Piranha: A GPU Platform for Secure Computation

本次介绍的是发表在Usenix Security'22上的论文Piranha，该论文来利用GPU加速MPC协议，提升了多个现有协议的性能，并支持隐私保护机器学习模型训练和预测。
论文链接：https://eprint.iacr.org/2022/892
代码链接：https://github.com/ucbrise/piranha

1. Introduction

利用GPU加速安全计算的效率是当前的一个主要研究方向，不仅仅是基于秘密分享的MPC协议，在同态加密领域，GPU加速也有很多工作。至于本文涉及的基于秘密分享的MPC协议，近年来的比较有影响力的工作主要有Delphi和CryptGPU，不过前者只是用GPU加速卷积计算，后者则将整数分解，然后利用Pytorch中的浮点数Kernel加速分解的子模块的安全三方计算，然后合并得到结果。这两篇论文的详细介绍可以参考我们之前的博客：Delphi博客，CryptGPU博客。

目前，还没有直接在GPU上实现整数Kernel，以支持MPC协议的工作。Piranha面向的则是这个问题。

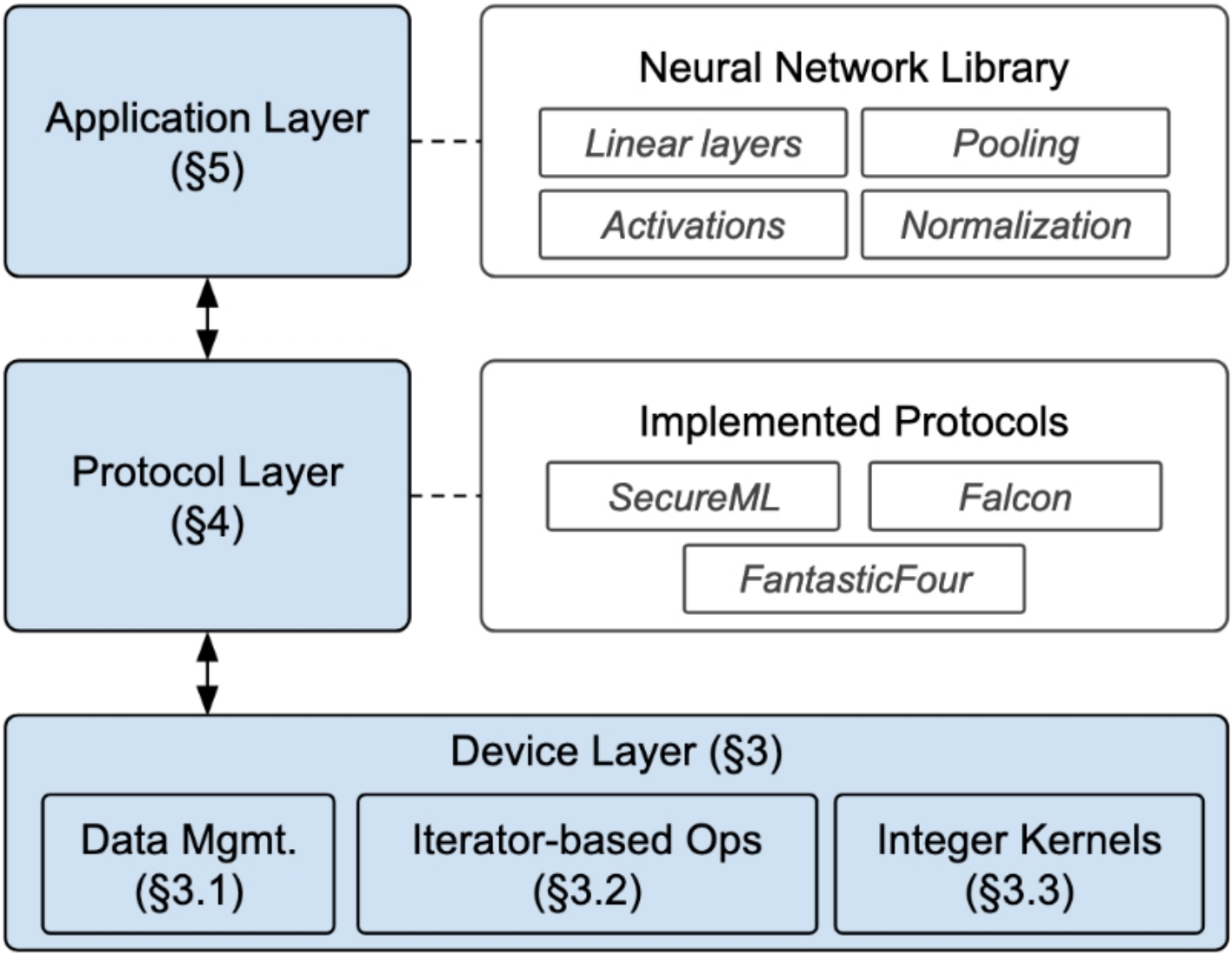
Challenges & Insights: 为了支持GPU加速的MPC协议，本文总结了三个挑战：

1. *Protrocol-independent acceleration.* 不同的MPC协议对于同样的功能模块计算流程是不同的，例如两方和三方的乘法计算协议就完全不同。如果加速技术依赖于协议本身的性质，那么每次希望在GPU加速基础上设计新协议的时候，就必须重新构建关于GPU加速的底层模块。另一方面，不同的计算协议都可以分解为 *本地计算+通信*。Piranha以vector shares为基本的处理单元来计算local operations，然后结合参与方之间的通信来实现计算。通过对local data的秘密分享抽象，Piranha能够将所有的数据和计算保持在GPU上，最大化GPU的加速效率并尽可能减少通过CPU的各个参与方之间的数据传输；
2. *Enabling integer-basec GPU Computation.* MPC协议是操作在整数上的，可惜的是GPU只关注在明文浮点数计算。为了填补这个空白，Piranha提供了一个integer-based Kernel并以此为基础表示integer-based shares、加速基本算子计算；
3. *Supporting large MPC problems in limited GPU memory.* 由于GPU的存储（e.g., 12、16GB）比内存（e.g., 64GB）小很多，因此会限制应用的规模。进一步，秘密分享会成倍的增加内存占用，因此会加剧存储的问题。问了缓解这个问题，本文：1）首先利用in-place operations 来进行本地份额的计算，当且仅当明显需要额外存储的时候才分配GPU内存；2）第二点，MPC协议可能会需要与整数Kernel不兼容的非标准访存模式。简单的将数据拷贝然后组织成预期的布局会需要额外的存储

空间。Piranha针对这个问题基于GPU内存视图提出了内存-高效的计算以支持in-place computation。从而避免申请临时内存或者数据传输。

为了验证Piranha的性能，本文实现了三种协议：1）SecureML（两方），2）Falcon（三方），和3）FantasticFour（四方）。和之前的CPU版本的协议相比，Piranha对于模型训练方面能够提升训练时间$16-48\times$。更重要的，Piranha是第一次实现能力能够大约一天时间完成一个实用化神经网络（比如VGG）的完整训练。

2. System Architecture



如上图所示，Piranha采用模块化设计，分为三层：

1. Device Layer：该层提供了目前GPU库缺少的整数Kernel，能够协议无关的加入任意基于秘密分享的MPC协议。具体来说，该层提供了对于GPU-based 整数向量的一个抽象，能够表示向量的多方分享的份额，这些份额保存在GPU上支持计算，而通信交互则通过 *GPU<-->CPU<-->GPU* 来进

行；另一方面，本文提出的整数Kernel支持对local分享份额的常见功能函数，比如对位相加、矩阵乘法等；

2. Protocol Layer：该层实现了多个协议，包括两方、三方和四方计算协议与黑盒模式下设计的算子。并且，为了最大利用GPU有限的存储并弥补Device Layer的不足，该层对非标准化访存模式提供了针对GPU上local份额的iterator-based视图以支持in-place computation从而满足GPU的存储约束；
3. Application Layer：该层使得上层应用（机器学习模型训练和预测）和底层协议完全独立。以PPML为例，其核心逻辑是将上层应用的函数转化为Protocol Layer对应的MPC协议，然后进一步调用Device Layer的基本本地运算和通信，从而实现需要的应用。

Threat Model: 虽然Falcon和FantasticFour支持恶意模型安全，但是本文还是只面向半诚实安全。

3. Device Layer

该层主要解决两个问题：1）处理向量化的GPU数据；2）支持加速基于整数的计算。

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提供了DeviceData抽象以访问GPU内存（Listing 1），其可以用C++的整数类型实现，例如uint32_t, uint64_t。逻辑上，DeviceData对应秘密分享的本地份额。其操作的基本单元是份额向量（share vectors）而不是单个的数。因此，其对应的是C++中的 std::vector<>，初始化如 Line 1-4。对位计算（element-wsie operations）可以直接进行，底层会调用GPU的多个线程进行 (Line 7-8)，对于矩阵乘法，在DeviceData的支持下，安全矩阵乘法的效率比CPU版本的提升了$200\times$ (Line 11)。而通信，则是借助CPU是实现。DeviceData也提供了简单的接口（Line 14-15）

3.2 Iterator-based operations

为了缓解GPU内存有限的问题，Piranha实用基于迭代器的抽象来尽可能减少冗余临时内存申请。Piranha的迭代器支持开发者对于数据向量构建基于视图（view）并基于视图遍历操作数据，而不是改变物理内存布局。例如，对于成对乘法，简单的方案是将索引为奇数和偶数的数据分别拷贝到两块临时内存，然后进行乘法。但是这需要申请额外的存储空间。Piranha中基于迭代器的方法允许开发者定义奇数、偶数视图，并根据视图进行in-place element-wise操作，不需要额外的存储（Line 18-21，令offset=0, 1可以定义偶数和奇数视图）。

3.3 Integer kernels

之前面向GPU的整数Kernel解决方案（CryptGPU）是将整数分解为若干个小整数的组合（$x = x_3 2^{48} + x_2 2^{32} + x_1 2^{16} + x_0$），然后利用浮点数Kernel的尾数部分计算小整数，然后根据小整数计算结果组合成最终结果。这种方法需要多次调用浮点数Kernel。本文直接在CUTLASS的一般性矩阵乘法和卷积Kernel的模板上开发整数Kernel，从而使得一次Kernel调用就可以满足整数计算。另外，本文的模块化设计方便在今后嵌入更加高效的Kernel实现。

4. Protocol Layer

本文实现了SecureML，Falcon，和FantasticFour协议的GPU版本，并且还提出了Memory-efficient协议来进一步减少内存开销。

4.1 MPC protocol implementation

利用DeviceData可以实现local shares的本地化操作。以三方replicated secret sharing为例，加法和数乘可以本地计算得到。乘法则需要resharing。更一般，乘法可以推广为矩阵乘法。矩阵乘法的代码示意如下：

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.

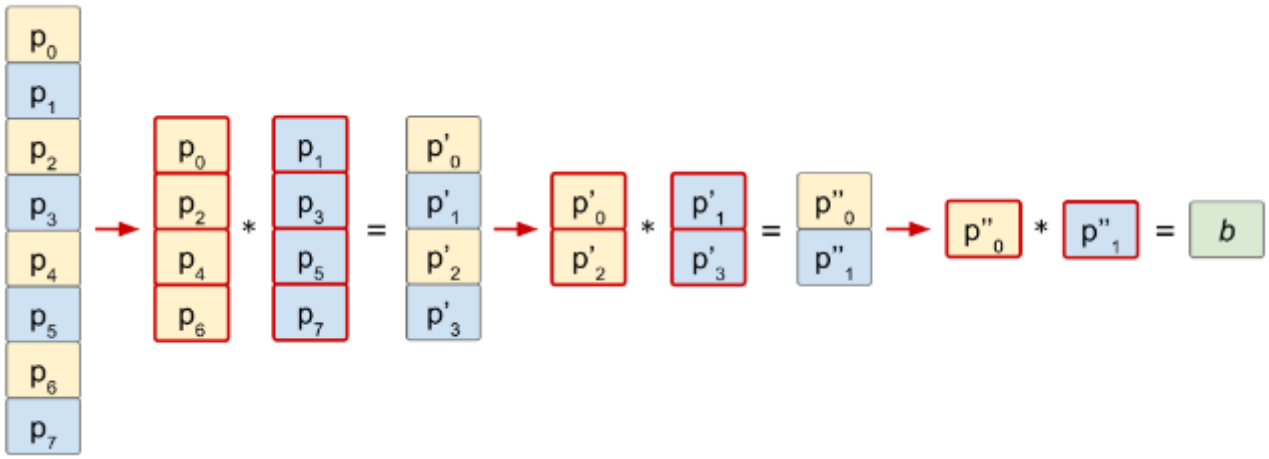
```
1  // Replicated secret sharing class
2  class RSS<T> {
3    DeviceData<T> shareA, shareB;
4  }
5
6  void RSS<T>::matmul(RSS<T> a, RSS<T> b,
7              RSS<T> c, ...) {
8    DeviceData<T> localC;
9
10   localC += gpu::gemm(a.shareA, b.shareA, ...);
11   localC += gpu::gemm(a.shareA, b.shareB,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

除此之外，Piranha利用GPU生成大量随机数用于MPC协议。

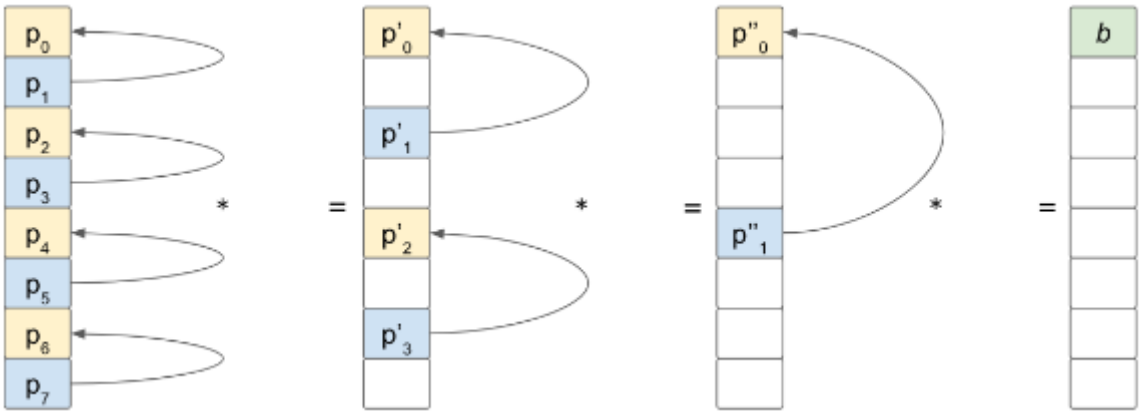
4.2 Memory-efficient protocols

在比较协议中，最重要的是提取最高有效位。进一步，最高有效位提取中最关键的是进位计算：CarryOut， 本节简述如何利用3.2节介绍的iterator-based抽象视图来优化CarryOut。给定 a 和 b 的比特分解 $(a_{k-1}, ..., a_0)$ 和 $(b_{k-1}, ..., b_0)$，CarryOut计算得到 c_k。基本思路是利用进位加法器计算 $\log_2 k$ 轮实现进位。对于第$i \in \{1, 2, ..., \log_2 k\}$轮，最关键的是计算邻接传播比特的AND门：$p'_j = p_{2j} \wedge p_{2j+1}$。

简单的方法是将是每一轮将所有p分为偶数、奇数两部分，然后进行对位乘法。但是这样会需要申请额外的内存（$\log_2 n$次额外的数据拷贝）。计算示意图如下所示：



本文则是基于iterator的视图，构建了访问偶数索引和奇数索引下数据的两个视图，以两个视图为基础使用GPU Kernel计算得到传播比特。并且得到的计算结果可以放在第一个视图对应的已申请内存中，不需要额外的数据拷贝和内存申请。示意图如下所示：



4.3 Reusable protocol components

其他重要的模块包括比较函数和近似计算。

比较函数: 比较函数本文使用edaBits：$[r]_M, [r_0]_2, [r_1]_2, ..., [r_m]_2$ 其中 $r \xleftarrow{\$} \mathbb{Z}_M$ 且 $m+1 = \log_2 M$ 来实现。利用edaBits可以将算数分享的比较问题转化为布尔分享的转化问题，而布尔分享的比较可以通过CarryOut实现。

近似计算: 对于平方根计算和倒数计算，本文首先求其输入的最近二次幂（和Falcon类似），然后将输入转化到0.5到1之间。进而使用如下的泰勒级数多项式近似：

$$\text{sqrt}(x) = 0.424 + 0.584(x)$$
$$1/x = 4.2445 - 5.857(x) + 2.630(x^2)$$

5. Application Layer

该层提供了面向神经网络模型训练的接口，具体来说是神经网络函数的实现。本文支持的面向NN的函数如下：

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.

这些实现独立于协议的实现。因此，可以在未来很方便的按照统一的接口嵌入新的协议。简单的调用如下所示：

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.

```
1  // Fully connected layer forward pass
2  template<typename Share>
3  void FCLayer<Share>forward(Share input) {
4    matmul(input, this->weights,
5        this->activations, ...);
6    this->activations += this->bias;
7  }
```

对于反向传播，假设NN最后一层的输出是$\mathbf{x} = [x_1, x_2, ..., x_k]$，则最后一层的logits计算：$p_i = \frac{e^{x_i}}{\sum_j e^{x_j}}$。本文发现

$$p_i = \frac{e^{x_i - x_{max}}}{\sum_j e^{x_j - x_{max}}}$$

其中 $x_{max} = max(x_1, .., x_k)$。
由于e^x现在的输入≤ 0，本文用多项式$\mathrm{appExp}(\cdot)$近似计算 e^x，并加入了 10^{-3} 的偏置。从而兼顾效率和保证激活函数的幅度（不增加）。该方法之前在CryptGPU中详细介绍过，可以参考之前的博客。

6. Evaluation

本文关注在线计算性能和通信，和基于MP-SPDZ的实现相比，大大提升了算子和整体训练的性能。

矩阵乘法、卷积和ReU算子的提升如下：

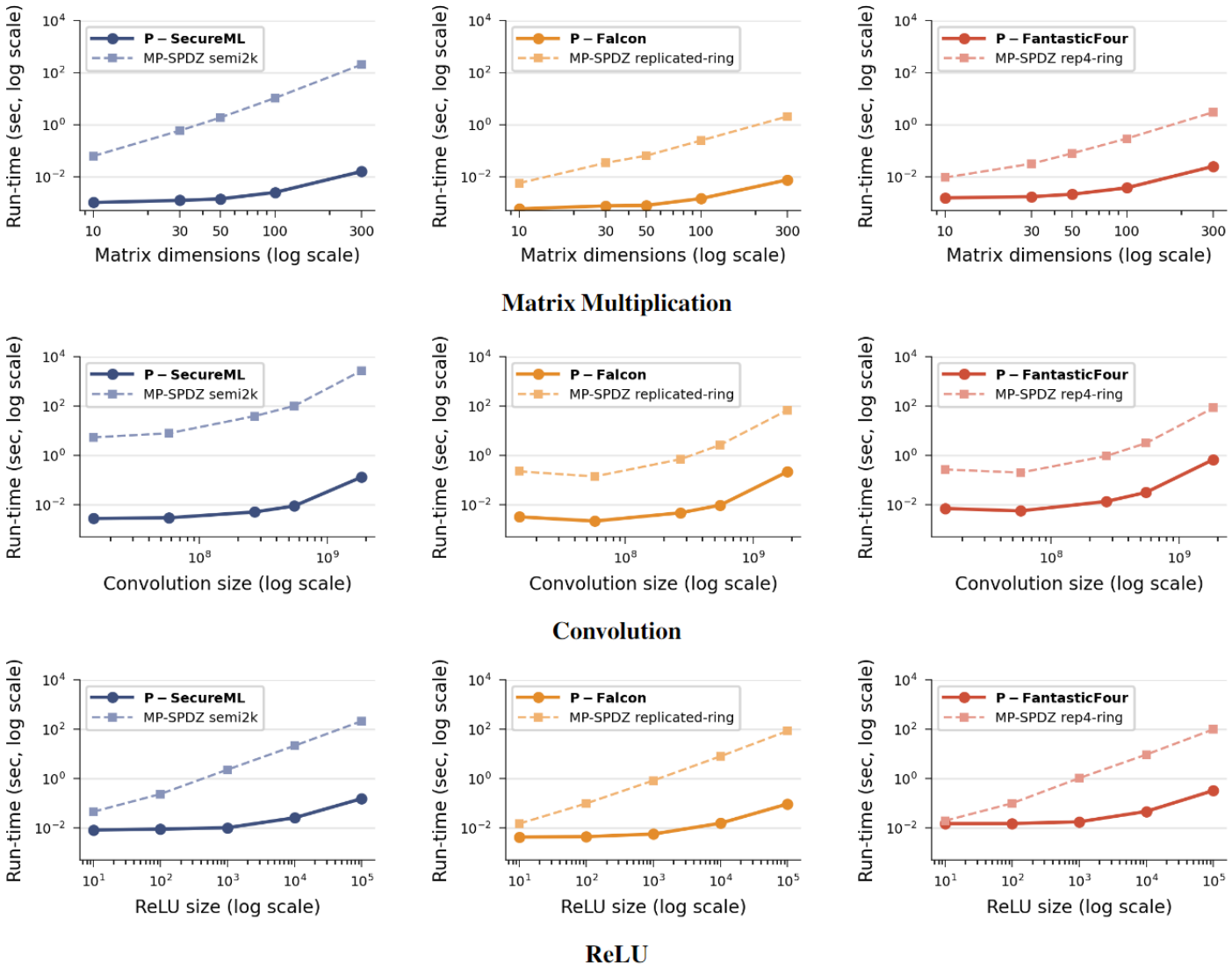


Figure 4: The figures benchmark secure protocols for matrix multiplication, convolutions, and ReLU across 2-, 3-, and 4-party protocols for various sizes of these computations. Piranha consistently improves the run-time of these computations, with improvements as large as 2-4 orders of magnitude for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

在线训练的时间和通信开销如下：

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.

三方协议下，和Falcon、CryptGPU做了训练和预测开销的对比。

	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.

本文还做了其他关于精度、计算-通信平衡、和内存方面的实现，该部分请参考原文。