84	97.37	96.56
	P-FantasticFour	23.39	33.01	97.37	96.56
LeNet (MNIST)	P-SecureML	87.55	683.18	96.78	96.80
	P-Falcon	71.56	485.90	96.88	97.10
	P-FantasticFour	219.20	676.13	96.88	97.11
AlexNet (CIFAR10)	P-SecureML	156.01	740.50	40.74	40.47
	P-Falcon	110.66	382.18	40.59	40.71
	P-FantasticFour	296.57	533.74	40.97	40.14
VGG16 (CIFAR10)	P-SecureML	3822.84	35454.91	55.02	54.35
	P-Falcon	1979.92	17235.35	55.13	54.26
	P-FantasticFour	7697.54	29106.24	55.02	54.35

Table 2: 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 end-to-end secure training of VGG16, a network with over 100 million parameters.

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.

6.3 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 [88] 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 2.

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 5; over every network architecture we evaluate, our approximation remains stable and allows the networks to train successfully. To ensure that even small gradients can backpropogate through each networks and train a useful model, we vary the level of fixed-point precision: 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. We further discuss how fixed-point precision impacts model accuracy in Section 6.4.

On a small dataset like MNIST, Table 2 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, compared to approximately 97% and 98% accuracy, respectively, when trained in plaintext. For larger networks, the cost of privacy-preserving matrix multiplication dominates the overall runtime [88]. This explains why P-FantasticFour generally takes 2 to 3× longer for the same training pass, because the 4-party protocol requires 7 local matrix multiplication operations for every privacy-preserving matrix multiplication, compared to only 3 local multiplications for the 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. We observed a 59% accuracy when training the same model in plaintext (note that an untrained network/random guessing achieves a 10% accuracy given that there are 10 classes). When considering VGG16, the largest 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 54% test accuracy (compare to 67% test accuracy in plaintext on the same model), which prior work estimated to take *14 days* but did not actually execute the training [88].

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 super-quadratically 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.

6.4 Impact of Fixed-point Precision

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. Figure 5 quantifies what precision is necessary to train each network, showing the final test accuracy after 10-epoch P-Falcon training runs at increasing amounts of precision from 10 to 26 bits. There is a clear distinction between precisions at which each network fails to train and those that allow the networks to do better than random guessing. While 13 bits of precision are sufficient for private inference, even SecureML cannot begin to train until more than 14 bits of precision are used. For the deeper networks, AlexNet and VGG16, which see very small gradients by the end of backpropagation, a higher precision (at least 22 and 24 bits, respectively) is

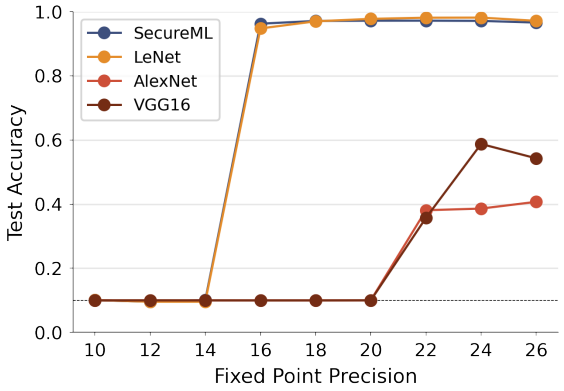


Figure 5: Test accuracy as the fixed-point precision increase for each network architecture, after 10 training epochs using P-Falcon. The dashed line indicates the baseline accuracy when randomly guessing. Sharp increases in training accuracy indicate that the model now has enough precision to fully backpropagate gradients.

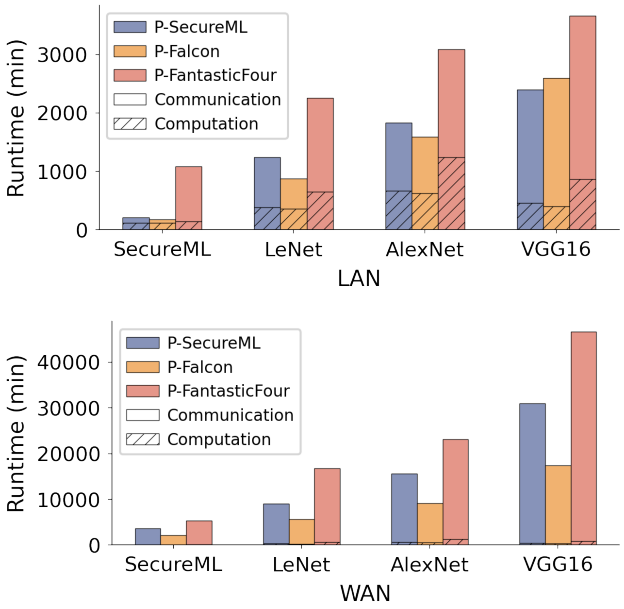


Figure 6: Computation and communication overhead for private training iterations in LAN and WAN settings. Piranha significantly accelerates local computation on a GPU, resulting in communication costs dominating overall runtime as latency between parties and network size increases.

needed. More importantly, these results indicate that computation over 64-bit integers is not just desirable for secure training of large networks but in fact, necessary. In addition, as the size and depth of MPC-trained models increases, the amount of fixed-point precision necessary will likely grow as well, or the use of adaptive fixed-point computation may be necessary.

6.5 Computation and Communication Cost

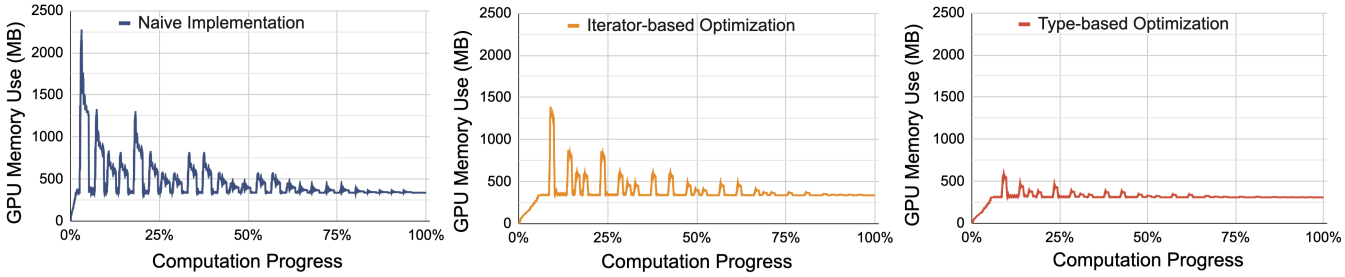
We measure Piranha's ratio of computation time (time spent performing GPU-accelerated local computation) to network overhead (time spent waiting for other parties) in Figure 6 for both LAN and WAN settings. In the LAN setting, all parties executed on GPUs in the same datacenter, with approximately 1.5 ms of observed latency, while in the WAN setting, we run the parties in datacenters in different geographic locations with 60ms of latency in between. When the network is fast, so is the end-to-end runtime: Piranha completes training iterations over each network architecture in ∼3 seconds or less over LAN but takes up to 40 seconds over WAN to perform a 4-party training iteration for VGG. We note that the raw time spent on local computation is the same in both settings, but the computation-communication ratio is very different. We observed that parties in the LAN setting spent between 15% and 60% of the time on compute (on Secure ML and VGG16, respectively), while in comparison, parties in the WAN setting never spent more than 6% of their time on computation. Piranha inherits its communication behavior from the protocol that it is accelerating, and so it does not fundamentally alter the network overhead that would be observed. It is likely that future protocols performing increased computation in favor of minimizing communication [83, 84] would see a large benefit from executing on Piranha in a WAN setting.

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

Table 3: 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 limited to 32 [79].

6.6 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



(a) (b) (c)

Figure 7: Memory footprint over a VGG16 forward pass. Each point is a snapshot of the total GPU memory allocation (in MB) at each memory operation (allocation or de-allocation). Figure 7a corresponds to a naive GPU implementation, Figure 7b measures the footprint after iterator-based optimizations, and Figure 7c after efficiently sizing bit-containing data structures.

sizes of 32 [79]. 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.

We illustrate the benefits of two main memory-based modifications we discussed in Section 4.2 in reducing our memory footprint. We consider three Piranha versions: (1) a naive computation approach with large uniform data types and minimal in-place computation, (2) an iterator-based implementation that seeks to avoid memory allocation when at all possible, and (3) a version that correctly sizes data types to minimize wasted memory (e.g. in the case of secret-shared bits). For each version, Figure 7 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. We also measure the maximum VGG16 batch size that Piranha can support with on-GPU memory, and total runtime and peak memory footprint with a batch size of 32 to compare between versions. Peak memory footprint indicates the amount of temporary allocations necessary at runtime, which can significantly strain the GPU's available memory and preclude larger batch sizes.

Figure 7a shows the memory allocation trace for the naive P-Falcon implementation described in Figure 2(a), which requires a significant amount of data allocation while executing ReLU comparisons, where secret-shared values are expanded into bitwise format. Driven by the initial network layers with larger inputs, the peak GPU memory load is 2.28 GB, a 7× increase over the allocation required for the network itself (345 MB). The total number of memory operations is high: almost 16,000 such allocations and frees are performed over the course of the computation. During a 32 batch size run, this approach can complete a training iteration in 27 seconds with a peak memory footprint of 14.9 GB, or 93% of available GPU memory.

Figure 7b 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 7a. The number of GPU memory operations also drops, resulting in almost 4× 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 7b would not support training runs over 128-image batches. Similar to the naive implementation, the maximum batch size the iterator-based version can train with is 32, but only incurs a maximum memory footprint of 8.9 GB, an approximately 60% improvement. Execution time increases slightly to 35 seconds per pass, which we suspect is due to the inherent cost of non-contiguous and indirect memory access.

In Figure 7c, we evaluate the impact of sizing memory appropriately for data at the protocol layer. In the previous versions analyzed above, 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 64-bit values. Modifying Piranha protocols to closely match the size of allocated values with their logical sizes significantly cuts the peak memory usage in Figure 7c by a factor of 2, to 581 MB, or only 250 MB above the baseline model memory requirements. This has an outsized effect on training execution time, as smaller data types require less communication overall: with this change, Piranha can support P-Falcon-based training iterations with a batch size of 256 in just 7.6 seconds, with a maximum memory footprint of 1.8 GB.

Finally, Table 3 shows peak GPU memory usage for Piranha over all networks as it performs training passes using the protocols we implemented on Piranha. For SecureML in particular, the baseline memory used by 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, due to the increased number of local shares it must maintain for each secret-shared value, proportionally increasing memory load.

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

Table 4: 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 [79] 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.

6.7 Comparison with Prior Work

Finally, we compare the runtime and communication overhead of Piranha relative to state-of-the-art protocols for neural network training: a CPU-based implementation, Falcon [88], and a GPU-based implementation, CryptGPU [79]. 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 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 4.

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 4 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 4 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 [21]'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 4 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.

7 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.* [42] and Frederiksen and Nielsen [35] explore the use of GPUs for improving secure computation using garbled circuits and OT extensions. Delphi [57] uses GPUs to improve the performance of linear components of the computation. In a more recent work, CryptGPU [79] building on top of the CrypTen framework [21] uses GPUs for the entire computation. Recently, GForce [61] shows the benefits of GPU acceleration for secure inference. In a somewhat related effort, cuHE [22] and PixelVault [81] use GPUs for homomorphic encryption, securing keys, and encryption operations. Visor [67] has looked at using GPUs for secure computation over enclaves, while Slalom [80] investigates NN inference on trusted hardware.

A number of general purpose frameworks have improved the practical performance of MPC. In the dishonest majority setting, a number of works [8, 19, 34, 46, 47, 73] improve the performance of the original SPDZ protocols [25, 27]. Helen [93] proposes a system to train a linear model in a dishonest majority setting. Poseidon [75] 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 [43, 44, 55, 57, 60, 63, 70, 71] 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 [18, 23, 49, 59, 64, 79, 82]. Similarly, 4-party computation frameworks include [17, 23, 49, 69]. Other proposed frameworks include [68, 92]. 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 [14, 15, 76, 90]. Finally, our platform can be used to implement efficient protocols for other applications such as sorting networks [20], ORAMs [38, 85], and differential privacy [32, 87].

A number of libraries with varying infrastructures are open sourced. MP-SPDZ and SCALE-MAMBA [7, 45] implement a number of protocols, including most of the dishonest majority protocols. CrypTen [21] implements a few protocols over PyTorch. Other popular libraries providing a number of useful secure computation tools include [29, 72]. There also exist open-source libraries for privacy-preserving machine learning such as Rosetta and PySyft [1, 2], but no open source library that enables general secure computation applications to benefit from the use of GPUs or the development of new accelerated protocols. Piranha can not only fill this gap, but reduce the performance gap between plaintext and privacy-preserving computation.

8 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. Piranha's modular structure provides wide applicability for other projects to use GPU acceleration without requiring expert knowledge. To demonstrate that Piranha as a general-purpose platform 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× performance improvement over CPU-based implementations. Finally, using Piranha, we are able to securely train a realistic neural network end-to-end, with over 100 million parameters, in a little over a day.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Mayank Varia, for their helpful feedback. We would also like to thank Vikash Sehwag for his help on learning stability, Haicheng Wu and Daguang Xu for their CUTLASS pointers, and Vivian Fang and students in the RISELab security group for providing feedback and illustration that improved the presentation of the paper. This work is supported by NSF CISE Expeditions Award CCF-1730628, NSF CAREER 1943347, and gifts from the Sloan Foundation, Alibaba, Amazon Web Services, Ant Group, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk, and VMware. This work is also supported by a National Defense Science & Engineering Graduate Fellowship.

References

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

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

[3] cublas. https://developer.nvidia.com/cublas.

[4] Cutlass: Cuda templates for linear algebra subroutines.

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

[6] S. Albawi, T. A. Mohammed, and S. Al-Zawi. Understanding of a convolutional neural network. In *2017 International Conference on Engineering and Technology (ICET)*, pages 1–6. Ieee, 2017.

[7] 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.

[8] 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.

[9] 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.

[10] T. Askar, B. Shukirgaliyev, M. Lukac, and E. Abdikamalov. Evaluation of pseudo-random number generation on gpu cards. *Computation*, 2021.

[11] J. L. Ba, J. R. Kiros, and G. E. Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[12] 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.

[13] K. W. Bowyer, K. Hollingsworth, and P. J. Flynn. Image understanding for iris biometrics: A survey. *Computer vision and image understanding*, 110(2):281–307, 2008.

[14] 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)*, pages 291–308, 2019.

[15] 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*, pages 489–518, 2019.

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

[17] 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.

[18] 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.

[19] 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.

[20] R. A. Chowdhury, V. Ramachandran, F. Silvestri, and B. Blakeley. Oblivious algorithms for multicores and networks of processors. *Journal of Parallel and Distributed Computing*, 2013.

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

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

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

[24] 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.

[25] 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*, pages 1–18. Springer, 2013.

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

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

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

[29] 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.

[30] 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*, pages 86–95, 2005.

[31] R. A. Dunne and N. A. Campbell. On the pairing of the softmax activation and cross-entropy penalty functions and the derivation of the softmax activation function. In *Proc. 8th Aust. Conf. on the Neural Networks, Melbourne*, volume 181, page 185. Citeseer, 1997.

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

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

[34] 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.

[35] 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.

[36] 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.

[37] C. Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. crypto.stanford.edu/craig.

[38] O. Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *ACM Symposium on Theory of Computing (STOC)*, 1987.

[39] 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.

[40] 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.

[41] R. Hecht-Nielsen. Theory of the backpropagation neural network. In *Neural networks for perception*, pages 65–93. Elsevier, 1992.

[42] 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.

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

[44] 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.

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

[46] 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*, pages 830–842. ACM, 2016.

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

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

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

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

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

[52] 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.

[53] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[54] D. T. Lee Howes. Chapter 37. efficient random number generation and application using cuda. https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-37-efficient-random-number-generation-and-application.

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

[56] 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.

[57] 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.

[58] MNIST database. http://yann.lecun.com/exdb/mnist/. Accessed: 2017-09-24.

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

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

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

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

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

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

[65] C. Perlich, B. Dalessandro, T. Raeder, O. Stitelman, and F. Provost. Machine learning for targeted display advertising: Transfer learning in action. *Machine learning*, 95(1):103–127, 2014.

[66] J. Perols. Financial statement fraud detection: An analysis of statistical and machine learning algorithms. *Auditing: A Journal of Practice & Theory*, 30(2):19–50, 2011.

[67] 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.

[68] 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.

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

[70] 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.

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

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

[74] 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.

[75] 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.

[76] 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.

[77] 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)*, pages 99–104. IEEE, 2016.

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

[79] 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.

[80] F. Tramer and D. Boneh. Slalom: Fast, verifiable and private execution of neural networks in trusted hardware. *arXiv preprint arXiv:1806.03287*, 2018.

[81] 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.

[82] S. Wagh. *New Directions in Efficient Privacy Preserving Machine Learning*. PhD thesis, Princeton University, 2020.

[83] S. Wagh. BarnOwl: Secure Comparisons using Silent Pseudorandom Correlation Generators. In *Tech Report*, 2022.

[84] S. Wagh. Pika: Secure Computation using Function Secret Sharing over Rings. In *Privacy Enhancing Technologies Symposium (PETS)*, 2022.

[85] S. Wagh, P. Cuff, and P. Mittal. Differentially private oblivious RAM. In *Privacy Enhancing Technologies Symposium (PETS)*, 2018.

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

[87] S. Wagh, X. He, A. Machanavajjhala, and P. Mittal. DP-Cryptography: marrying differential privacy and cryptography in emerging applications. In *Commun. ACM*, 2020.

[88] 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.

[89] T. Yamanouchi. Chapter 36. aes encryption and decryption on the gpu. https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-36-aes-encryption-and-decryption-gpu.

[90] 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)*, pages 1607–1626, 2020.

[91] A. Yao. Protocols for Secure Computations. In *Foundations of Computer Science (FOCS)*, 1982.

[92] 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.

[93] 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.

A Discussion, Limitations, and Future Work

We show in Section 6 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 [3], 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 [30].

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 1, 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 [91], or adding homomorphic encryption support [22] 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.

B Comparison with Floating Point Kernels

We mention in Section 3 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 5, we compare the 32- and 64-bit kernels that Piranha uses, implemented with CUTLASS [4], against state-of-the-art 32- and 64-bit floating point kernels from cuBLAS [3]. 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.

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 5: Runtime for matrix multiplication kernels used in Piranha vs. the cuBLAS implementation for different sizes.

We benchmark the runtimes for the matrix multiplication kernels used in Piranha vs. the cuBLAS implementation on various sizes of matrices in Table 5. 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.