这次分享的是Sijun Tan等人发表在IEEE S&P'21的CryptGPU。

Background & Motivation

目前大多数基于安全多方计算（MPC）的隐私保护机器学习方案都是运行在CPU上的，但是在明文机器学习领域GPU已经成为一项不可缺少的硬件设备。GPU强大的算力能够大大加快机器学习模型，尤其是神经网络的计算速度。为了让MPC的隐私保护机器学习方案也能够充分利用GPU的计算性能、加快基于MPC的隐私保护机器学习训练和预测效率，本文提出了CryptGPU。

本文的主要贡献在于将基于MPC的密码学协议迁移到GPU上，实现了大幅度的性能优化。然而，将MPC技术从CPU迁移到GPU会遇到一系列的挑战。虽然NVIDIA的CUDA平台能够支持一般性的计算，但是将面向CPU的代码直接运行在GPU上并不能直接带来预期的性能提升。为了实现高效的基于GPU的MPC技术，设计者必须要处理CPU和GPU之间的架构差异：

1. 利用已有的CUDA Kkernels：第一个挑战则是现有的面向深度学习的、高度优化的CUDA kernels是面向浮点数（floating-point）计算设计的，并且这些kernels并不能直接在整数（integers）上计算。而MPC计算则主要计算在离散空间（环或者域）。为了利用现有的、高度优化的CUDA kernels，我们必须将整数计算无损地嵌入到浮点数计算中。为了解决这个问题，CryptGPU 构造了 'CUDALongTensor' 类，其能够对整数张量进行建模，并将整数计算无损的转化为对应的浮点数计算。
2. GPU友好的密码技术：GPU的架构适合对一块数据（block）进行简单快速的计算，例如component-wise的加法和乘法。但是，GPU却对条件语句不是很友好。因此，虽然现有的GPU可以支持整数加法，但是在取素数模下做整数加法、乘法会造成大量开销（例如，对于point-wise加法40倍的差距）。因此，将密码学技术迁移到GPU上必须要考虑这些架构影响。例如，和姚氏混乱电路相比，基于向量化秘密分享的MPC协议更能发挥GPU的并行性能。进一步，为了避免模素数运算，CryptGPU采用模 $\mathbb{Z}_{2^k}$ 的环，例如 $\mathbb{Z}_{2^{64}}$ 。

实验显示，对于Falcon中基于CPU的线性层方案，CryptGPU可以带来25-72倍的性能提升。对于卷积操作（卷积核和数据都是秘密分享的），CryptGPU快150倍。即便是对非线性函数，例如ReLU，CryptGPU也能带来10倍的加速。最后，作者在CrypTen的技术上实现了CryptGPU并开源，链接。

Systen Design and Architecture

如前所述，CrypGPU的设计原则：1）充分利用现有的线性代数计算CUDA kernerls；2）使所有的计算都在GPU上进行。CryptGPU面向64比特的整数环上的计算，而现有的GPU线性计算库却是支持64比特浮点数。接下来将主要如何将 $\mathbb{Z}_{2^{64}}$ 上的 整数计算无损的嵌入到54比特浮点数计算中。为了实现这个目标，本文有如下最重要的观察：

1. Exact Computation for Small Values: 64比特的浮点数具有52比特的精度，而且可以表示所有在区间 $[-2^{52}, 2^{52}]$ 内的整数。也就是说，我们可以准确计算乘法ab，其中 $a, b \in \mathbb{Z} \cap [-2^{26}, 2^{26}]$。

2. Bilinearity: 矩阵乘法和卷积操作被称为双线性操作（bilinear）。即对于任意 $\mathbf{A}_0, \mathbf{A}_1, \mathbf{B}_0, \mathbf{B}_1$，有
$(\mathbf{A}_0 + \mathbf{A}_1) \circ (\mathbf{B}_0 + \mathbf{B}_1) = \mathbf{A}_0 \circ \mathbf{B_0} + \mathbf{A}_0 \circ \mathbf{B}_1 + \mathbf{A}_1 \circ \mathbf{B}_0 + \mathbf{A}_1 \circ \mathbf{B}_1,$
其中 $\circ$ 表示任意双线性计算。假设我们将输入表示为更小的基的形式，那么有$\mathbf{A} = \mathbf{A}_0 + 2^{16}\mathbf{A}_1$ 和 $\mathbf{B} = \mathbf{B}_0 + 2^{16}\mathbf{B}_1$ 。 双线性保证了 $\mathbf{A} \circ \mathbf{B}$ 可以通过 $\mathbf{A}_0 \circ \mathbf{B_0}, \mathbf{A}_0 \circ \mathbf{B}_1, \mathbf{A}_1 \circ \mathbf{B}_0, \mathbf{A}_1 \circ \mathbf{B}_1$ 的线性组合来获得。线性组合的过程只涉及到 element-wise 加法和数乘。
3. CUDA kernerls for element-wise operations: 现有的CUDA kernerls可以支持整数上的element-wise加法和数乘。

为了实现双线性操作 $\circ$, 例如矩阵乘法和卷积，CryptGPU首先将输入矩阵 $\mathbf{A}, \mathbf{B} \in \mathbb{Z}_{2^{64}}^{n \times m}$ 分解为数值范围小的若干个矩阵组合 $\mathbf{A}_1, ..., \mathbf{A}_k$ 和 $\mathbf{B}_1, ..., \mathbf{B}_k \in \mathbb{Z}_{2^w}^{n \times m}$，满足 $\mathbf{A} = \sum_{i=1}^{k} 2^{(i-1)w}\mathbf{A}_i$ 和 $\mathbf{B} = \sum_{i=1}^{k} 2^{(i-1)w}\mathbf{B}_i$。进一步，利用浮点数CUDA kernerls在GPU上求k^2个$\mathbf{A}_i \circ \mathbf{B}_j$。因为$\mathbf{A}_i \circ \mathbf{B}_j$ 的取值范围不会超过 2^{52}，这些中间结果都可以准确计算。最终，这些成对的乘积被重新解释编码为64比特的整数。如此，$\mathbf{A} \circ \mathbf{B}$ 可以通过 $\mathbf{A}_i \circ \mathbf{B}_j$ 的线性组合计算得到。因为最终的结果需要模 2^{64}，所以只需要计算下标满足 $w(i + j - 2) < 64$ 的 $\mathbf{A}_i \circ \mathbf{B}_j$。在计算浮点数卷积的时候，CryptGPU取 $k = 4, w = 16$。如此每个双线性计算被分解为10个成对的双线性积。另外，有如下三个Remark:

1. Smaller Number of Blocks： 直觉上可以设 $k = 3$ 从而每个块需要包含22比特的值。乘法得到44比特的值不会超过范围，但是矩阵乘法或者卷积中的加法使得只能进行 $2^8 = 256$ 次加法，这在实际应用中很快就会被超过。如果分为 $k = 4$ 块，每块包含 16 比特的值，那么可以进行 2^{20} 的中间加法，可以满足实际应用。
2. Overhead of Block-wise Decomposition: 每个双线性计算被分解为 $O(k^2)$ 个同样大小的双线性计算，为了减少计算开销，CryptGPU利用GPU的并行技术同时进行多个计算。对于卷积，利用 cudnnConvolutionForward 加速；对于矩阵乘法，利用 cublasSgemm 加速。对于小输入(64×64)，只带来2倍的额外计算开销；对大的输入(224×224)，额外引入$9\times$的时间。虽然时间开销引入的不多，但是带来了很大的存储需求，因此大大限制了在隐私训练中的batchsize。
3. Comparsion with Delphi: 之前的Delphi方案也提出利用GPU提升计算性能，但是该方案是面向数据是秘密分享，模型是对服务器一方来说是明文的情况（类似MiniONN）。所以，该方案可以限制参数和卷积、矩阵乘法结果不超过 $[-2^{52}, 2^{52}]$。实际参数上，Delphi选择了环 $\mathbb{Z}_{2^{32}}$ 和15比特的小数部分精度。而CryptGPU是完全计算在秘密分享上的，而且扩展到了更深的神经网络和更大的数据集上。

CUDALongTensor abstraction: CryptGPUt提供了CUDALongTensor来将64比特整数计算嵌入到64比特的浮点算术计算。基于CrypTen中的 MPCTensor，本文实现了 CUDALongTensor。在计算中：

1. 如果现有的CUDA kernerls能够支持该运算，例如element-wise加法和乘法，直接利用现有的 CUDA kernerls;
2. 而在卷积和矩阵运算双线性计算，则调用 CUDALongTensor。

Threat Model & Cryptographic Design

接下来介绍威胁模型和协议设计。

Threat Model

CryptGPU适用于三方计算，能够在诚实大多数（honest-majority）场景下抵抗半诚实的敌手，保护数据隐私。在计算过程中，所有的数据都是在秘密分享模式下。本文忽略输入分享和输出重构的一点开销，着重于计算过程中的开销。

Cryptographic Building Blocks for Private Inference

CryptGPU采用ABY3中的2-out-of-3 replicated secret sharing，数据和模型都以秘密分享的形式分布在三个不合谋的服务器上。为了编码浮点数，本文和之前的方案一样将浮点数保留 $t = 20$ 比特小数位，并编码到 $\mathbb{Z}_{2^{64}}$。

1. Protocol Intialization: 协议初始化将生成用于resharing的随机数，具体见ABY3；
2. Linear Operations：包括加法和线性组合（权重是明文），该部分不需要交互；
3. Multiplication： 乘法需要本地计算得到3-out-of-3 sharing之后进行一次resharing，需要一次交互。额外的，要进行rescale，截断t比特小数位防止高位溢出。本文采用ABY3中的第一种截断方法。如此，一共需要两轮交互。
4. Convolution & Matrix Multiplication: 调用前文提到的分解方案方法分解输入，进行卷积或者矩阵乘法，然后线性组合得到结果。分解之后的计算和ABY3中的计算一样。
5. Most Significant Bit: 为了计算激活函数，要首先提取输入 $[x]^n = (x_1, x_2, x_3)$ 的最高有效位。本文采用ABY3中A2B的方法，对 $[x_1]^2 = (x_1, 0, 0), [x_2]^2 = (0, x_2, 0)$，和 $[x_3]^2 = (0, 0, x_3)$ 运行加法电路求和，得到 $[x]^2$。然后提取最高位 $[msb(x)]^2$。这需要 $\log_2(n)$ 轮交互。
6. ReLU: 得到 $msb(x)$ 之后，调用ABY3中的 bit injection协议可以计算ReLU。

Cryptographic Building Blocks for Private Training

为了实现基于SGD的深度学习模型训练，我们需要计算softmax函数。对于输入 $[\mathbf{x}, \mathbf{y}]$, 其交叉熵损失为 $\ell_{\mathsf{CE}}(\mathbf{x}, \mathbf{y}) = -\sum_{i\in[d]} y_i \log \tilde{z}_i$, 其中 $\tilde{\mathbf{z}} = \mathsf{softmax}(\mathbf{z})$. 对于 $\mathbf{x} \in \mathbb{R}^d$, softmax 定义为

$$\mathsf{softmax}_i(\mathbf{x}) = e^{x_i} / (\sum_{i\in[d]} e^{x_i}).$$

因此，输出层的梯度计算为 $\bigtriangledown_{\mathbf{z}} \ell_{\mathsf{CE}} = \mathsf{softmax}(\mathbf{z}) - \mathbf{y}$。可以看到，我们不需要交叉熵的值，所以不需要计算 $\log(\cdot)$ 函数。

1. Softmax: 首先，为了限制输入的范围，在做 softmax 之前，先对其正则化 $(\mathbf{x} - \mathsf{max}_i x_i)$。由于 $\mathsf{softmax}(\mathbf{x} - \mathsf{max}_i x_i) = \mathsf{softmx}(\mathbf{x})$，这种变化并不会影响训练结果。正则化之后，$e^{x_i}$ 中 $x_i \leq 0$,从而使得 softmax 计算中的分母在 $[1, d]$ 中。
2. Exponentiation：为了近似计算指数函数 e^x，本文采用极限表征 $f_m(x) = (1 + \frac{x}{m})^m$.

根据 $\ln(1+x)$ 的泰勒展开，在 $|x| < m$ 的条件下，有 $\frac{f_m(x)}{e^x} = \frac{e^{m\ln(1+x/m)}}{e^x} = e^{-O(x^2/m)}$.

因此，degree-m 的 f_m 在以0为中心的 $O(\sqrt{m})$ 范围内近似 e^x。另一种方法是用泰勒级数近似，虽然该方法可以使用 degree-m 的多项式在 $O(m)$范围内提供更好的近似，但是该方法:

- 计算degree-m 多项式需要 m 次乘法和 $O(\log_2(m))$ 次交互，而计算 f_m 只需要 $\log_2 m$ 次乘法；
- 泰勒级数需要的系数可以取到 $\frac{1}{m!}$，在定点表示下很可能近似为0。虽然我们可以使用 $\Pi_{i\in[m]}\frac{x}{i}$ 来计算 $x^m/m!$，但是这需要 $O(m)$ 轮乘法。
- 在我们的设置中，e^x 的输入范围是 $[-\infty, 0]$, 并且 f_m具有当 $x \to -\infty$, $f_m \to 0$ 的趋势，这和 e^x 是一致的。而泰勒级数当 $x \to -\infty$时，则是发散的，会带来很大误差（除非degree特别高）。在本文中， $x \in [-45, 0]$，这需要degree非常高的泰勒级数。

所以，本文采用 f_m 近似 e^x 计算。具体实验中，取 $m = 2^9 = 512$。因此计算f_m需要 $\log_2 m = 9$ 轮乘法。在 $t = 20$的情况下，对于 $\forall x \le 0$，误差 $\le 6 \cdot 10^{-4}$.

3. Division：给定 $[x]^n$ 和 $[y]^n$, 求 $[x/y]^n$。本文中的输入 $1 \le y \in Y$。为了计算倒数 $[1/y]^n$，和之前的工作一样，本文采用基于Newton-Raphsion的迭代近似算法。给定初始解 z_0, 迭代计算 $z_i = 2z_{i-1} - yz_{i-1}^2$。令初始解 $z_0 = 1/Y$，$O(\log_2 Y)$次迭代就能得到 $1/y$ 非常准确的近似值。具体来说，$error_i = |1/y - z_i| = \frac{1}{y}|1 - z_i y| \le \varepsilon$，其中 $\varepsilon = |1 - z_i y|$。代入迭代算法，有 $\varepsilon_i = \varepsilon_{i-1}^2$. i 次迭代之后，误差 $(1 - 1/Y)^{2^i} \le e^{-2^i/Y}$。在实验中，$Y = 200$, 迭代13次，$t = 20$ 情况下，在 $[1, Y]$ 内的输入误差 $\approx 10^{-4}$ （浮点数计算误差 $\approx 10^{-9}$）。
4. Maximum：求最大值则是两个组做个减法然后求差的 msb，然后做个乘法即可。对于 n 个数中求最大值，则是采用tree-manners从而只需要 $\log_2 m$ 交互复杂度。
5. DReLU：求 $\mathrm{ReLU}(x)$ 的导数，只需要求得 $\mathrm{msb}(x) \oplus 1$即可。

Evaluation

本文基于CryTen实现了CrypGPU本文进行了大量的实验，并在LeNet, AlexNet, 和ResNet上和之前的方案进行了对比。三方使用带宽1.25GB/s、延迟0.2ms的局域网连接。为了更好的性能，本文将Max-Pooling替换为Average-Pooling。

Private Inference

	LeNet (MNIST)		AlexNet (CIFAR)		VGG-16 (CIFAR)		AlexNet (TI)		VGG-16 (TI)	
	Time	Comm. (MB)	Time	Comm. (MB)	Time	Comm. (MB)	Time	Comm. (MB)	Time	Comm. (MB)
FALCON	0.038	2.29	0.11	4.02	1.44	40.45	0.34	16.23	8.61	161.71
CRYPTGPU	0.38	3.00	0.91	2.43	2.14	56.2	0.95	13.97	2.30	224.5
Plaintext	0.0007	—	0.0012	—	0.0024	—	0.0012	—	0.0024	—
	AlexNet (ImageNet)		**VGG (ImageNet)**		**ResNet-50 (ImageNet)**		**ResNet-101 (ImageNet)**		**ResNet-152 (ImageNet)**	
	Time	Comm. (GB)	Time	Comm. (GB)	Time	Comm. (GB)	Time	Comm. (GB)	Time	Comm. (GB)
CRYPTFLOW	—	—	—	—	25.9	6.9	40*	10.5*	60*	14.5*
CRYPTGPU	1.52	0.24	9.44	2.75	9.31	3.08	17.62	4.64	25.77	6.56
Plaintext	0.0013	—	0.0024	—	0.011	—	0.021	—	0.031	—

*Value estimated from [5, Fig. 10]

TABLE I: Running time (in seconds) and total communication of private inference for different models, datasets, and systems in a LAN setting. The "TI" dataset refers to the Tiny ImageNet dataset [15]. The plaintext measurements correspond to the cost of inference on plaintext data on the GPU (using PyTorch). Performance numbers for CRYPTFLOW are taken from [5]. Performance numbers for FALCON are obtained by running its reference implementation [45] on three compute-optimized AWS instances in a LAN environment (see Section IV-B). As discussed in Section IV-A, when testing the performance of CRYPTGPU, we replace max pooling with average pooling in all of the networks.

从表1可以看出，对于VGG16，CryptGPU比FALCON快$3.7\times$；但是对于小模型，FALCON性能更好。这是因为小模型不能充分显示GPU的并行能力。和CrypTFlow相比，CryptGPU快大约$2.2\times$。但是和GPU上的明文预测相比，还有$1000\times$的差距。

	$k=1$		$k=64$	
	Time	Comm.	Time	Comm.
AlexNet	0.91	0.002	1.09	0.16
VGG-16	2.14	0.056	11.76	3.60

TABLE II: Running time (in seconds) and total communication (in GB) for batch private inference on CIFAR-10 using a batch size of k.

	$k=1$		$k=8$	
	Time	Comm.	Time	Comm.
ResNet-50	9.31	3.08	42.99	24.7
ResNet-101	17.62	4.64	72.99	37.2
ResNet-152	25.77	6.56	105.20	52.5

TABLE III: Running time (in seconds) and total communication (in GB) for batch private inference on ImageNet using a batch size of k.

进一步，表2和3显示了CryptGPU在batch inference下的性能提升。均摊时间有了$12-53\times$的提升。

Private Training

	LeNet (MNIST)		AlexNet (CIFAR-10)		VGG-16 (CIFAR-10)		AlexNet (TI)		VGG-16 (TI)	
	Time	Comm.	Time	Comm.	Time	Comm.	Time	Comm.	Time	Comm.
FALCON[*]	14.90	0.346	62.37	0.621	360.83[†]	1.78[†]	415.67	2.35	359.60[‡]	1.78[‡]
CRYPTGPU	2.21	1.14	2.91	1.37	12.14[†]	7.55[†]	11.30	6.98	13.89[‡]	7.59[‡]
Plaintext	0.0025	—	0.0049	—	0.0089	—	0.0099	—	0.0086	—

[*]The provided implementation of FALCON does not support computing the gradients for the output layer, so the FALCON measurements only include the time for computing the gradients for intermediate layers. All measurements for FALCON are taken *without* batch normalization.
[†]Using a smaller batch size of 32 images per iteration (due to GPU memory limitations). We make the same batch size adjustment for FALCON.
[‡]Using a smaller batch size of 8 images per iteration (due to GPU memory limitations). We make the same batch size adjustment for FALCON.

TABLE IV: Running time (in seconds) and total communication (in GB) for a single iteration of private training with a batch size of 128 images for different models, datasets, and systems in a LAN setting. The "TI" dataset refers to the Tiny ImageNet dataset [15]. The plaintext measurements correspond to the cost of training on plaintext data on the GPU. Performance numbers for FALCON are obtained by running its reference implementation [45] on three compute-optimized AWS instances in a LAN environment (see Section IV-B). As discussed in Section IV-A, when testing the performance of CRYPTGPU, we replace max pooling with average pooling in all of the networks.

如表4，对于模型训练，CryptGPU比FALCON减少时间开销达$7-36\times$。但是和GPU明文训练相比，还有$2000\times$差距。

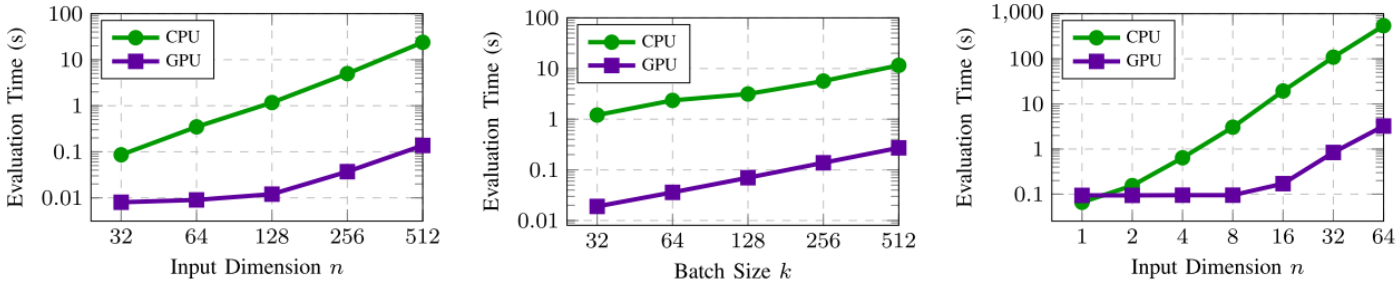
	Linear		Pooling		ReLU		Softmax	
	FALCON	CRYPTGPU	FALCON	CRYPTGPU	FALCON	CRYPTGPU	FALCON	CRYPTGPU
LeNet (MNIST)	13.07	0.49	1.34	0.076	0.47	1.00	—	0.53
AlexNet (CIFAR)	59.23	0.86	2.65	0.077	0.41	1.33	—	0.55
VGG-16 (CIFAR)[*]	355.16	6.33	2.86	0.21	5.40	4.74	—	0.53
AlexNet (TI)	402.45	5.60	10.20	0.37	1.92	4.16	—	1.04
VGG-16 (TI)[†]	355.84	7.61	2.87	0.32	5.37	4.73	—	0.98

[*]Using a smaller batch size 32 images per iteration (due to GPU memory limitations). We make the same batch size adjustment for FALCON.
[†]Using a smaller batch size 8 images per iteration (due to GPU memory limitations). We make the same batch size adjustment for FALCON.

TABLE V: Runtime (in seconds) of FALCON [6] and CRYPTGPU for evaluating the linear, pooling, ReLU, and softmax layers for different models and datasets during private *training*. The "linear" layers include the convolution and the fully-connected layers. The "pooling" layer refers to max pooling in FALCON, and average pooling in CRYPTGPU. The implementation of FALCON [45] does not currently support softmax evaluation (and correspondingly, gradient computation for the output layer). Performance numbers for FALCON are obtained by running its reference implementation [45] on three compute-optimized AWS instances in a LAN environment (see Section IV-B).

如表5，和FALCON相比，CryptGPU提升线性层的性能达到$25-70\times$。不过对于ReLU函数，FALCON中的方法和本文采用的A2B方法性能差距不大，而且对于小模型，FALCON中的方法更加高效。

Microbenchmarks



(a) Convolution on an $n \times n \times 3$ input with an 11×11 kernel, 64 output channels, 4×4 stride, and 2×2 padding.

(b) Convolution on batch of k $32 \times 32 \times 3$ inputs with an 11×11 kernel, 64 output channels, 4×4 stride, and 2×2 padding.

(c) Convolution on an $n \times n \times 512$ input with a 3×3 kernel, 512 output channels, 1×1 stride, and 1×1 padding.

Fig. 1: Comparison of total protocol execution time (in a LAN setting) for privately evaluating convolutions on the CPU and the GPU. Parameters for convolution kernels are chosen based on parameters in AlexNet [16]. The stride and padding parameters specify how the filter is applied to the input. All of the figures are log-log plots.

图1展示了对于不同的卷积和输入大小，GPU的方案大大优于CPU方案。大约有$10-174\times$的提升。

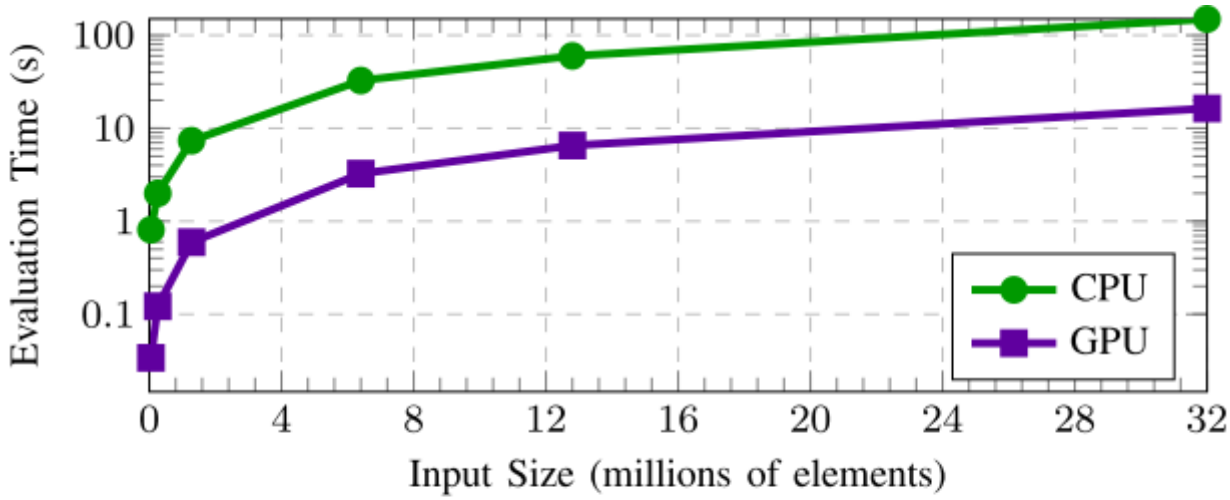


Fig. 2: Comparison of total protocol execution time (in a LAN setting) on the CPU vs. the GPU for point-wise evaluation of the private ReLU protocol on different-sized inputs.

如图2，对于ReLU函数，GPU也可以加速（同样的计算协议）。实验中大约有$9-16\times$的时间提升。

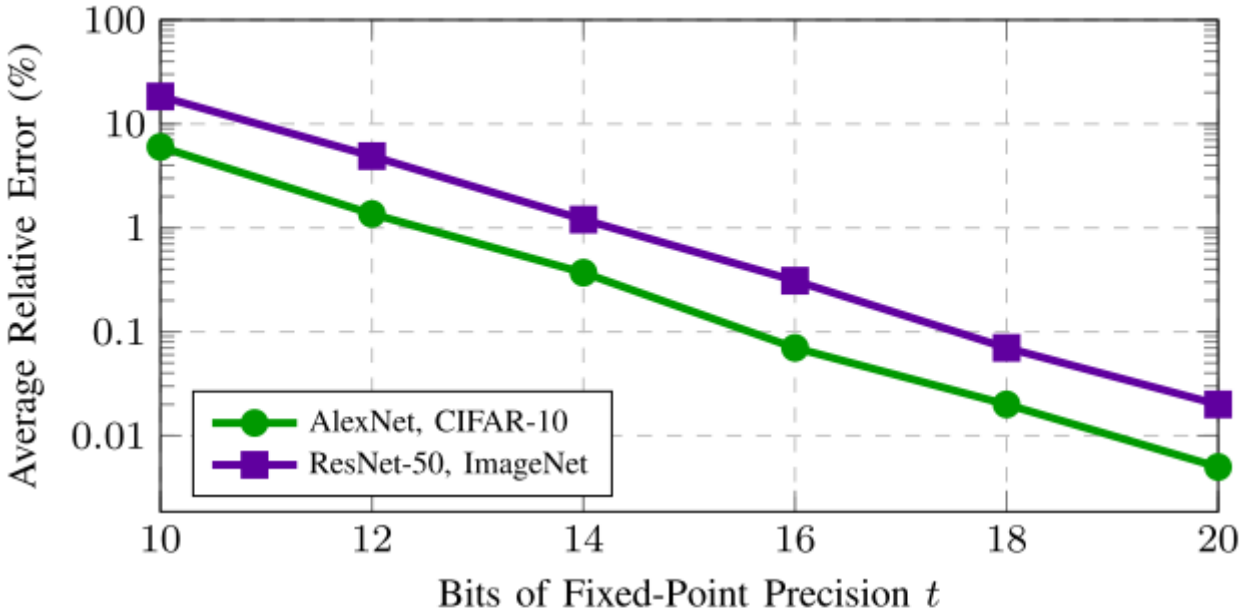


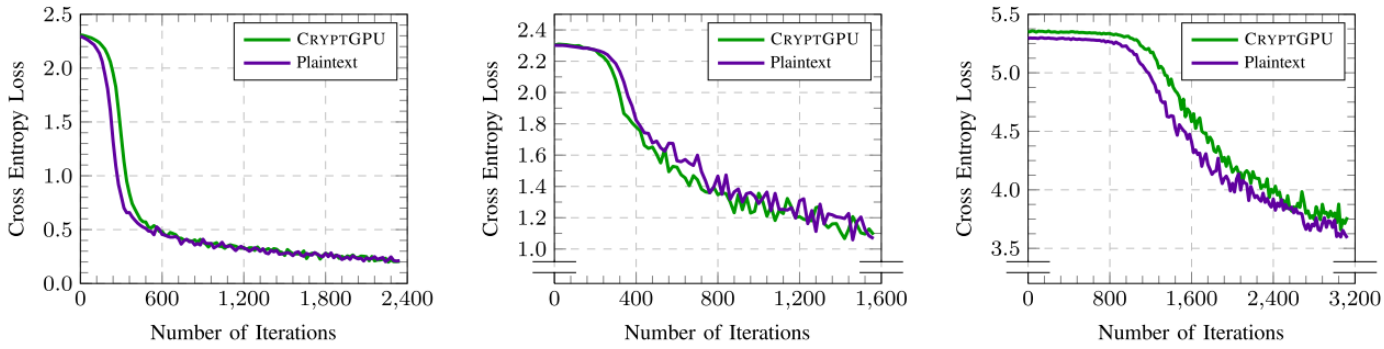
Fig. 3: Average relative error between the model outputs computed using the private inference protocol in CRYPTGPU with t bits of fixed-point precision (i.e., an integer $x \in \mathbb{R}$ is represented as the nearest integer to $x \cdot 2^t$) and the output computed using *plaintext* floating-point inference. Analysis based on evaluating AlexNet on CIFAR-10 and ResNet-50 on ImageNet, and averaged over 10 randomly-chosen instances.

图3是相对误差随着比特位数的变化，大约15比特的小数位就能达到1%的误差。

	ResNet-50	ResNet-101	ResNet-152
Average Relative Error	0.015%	0.020%	0.021%
Top-1 Acc. (CRYPTGPU)	78%	82%	79%
Top-1 Acc. (Plaintext)	78%	82%	79%
Top-5 Acc. (CRYPTGPU)	92%	90%	93%
Top-5 Acc. (Plaintext)	92%	90%	93%

TABLE VI: Comparison of outputs of CRYPTGPU's private inference protocol on ImageNet with the ResNet models with those of the plaintext algorithm (using PyTorch). The average relative error is computed between the outputs of the private inference protocol and those of the plaintext execution (on the same input). The Top-1 and Top-5 accuracies for both settings are computed based on the outputs of model inference with respect to the ground truth label. The measurements are taken over a random set of 100 examples drawn from the ImageNet validation set.

表6展示了在ResNet上的预测相对误差，不超过0.021%。



(a) LeNet on MNIST (trained for 5 epochs with a batch size of 128).

(b) AlexNet on CIFAR-10 (trained for 1 epoch with a batch size of 32).

(c) AlexNet on Tiny ImageNet (trained for 1 epoch on a batch size of 32).

Fig. 4: Moving average of the cross-entropy loss as a function of the number of training iterations using CRYPTGPU and using a plaintext protocol for different models and datasets. In each setting, we use the same initialization and learning rate (for stochastic gradient descent) for both private and plaintext training. For LeNet, we use a random initialization. For the AlexNet experiments, we use PyTorch's default AlexNet architecture [46] for both the plaintext training and the private training experiments (which is a variant of the standard AlexNet architecture described in [16]). We use PyTorch's pre-trained initialization for AlexNet as our initialization. The moving average is computed over a window of size 20 (i.e., the value reported for iteration i is the average of the cross entropy loss on iterations $i-10,\ldots,i+9$).

图4展示了训练交叉熵随着训练迭代的变化，CryptGPU和明文训练基本一致。

	Baseline	CrypTGPU	Plaintext
LeNet, MNIST*	10%	93.97%	93.34%
AlexNet, CIFAR-10†	10%	59.60%	59.77%
AlexNet, Tiny ImageNet‡	2%	17.82%	17.51%

*Trained for 5 epochs (2345 iterations) with a batch size of 128.
†Trained for 1 epoch (1563 iterations) with a batch size of 32.
‡Trained for 1 epoch (3125 iterations) with a batch size of 32.

TABLE VII: Validation set accuracy for different models trained using CrypTGPU and the plaintext training algorithm. For each configuration, both training approaches use the same initialization and learning rate (for stochastic gradient descent). For LeNet, we use a random initialization. For the AlexNet experiments, we use PyTorch's default AlexNet architecture [46] for both the plaintext training and the private training experiments. Here, we use PyTorch's pre-trained weights for AlexNet to speed up convergence. We also report the baseline accuracy for each configuration (i.e., accuracy of the "random-guess" algorithm). Note that training for more iterations will increase the accuracy; the intent of this comparison is to demonstrate a close similarity in model accuracies for the the model output by the private training protocol with the model output by plaintext training after a few thousand iterations of stochastic gradient descent.

表7是验证集准确率，差距小于1%。

	Max Pooling	Average Pooling
AlexNet	76.15%	73.35%
VGG-16	82.37%	83.17%

TABLE VIII: Validation set accuracy for plaintext training of AlexNet and VGG-16 over the CIFAR-10 dataset using max pooling vs. average pooling. All networks were trained using 50 epochs using a standard stochastic gradient descent (SGD) optimizer in PyTorch.

最后，验证了将Max-Pooling替换为Average-Pooling带来的准确率影响。对AlexNet带来了3%的降低，对VGG反而增加了1%的准确率。这表明替换并不会对模型性能带来显著降低。

Conclusion

本文使用的协议都是经典协议，但是却利用GPU实现了大幅度的性能提升。不过本文的实验都是在LAN下进行的，在WAN下通信带来的开销是否会显著降低这种提升很值得探索一下。最近还有其他的论文也在探索利用GPU等硬件加速安全协议，值得进一步的学习。