GForce: GPU-Friendly Oblivious and Rapid Neural Network Inference

Lucien K. L. Ng
Department of Information Engineering
The Chinese University of Hong Kong
Shatin, N.T., Hong Kong

Sherman S. M. Chow*
Department of Information Engineering
The Chinese University of Hong Kong
Shatin, N.T., Hong Kong

Abstract

Neural-network classification is getting more pervasive. It captures data of the subjects to be classified, *e.g.*, appearance for facial recognition, which is personal and often sensitive. Oblivious inference protects the data privacy of both the query and the model. However, it is not as fast and as accurate as its plaintext counterpart. A recent cryptographic solution Delphi (Usenix Security 2020) strives for low latency by using GPU on linear layers and replacing some non-linear units in the model at a price of accuracy. It can handle a query on CIFAR-100 with ∼68% accuracy in 14s or ∼66% accuracy in 2.6s.

We propose GForce, tackling the latency issue from the root causes instead of approximating non-linear computations. With the SWALP training approach (ICML 2019), we propose *stochastic rounding and truncation (SRT) layers*, which fuse quantization with dequantization between non-linear and linear layers and free us from floating-point operations for efficiency. They also ensure high accuracy while working over the severely-finite cryptographic field. We further propose a suite of GPU-friendly secure online/offline protocols for common operations, including comparison and wrap-around handling, which benefit *non-linear* layers, including our SRT.

With our two innovations, GForce supports VGG-16, attaining ∼73% accuracy over CIFAR-100 for the first time, in 0.4s. Compared with the prior best non-approximated solution (Usenix Security 2018), GForce speeds up non-linear layers in VGG by >34×. Our techniques shed light on a new direction that utilizes GPU throughout the model to minimize latency.

1 Introduction

Machine learning is becoming more prevalent. *Deep neural networks* (DNNs) achieved great success, notably in image recognition tasks with applications in surveillance or medical check. These applications often process sensitive or at least personal data. Clients can be reluctant to hand in their data to the model owner (or the server). Meanwhile, sending the model to the clients for evaluation is often impossible, not to say its financial and privacy implications.

Oblivious inference resolves this dilemma. The server with a deep neural network $\mathsf{DNN}(\cdot)$ can return the classification result $\mathsf{DNN}(x)$ to any client while remains oblivious about x without leaking its model $\mathsf{DNN}(\cdot)$. From the perspective of computation nature, a neural network can be divided into linear layers and non-linear layers. Cryptographic solutions often handle linear layers and non-linear layers separately, such as using *additive homomorphic encryption* (AHE) and *garbled circuits* (GC), respectively, but these tools impose high overheads. A recurrent research problem is *how to perform secure computations of non-linear functions efficiently*.

1.1 Two Open Challenges

We reckon GPU as a promising tool for reducing latency. It is highly-optimized for computing in parallel, accelerating DNN computation when compared with CPU, primarily on parallelizable linear operations. Delphi [18], the state-of-the-art cryptographic framework, utilizes GPU to accelerate linear layers but fails to benefit non-linear layers. Instead, Delphi encourages the training scheme to replace ReLU layers by their quadratic approximation (*i.e.*, x^2), which lowers the latency but still sacrifices accuracy. The non-linear computations, including those remaining ReLU layers and maxpool layers, are still handled by less efficient tools such as GC. Unfortunately, it is unclear how GC can leverage GPU parallelism.

Most (plaintext) neural networks (especially those with high accuracy) run over floating-point numbers ("floats") with large fluctuations in the model parameters (in magnitude represented by 256 bits). In contrast, cryptographic frameworks, utilizing primitives such as AHE, GC, and secret sharing, mostly handle values in a small magnitude (usually $20 \sim 40$ bits) range. Extending the *bit-width* inside the cryptographic tools for higher precision slows down all operations. Some recent works (*e.g.*, XONN [21]) adopt binarized neural networks with accuracy lower than the original one. The inherent tension between accuracy and efficiency remains.

*Supported by General Research Fund (CUHK 14210319) of UGC, HK.

1.2 Our Contributions

This paper tackles the latency versus accuracy issue from the root causes. Our framework, which we call GForce, is a new paradigm of oblivious inference based on specially-crafted cryptographic protocols and machine-learning advances.

On the machine-learning front, we formulate *stochastic rounding and truncation (SRT) layers*, making a quantization-aware training scheme SWALP [28] more compatible with (our) cryptographic tools. SWALP trains a DNN under a low-precision setting while keeping accuracy, but its extra processing introduces latency during oblivious inference. Our SRT layer serves as a "swiss-army knife," which contributes to reduced latency and communication complexity while keeping the intermediate values of DNN evaluation "small."

On the cryptography front, we propose a suite of GPU-friendly protocols for *both* linear layers and common *non-linear layers* to enjoy the power of GPU parallelism. It enables an elegant approach to oblivious inference, in contrast to existing approaches of switching between different cryptographic primitives (*e.g.*, arithmetic, boolean, and Yao's shares) across different layers (*e.g.*, three-non-colluding-servers approaches [27]) or customizing alternatives (*e.g.*, polynomial approximation [5] or replacement with square [18]).

High-accuracy Networks in the Low-precision Setting. To overcome the low-precision issue that bars our way to our high-accuracy goal, we adopt SWALP [28], a scheme to fit a neural network into the low-precision setting. It takes as inputs the DNN architecture, hyper-parameters, and training data and returns a trained DNN whose linear layers can run in a low-precision environment. SWALP reported that the accuracy loss due to the low-precision setting is $<$1pp [28].

While it sounds fitting our purpose exactly, making it secure and efficient is still not easy (see Section 2). SWALP requires (de)quantization for intermediate DNN results, which can be seen as truncation that confines the magnitude range to prevent overflow. Secure computation of the needed operations, especially *stochastic rounding*, is rarely explored. A recent work [27] explicitly mentioned that truncation is expensive.

To reduce computation and the complexity of individual cryptographic operation, we formulate SRT layers, which fuse dequantization, quantization, and stochastic rounding. Such formulation may inspire further improvement in the seamless integration of machine learning and cryptography.

As the tension between working in a limited plaintext space and not risking overflowing still exists, we also derive parameters for striking a balance under such an inherent trade-off.

GPU-Friendly Protocols for Non-linear Operations. It is unclear how we can leverage GPU for non-linear layers. For the first time, we propose a suite of GPU-friendly protocols for primitive operations in popular non-linear layers and our newly formulated stochastic rounding and truncation layers.

Secure comparison is a core functionality necessary for computing ReLU (approximated by Delphi) and maxpool layers (failed to be optimized by Delphi). Existing secure comparison protocols involve computations that fail to leverage the power of GPU. Our technical contribution here is a semi-generic approach that transforms AHE-centric protocols to their functionally-equivalent GPU-friendly version. We call it our SOS trick (see Section 3.3), which stands for secure online/offline share computation. Our protocols have a lower online communication cost than their GC-based counterparts. Moreover, to twist the performance to the extreme, we design our protocols with the precision constraints of cryptographic tools and GPUs in mind. We also need to develop GPU-friendly protocols for truncation and wrap-around handling to enable GForce to run in low-precision without error.

All our protocols do not require any approximation. Using them over a DNN can attain its original accuracy in the (low-precision) plaintext setting. Concretely, when compared with prior works that also avoid approximating ReLU units (Gazelle [11] and Falcon [13]), GForce is at least 27$\times$ faster when handling a large number (2^{17}) of inputs (see Table 4 in Section 4.1). As a highlight, for a CIFAR-100 recognition task (see Section 4.2), GForce attains 72.84% accuracy with 0.4s. (The prior best result by Delph handles a query in 14.2s with 67.81% accuracy or 2.6s with 65.77% accuracy.)

To summarize, we make the following contributions.

1) We complement quantization-aware training with our stochastic rounding and truncation layers that normalize intermediate results and reduce computational and communication complexities throughout the model while keeping accuracy.

2) We propose a suite of protocols for non-linear operations, which exploits GPU parallelism and reduces latency.

3) We implement our framework and demonstrated our superior accuracy and efficiency, notably, even over the state-of-the-art approach of using three non-colluding servers [27].

4) Technical insights in GForce, *e.g.*, SWALP adoption, SRT layer, and GPU-friendly secure protocols for (non-)linear layers, can benefit some existing and future frameworks.

2 Technical Overview

GForce is an online/offline GPU/CPU design. In the offline phase when the query is unknown, some precomputation is done without knowing the actual query. Upon receiving a (private) query in the online phase, we ask the GPU to quickly perform"masked" linear computation (in batch) online, *even for non-linear layers*. All our cryptographic protocols share this core feature. In particular, GForce only precomputes the relatively costly AHE-related operations offline. Online computations use the much more efficient *additive secret sharing* (SS), which provides the masking we need. Both AHE and SS operate over fixed-point numbers in $\mathbb{Z}_q$.

2.1 Issues in using GPU for Cryptography

Low-precision Setting in GPU. GPU is optimized for 32-bit and 64-bit floats while supporting 24-bit and 52-bit integer arithmetic operations, respectively. Overflowing (on GPU's integer part) will lead to precision loss or even trash the values represented in floats. We need to ensure the value being secret-shared does not exceed 52 bits after each GPU addition and multiplication. It only leaves us $\sim$20 bits as the plaintext space, which we call *bit-width*, denoted by ℓ.

Furthermore, secure protocols, including those we propose, have computation and communication costs of at least $\Omega(\ell)$. Running under less bit-width is vital for performance.

Quantization to the Low-precision Setting. DNN operations are mostly over floats. A careful quantization is needed to store them in fixed points; otherwise, they may overflow when they are too large or become 0 when they are too small.

2.2 GPU-Friendly Secure Comparison

GForce focuses on leveraging GPU for comparison, which is a crucial operation in *non-linear layers*, including ReLU and maxpool in many popular neural networks (*e.g.*, [10, 22]). These non-linear layers can be securely computed via *the secure comparison protocol of Damgård–Geisler–Krøigaard* (DGK protocol) [7]; however, it heavily relies on AHE and other non-linear operations that are still inefficient over GPU.

A novel component of GForce is its GPU-friendly secure comparison protocol, which we built by first decomposing the original DGK protocol into a bunch of linear operations and inexpensive non-linear operations, *e.g.*, bit-decomposition on plaintexts. We also prove that, as long as the values in those linear operations are not leaked, those non-linear operations are safe to perform without protection. We can then adopt the online/offline GPU/CPU paradigm to speed up all layers.

2.3 Issues in Oblivious Inference with SWALP

To run neural networks over a low bit-width finite field for high performance while maintaining accuracy, we use *Stochastic Weight Averaging in Low-Precision Training* (SWALP) [28] for linear layers. Intuitively, as SWALP trains a DNN under low bit-width integers, its trained parameters and hence its accuracy are optimized for fixed-point integers.

Using SWALP within a cryptographic framework poses several challenges. Specifically, the (de)quantization scales up/down and rounds up the values according to the maximum magnitude among all input values. A direct adoption requires the rather inefficient secure computation of maximum, rounding, and division. Furthermore, we still need to dequantize the output of linear layers before feeding it back to the non-linear layers (the second row of Figure 1); this would bring us back to securely computing over floating-point numbers.

GC (§1, §7) AHE (§1, §3.1) SS (§2, §3.1)	Garbled Circuit (not used by GForce) Additive Homomorphic Encryption Secret Sharing
DGK (§2.2, §3.4)	Damgård *et al.*'s secure comparison
SWALP (§1, §2.4, §3.7)	Stochastic Weight Averaging in Low-Precision Training

Table 1: Acronyms for Existing Concepts

2.4 Stochastic Rounding and Truncation

Precomputing Maximum Magnitudes. Instead of finding the maximum, we employ the heuristics (Section 3.7.1) of gathering statistics from training data to estimate for the queries, which fixes the required parameters in advance. Only a few bits of information (per layer) need to be shared with the client for (de)quantization (more in Section 5.1).

Fusing (De)Quantization. We observe that we can bring forward the dequantization before comparison-based non-linear layers (*e.g.*, ReLU and maxpool) to be after those non-linear layers, resulting in fusing dequantization with quantization (as Figure 1 illustrates). We prove (in Section 3.7.2) that the resulting computation is equivalent. Such fusion allows us to handle the values throughout all layers in a fixed-point, low-bit-width representation. Thus, it reduces the number of (now fused) cryptographic operations and the complexity for each of them while avoiding overflow or underflow.

The fused (de)quantization may become scaling up or down depending on the dataset and DNN architecture. In our experiment (mainly over VGG DNN [22]), it is always scaling down. As we always scale down by a power of 2 as in bit-truncation, we call it *stochastic rounding and truncation* layer.

Rounding Efficiently while Avoiding Truncation Error. Truncating the least-significant bits of additive SS (used in prior works for scaling down, *e.g.*, [18]) may incur errors trashing the values when wrap-around occurs (see Section 3.7.3). More specifically, reducing 1 bit of the plaintext space doubles the error probability. In the low-precision setting, such errors are very likely. To balance off such value-trashing error (if it exists), we introduce a GPU-friendly *wrap-around handling* protocol. However, even after fixing this error, an off-by-one error in truncation may still happen. We observe that the error distribution due to off-by-one error is very close to that of stochastic rounding (which we prove in Section 3.7.3). Our truncation protocol then exploits that for the effect of *stochastic rounding*, the rounding method specified by SWALP, on the scaled-down results, killing two birds with one stone.

Putting the quantization, dequantization, and stochastic rounding altogether, we establish the SRT layers, in which we consider the scaling down for (de)quantization as truncation.

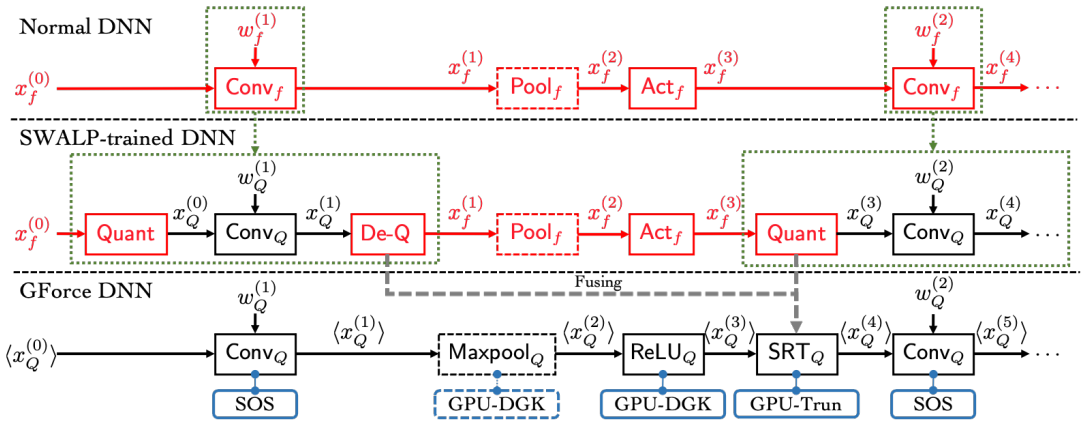


Figure 1: Adopting SWALP (in Green) and Overcoming the Hard Parts (in Red) for Crypto Tools via SRT Layers and Our Protocols (in Blue) (Conv: Convolution, Quant: Quantization, De-Q: De-Quantization, (Max)Pool: (Max-)Pooling, Act: Activation, ReLU: Rectified Linear Unit)

3 GPU-Friendly Oblivious Computation

3.1 Cryptographic Toolbox and Notations

Additive Homomorphic Encryption (AHE). AHE is (public-key) encryption that features additive homomorphism, *i.e.*, $[x+y] = [x]+[y]$, where $[m]$ denotes a ciphertext of m. One can also multiply $[x]$ with a plaintext m, *i.e.*, $[mx] = m\cdot[x]$. Homomorphic operations can be fused into a linear function $f([m]) := \mathsf{mult}\cdot[m]+\mathsf{bias}$ over vectors/matrices/tensors $m \in (\mathbb{Z}_p)^n$, where $[m] = ([m_0],[m_1],\ldots,[m_{n-1}])$ and f can output multiple values. We use AHE_q (or simply AHE) and AHE_p to denote an AHE scheme over $\mathbb{Z}_q$ and $\mathbb{Z}_p$, respectively. We mostly omit the field size, *e.g.*, as in $[m]$ instead of $[m]_p$.

AHE is supposed to have *circuit privacy*, *i.e.*, with $[m]$ and sk, one cannot learn mult and bias from $\mathsf{ct} = \mathsf{mult}\cdot[m]+\mathsf{bias}$.

Additive Secret Sharing. A client C can secret-share its private $x \in \mathbb{Z}_q$ to a server S by randomly picking $r^{\mathsf{S}} \in \mathbb{Z}_q$, sending it to S, and keeping $(x - r^{\mathsf{S}})$ mod q locally. Either share alone has no information about x. We let $\langle x\rangle_q^{\mathsf{S}}, \langle x\rangle_q^{\mathsf{C}} \in \mathbb{Z}_q$ be the shares of x held by S and C, respectively. For brevity, we use the notation of $\langle x\rangle = \{\langle x\rangle^{\mathsf{S}}, \langle x\rangle^{\mathsf{C}}\}$ to denote both shares, and omit the underlying field when it is clear. When the field size should be emphasized (for both the secret share and its ciphertext), we may run into notation such as $[\langle\beta\rangle_p^{\mathsf{C}}]_p$.

S and C can jointly compute secret shares of $c = a\cdot b$ using Beaver's trick [3] (Protocol 8) if they had $\langle u\rangle, \langle v\rangle$, and $\langle z\rangle$ s.t. $u\cdot v = z$. The core idea is to first reconstruct $\mu = u-a$ and $\nu = v-b$, then the shares are $\langle z\rangle^i - \mu\langle v\rangle^i - \nu\langle u\rangle^i + i\mu\nu$, where $i \in \{0,1\}$ represents $\{\mathsf{S},\mathsf{C}\}$. Operating over secret shares is very efficient on GPU and incurs less overhead than AHE. It can be generalized to matrix operations and tensor convolutions.

Additive SS has a near-to-plaintext performance for addition and plaintext-SS multiplication ($c\cdot\langle x\rangle = \langle c\cdot x\rangle$). Vectorizing these operations using GPU, which is extensively done by GForce, hugely outperforms their counterparts using AHE.

3.2 Overview of GForce

In supervised learning, every training data is a data point $\mathbf{x}$ associated with a label $\mathbf{y}$. A DNN tries to learn the relationship between $\mathbf{x}$ and $\mathbf{y}$. Inference outputs a label $\mathbf{y}$ of query $\mathbf{x}$.

GForce allows a server S with a DNN model $\mathsf{DNN}(\cdot)$ to provide oblivious inference. It returns $\mathsf{DNN}(x)$ to client C without knowing the client query x and $\mathsf{DNN}(x)$. Meanwhile, C remains oblivious to the learnable parameters of DNN.

Most DNNs consist of many linear and non-linear layers. In GForce, each layer i outputs additive SS $\langle x^{(i)}\rangle$ to the server and the client, which in turn acts as the input to the next layer.

For linear layers, GForce supports fully-connected layers, which multiply the input by a learnable weighting matrix, and convolution layers, which convolute learnable kernels over the input. Secure computation of linear function is typically done via the homomorphism of AHE (reviewed in Section 3.1). We propose AHE-to-SOS transformation (in Section 3.3), which transforms the traditional AHE-based approach into our GPU-friendly linear computation protocol over secret shares.

For non-linear layers, we focus on comparison as a core operation. We propose GPU-friendly secure comparison protocols (in Section 3.4) built on top of DGK protocols [7], with any wrap-around error fixed (in Section 3.5). GForce thus supports the most common choices of activation and pooling layers, *i.e.*, ReLU and maxpool (in Section 3.6), respectively.

GForce also specifically considers SWALP-trained DNNs embodied by the SRT layers (in Section 3.7), which efficiently divide and wrap around the inputs in additive SS, whose resulting value distribution is close to stochastic rounding.

To summarize, C produces an additive SS of its query $\langle x^{(0)}\rangle$. C and S then sequentially invoke our protocols according to the architecture of DNN over their additive SS $\{\langle x^{(i)}\rangle\}$, and eventually, C recovers $\mathsf{DNN}(x)$ from the additive SS of the last layer. Tables 1-2 list the (existing and new) building blocks.

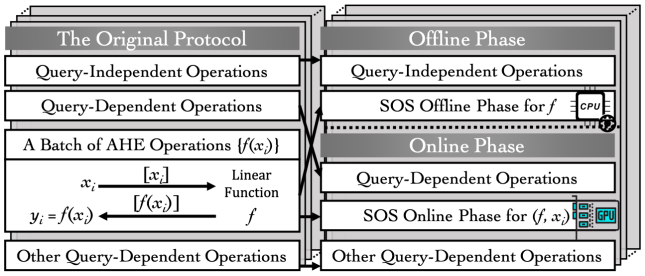


Figure 2: Our AHE-to-SOS Transformation for Crypto Protocols

SOS (§3.3)	Secure On/off Share Computation
SC-DGK (§3.4)	Share-Computation variant of DGK
GPU-DGK / -Wrap $^{\text{off/on}}$ (§3.4/§3.5)	GPU-friendly DGK or Wrap protocol and its offline or online sub-protocol
SRT layer (§1, §2.4, §3.7)	Stochastic Rounding and Truncation tailored for SWALP-trained DNN

Table 2: Acronyms for New Concepts in GForce

3.3 Secure Online/Offline Share Computation

One of our core ideas is to replace the online computation over AHE ciphertexts of the query with the offline computation over AHE ciphertexts of some *query-independent* randomness and the (fast) computation over secret shares of the query. Table 3 lists the notations for describing our protocols.

AHE-to-SOS Transformation. An AHE-based protocol (Figure 2) starts by C sending an encrypted value $[x]$ to S. S then applies its private linear function f on $[x]$ and returns the result to C. Figure 3 describes the resulting protocol obtained after AHE-to-SOS transformation. We call this trick secure online/offline share computation (SOS).[1] As our most basic usage of AHE, our protocol in Figure 3 is also named SOS.

In the offline phase, C randomly picks r^{C} and encrypts it to S. S then applies f over this AHE ciphertext [11,13], masks it with r^{S}, and sends the results back to C. C decrypts it and keeps the result as an output share r^{C} for the online phase.

GForce leverages the linearity[2] $f(\chi) = f(\chi - r) + f(r)$ to protect χ. In the online phase, S and C each hold an *input share*, $\langle\chi\rangle^{\mathsf{S}}$ and $\langle\chi\rangle^{\mathsf{C}}$. C additively masks its input share with r^{C} and sends it to S. S reconstructs another additive SS $(\chi - r^{\mathsf{C}})$ and computes $\langle f(\chi)\rangle^{\mathsf{S}} := f(\chi - r^{\mathsf{C}}) - r^{\mathsf{S}}$ on GPU. $\langle f(\chi)\rangle^{\mathsf{S}}$ and $\langle f(\chi)\rangle^{\mathsf{C}} := f(r^{\mathsf{C}}) + r^{\mathsf{S}}$ are the *output shares*. Note that $\langle f(\chi)\rangle^{\mathsf{S}} + \langle f(\chi)\rangle^{\mathsf{C}} = f(\chi)$.

[1] The naming of our (secret) shares may be "abused" in some sense, *e.g.*, an "output share" can be created even before knowing the output because one can create the corresponding share that matches with it when the output is known in a later time. For example, in our SOS, the client has $\langle f(\chi)\rangle^{\mathsf{C}} := f(r^{\mathsf{C}}) + r^{\mathsf{S}}$ in SOS's offline phase even though $f(\chi)$ is unknown.

[2] Slalom [24] precomputes $f(r)$ in $f(\chi) = f(\chi - r) + f(r)$ within the trusted environment. Here, we precompute $f(r)$ with AHE.

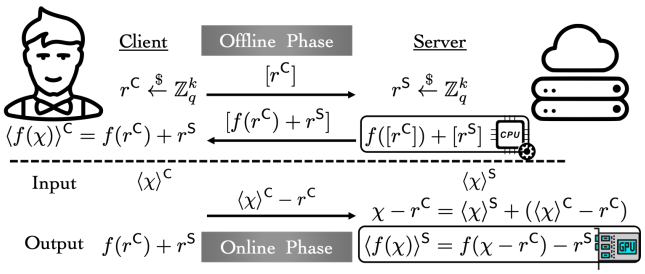


Figure 3: Our Secure Online/Offline Share Computation (SOS) for Linear Functions: $[\cdot]$ is an AHE ciphertext. $\langle\cdot\rangle$ is an additive SS.

SOS reduces the online computation time (of using AHE). The transformed protocol processes a batch of inputs in additive SS to fully utilize GPU's batch-processing performance. Using SS instead also reduces the online communication.

Applications. To apply SOS (Figure 3), S needs to know f, including its internal parameters, in the offline phase. This requirement is trivial for linear layers, such as convolution and fully-connected layers, because S knows the weight.

Beyond linear layers, we also apply the SOS trick to our other protocols that use AHE, *e.g.*, DGK for comparison. For these protocols, the internal parameters of f are usually secret random values generated by S, which we can somehow move to the offline phase, as Sections 3.4 and 3.5 will show.

3.4 GPU-Friendly Secure Comparison

In the DGK protocol [7] (Protocol 5), the server S and the client C hold private integers $\alpha_{\ell-1}\cdots\alpha_1\alpha_0$ and $\beta_{\ell-1}\cdots\beta_1\beta_0$ respectively. It processes from $\ell-1$ to 0 to locate the first differing bit via computing b_i, which is 0 iff $(\alpha_j = \beta_j)_{\forall j: i<j<l}$ and $\alpha_i \neq \beta_i$. For that, C sends all $[\beta_i]$ to S. S then computes

$$[b_i]_{\forall i\in\{\ell-1,\ldots,0\}} = [a] + ([\alpha_i] - [\beta_i]) + 3\sum_{j=i+1}^{\ell-1}[\alpha_j \oplus \beta_j] \quad (1)$$

with $a = 1 - 2\delta^{\mathsf{S}}$ and a random *bit* δ^{S} picked by S *offline*. To test also if $\alpha = \beta$, S computes $[b_{-1}] = [\delta^{\mathsf{S}}] + \sum_{j=0}^{\ell-1}[\alpha_j \oplus \beta_j]$. S can compute $[\alpha_j \oplus \beta_j]$ via AHE: $(1 - 2\cdot\alpha_j)\cdot[\beta_j] + [\alpha_j]$.

S sends $\{[b_i]\}$ back to C after shuffling their orders and multiplying each of them by a different random number $r^{\mathsf{S}}_{\times,i}$. With the decryption key, C sets $\delta^{\mathsf{C}} := 1 \in \mathbb{Z}_2$ if any ciphertext decrypts to 0; 0 otherwise, where $\delta^{\mathsf{S}} \oplus \delta^{\mathsf{C}} = (\alpha \le \beta)$.

Removing AHE from (Online Phase of) DGK Protocol. We assume the server knows α offline at the moment. When the server picks the randomness (*e.g.*, a) offline, we can re-write the multiplication of Equation 1 with $r^{\mathsf{S}}_{\times,i}$ as follow, which is for applying our AHE-to-SOS trick over DGK: $f^{\text{SC-DGK}}_{i,a,\alpha,r^{\mathsf{S}}_{\times,i}}(\beta) = r^{\mathsf{S}}_{\times,i}\cdot(a + \alpha_i - \beta_i + 3\cdot f^{\oplus}_{i,\alpha}(\beta))$, where $f^{\oplus}_{i,\alpha}(\beta) =$

ℓ (§2.1, §3, §4)	Bit-width of the DNN's data
$\mathbb{Z}_q$ (§3,§4)	Finite field for the DNN's data
$\mathbb{Z}_p$ (§3,§4)	Finite field for result bits $\{b_i\}$ (Eq. 1)
$[x]_q$ or $[x]$ (§3)	AHE ciphertext of x under $\mathbb{Z}_q$
$\langle x\rangle_q^{\mathsf{S}}$ / $\langle x\rangle_q^{\mathsf{C}}$ (§3)	SS of x under $\mathbb{Z}_q$ held by S or C
k (§3.4)	Number of inputs in a batch
α / β (§3.4)	SC-DGK's Server or Client input
ϕ_i (§3.4)	$\alpha_i \oplus \beta_i$ (i-th bit of α or β)
d (§3.7)	Divisor of an SRT layer
v_d (§3.7.3, App. B)	v mod d for $v \in \{q, \tau, s\}$
τ (§3.4, §3.5, §3.7)	Additive mask for the shared input s
z (§3.4, §3.5, §3.7)	$s+\tau$ mod q in GPU-DGK or -Trun
wrap (§3.4, §3.7.3)	Value that offsets wrapped-around z

Table 3: Notations (and where are they mostly discussed)

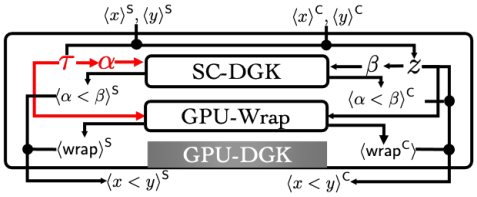


Figure 4: GPU-DGK prepares τ and α offline to enable efficient SOS computations of SC-DGK and GPU-Wrap.

$((1-2\alpha_i)\cdot\beta_i+\alpha_i)+f^{\oplus}_{i+1,\alpha}(\beta)$ if $i\neq\ell$, and $f^{\oplus}_{\ell,\alpha}(\cdot)=0$. The equivalence follows from $\alpha_i\oplus\beta_i=(1-2\cdot\alpha_i)\cdot\beta_i+\alpha_i$. The AHE-to-SOS transformation of DGK using the above (recursive) linear function (corresponding to Lines 6 to 11 and Lines 20 to 21) results in our Protocol 1, named SC-DGK for *share-based computation*, with ϕ_i denotes the output of $f^{\oplus}_{i,\alpha}$.

Protocol 1 processes a batch of k inputs, which we just denote any operand or result related to each of them as a single variable (*e.g.*, δ or P but not set/vector notation with subscript $\boldsymbol{\delta}_j$ or $\{\mathsf{P}_j\}$) to avoid running into double subscripts (*e.g.*, we need to break input β into its bit-representation). Looking ahead, Protocols 2, 3, and 4 also work on batches.

GPU-Friendly Secure Comparison. Beyond requiring an offline-known α, SC-DGK has two drawbacks. First, both α and β have to be non-negative, while the inputs to comparison-based layers can be negative. Second, the inputs need to be known to either S or C. GForce cannot use it to process any (intermediate) value protected by additive SS.

Inspired by the protocol of Veugen [26], our new protocol GPU-DGK (Protocol 2) can accept additive secret shares of *probably negative* input x and y from S and C, without assuming any online input is known in the offline phase. GPU-DGK reduces the comparison of $x\le y$ to that of $\alpha\le\beta$ in SC-DGK.

As illustrated in Figure 4, in GPU-DGK, S picks $\tau\in\mathbb{Z}_q$ and sets $\alpha=\tau \bmod 2^\ell$ offline. In the online phase, S and C got $\langle x\rangle$ and $\langle y\rangle$. S masks $\langle y-x\rangle^{\mathsf{S}}$ by τ and sends $\langle z\rangle^{\mathsf{S}}=\langle y\rangle^{\mathsf{S}}-\langle x\rangle^{\mathsf{S}}+\tau+2^\ell$. C obtains $\beta=z \bmod 2^\ell$ where $z=\langle z\rangle^{\mathsf{S}}+\langle y\rangle^{\mathsf{C}}-\langle x\rangle^{\mathsf{C}}$.

S and C then execute SC-DGK$^{\mathsf{on}}(\alpha,\beta)$ to compute $\langle\alpha\le\beta\rangle$. It indirectly compares $x\le y$ since $\alpha\le\beta$ equals to $\tau\le(y-x+\tau+2^\ell) \bmod 2^\ell$, we have $\alpha\le\beta \iff x\le y \bmod 2^\ell$.

Note that z may wrap around (due to a large τ). Our solution is to add a value wrap (also in shares) to the output, to be explained in Section 3.5. When z does not wrap around, the correctness of GPU-DGK can be derived similarly as an existing proof [25, Protocol 3]. Appendix B proves its correctness.

Protocol 1 Share-Computation Variant of DGK for Offline α

Offline Input (S|C)	$0\le\alpha<2^\ell$	sk_{AHE_q}, sk_{AHE_p}
Online Input (S|C)		$0\le\beta<2^\ell$
Output (S|C)	$\langle\alpha\le\beta\rangle_q^{\mathsf{S}}$	$\langle\alpha\le\beta\rangle_q^{\mathsf{C}}$
Constraints	k many (α,β) are processed together, $\ell\le\lfloor\log_2(q)\rfloor-2$, $q\equiv1 \bmod 2^\ell$	

1: **procedure** SC-DGK$^{\mathsf{off}}(\alpha,\emptyset)$
2: S decomposes $\alpha_{\ell-1}\cdots\alpha_0\leftarrow\alpha$ and sets $\alpha_{-1}\leftarrow0$
3: S: $\langle\alpha\le\beta\rangle_q^{\mathsf{S}}\leftarrow1\oplus\delta^{\mathsf{S}}$, $a\leftarrow1-2\cdot\delta^{\mathsf{S}}$, where $\delta^{\mathsf{S}}\in\mathbb{Z}_q^k$
4: **for** $i\leftarrow\{\ell-1,\ldots,-1\}$ **do**
5: C: picks $\langle\beta_i\rangle_p^{\mathsf{C}}\in\mathbb{Z}_p^k$ and sends $[\langle\beta_i\rangle_p^{\mathsf{C}}]_p$ to S
6: S: $\mathsf{mult}_{\alpha_i}\leftarrow1-2\cdot\alpha_i$, $\mathsf{bias}_{\alpha_i}\leftarrow\alpha_i$
7: S: $[\langle\phi_i\rangle_p^{\mathsf{C}}]_p\leftarrow\mathsf{mult}_{\alpha_i}\cdot[\langle\beta_i\rangle_p^{\mathsf{C}}]_p+[\mathsf{bias}_{\alpha_i}]_p$
8: S: generates random $r^{\mathsf{S}}_{\times,i}\in(\mathbb{Z}_p^*)^k$, $r^{\mathsf{S}}_{+,i}\in\mathbb{Z}_p^k$
9: S: $a\leftarrow\delta^{\mathsf{S}}$ if $i=-1$
10: S: $t_i\leftarrow[a+\alpha_i]_p-[\langle\beta_i\rangle_p^{\mathsf{C}}]_p+3\cdot\sum_{j=i+1}^{\ell-1}[\langle\phi_j\rangle_p^{\mathsf{C}}]_p$
11: S: $[\langle b_i\rangle_p^{\mathsf{C}}]_p\leftarrow r^{\mathsf{S}}_{\times,i}\cdot t_i+[r^{\mathsf{S}}_{+,i}]_p$
12: S: picks k permutations P of $\{-1,0,1,\ldots\ell-1\}$
13: S: shuffles all k $[\langle b_i\rangle_p^{\mathsf{C}}]_p$ by P and sends them to C
14: C: decrypts $[\langle b_i\rangle_p^{\mathsf{C}}]_p$ to get $\langle b_i\rangle_p^{\mathsf{C}}$ for $i\in[-1,\ell-1]$
15: S, C stores all their own values in $\mathsf{pre}^{\mathsf{S}}$ or $\mathsf{pre}^{\mathsf{C}}$, resp.
16: **procedure** SC-DGK$^{\mathsf{on}}(\mathsf{pre}^{\mathsf{S}},(\mathsf{pre}^{\mathsf{C}},\beta))$
17: C decomposes $\beta_{\ell-1}\cdots\beta_0\leftarrow\beta$ and sets $\beta_{-1}\leftarrow0$
18: **for** $i\leftarrow\{\ell-1,\ldots,-1\}$ **do**
19: C sends $\langle\beta_i\rangle_p^{\mathsf{S}}\leftarrow\beta_i-\langle\beta_i\rangle^{\mathsf{C}}$ to S
20: S: $\langle\phi_i\rangle_p^{\mathsf{S}}\leftarrow\mathsf{mult}_{\alpha_i}\cdot\langle\beta_i\rangle_p^{\mathsf{S}}+\mathsf{bias}_{\alpha_i}$
21: S: $\langle b_i\rangle_p^{\mathsf{S}}\leftarrow r^{\mathsf{S}}_{\times,i}\cdot(-\langle\beta_i\rangle_p^{\mathsf{S}}+3\cdot\sum_{j=i+1}^{\ell-1}\langle\phi_j\rangle_p^{\mathsf{S}})-r^{\mathsf{S}}_{+,i}$
22: S: shuffles all k $\langle b_i\rangle_p^{\mathsf{S}}$ by P and sends them to C
23: C: $\langle\alpha\le\beta\rangle_q^{\mathsf{C}}\leftarrow1$ if any recovered b_i is 0; otherwise 0.

Data Types for GPU-Friendly Protocols. We use the 53-bit significand plus a sign bit of 64-bit floating-point numbers. For not overflowing the result, $q^2n<2^{53}$, where n is the number of addition. As k is unknown before a network is given, we left some safety margin and set $\log_2(q)<23$. We thus use 32-bit floats that also minimize the communication cost.

Communication Cost. We transfer our additive SS in $\mathbb{Z}_p$ via 32-bit floats (which could be optimized to a 17-bit transfer).

Protocol 2 Our GPU-friendly Secure Comparison Protocol

Offline Input (S|C)	pk_{AHE_q}	sk_{AHE_q}
Online Input (S|C)	$\langle x\rangle^{\mathsf{S}}$, $\langle y\rangle^{\mathsf{S}}$	$\langle x\rangle^{\mathsf{C}}$, $\langle y\rangle^{\mathsf{C}}$
Output (S|C)	$\langle x \le y\rangle_q^{\mathsf{S}}$	$\langle x \le y\rangle_q^{\mathsf{C}}$
Constraints	$\log_2(p), \log_2(q) < 23$ $\ell \le \lfloor \log_2(q) \rfloor - 2$, $q \equiv 1 \bmod 2^\ell$	

1: **procedure** GPU-DGK$^{\mathsf{off}}$
2: S randomly picks $\tau \in \mathbb{Z}_q^k$ and computes $\alpha \leftarrow \tau \bmod 2^\ell$
3: S and C run SC-DGK$^{\mathsf{off}}(\alpha)$ and GPU-Wrap$^{\mathsf{off}}(\tau)$
4: S, C has every values stored in $\mathsf{pre}^{\mathsf{S}}$ or $\mathsf{pre}^{\mathsf{C}}$, resp.
5: **procedure** GPU-DGK$^{\mathsf{on}}(\{\mathsf{pre}^{\mathsf{role}}, \langle x\rangle^{\mathsf{role}}, \langle y\rangle^{\mathsf{role}}\}_{\mathsf{role}\in\{\mathsf{S},\mathsf{C}\}})$
6: S sends $\langle z\rangle^{\mathsf{S}} \leftarrow \langle y\rangle^{\mathsf{S}} - \langle x\rangle^{\mathsf{S}} + 2^\ell + \tau$ to C
7: C recovers $z \leftarrow \langle y\rangle^{\mathsf{C}} - \langle x\rangle^{\mathsf{C}} + \langle z\rangle^{\mathsf{S}}$, sets $\beta \leftarrow z \bmod 2^\ell$
8: S and C run SC-DGK$^{\mathsf{on}}(\mathsf{pre}^{\mathsf{S}}, (\mathsf{pre}^{\mathsf{C}}, \beta))$ to get share $\langle \alpha \le \beta\rangle_q^{\mathsf{S}}$ and $\langle \alpha \le \beta\rangle_q^{\mathsf{C}}$, resp.
9: S and C run GPU-Wrap$^{\mathsf{on}}(\mathsf{pre}^{\mathsf{S}}, (\mathsf{pre}^{\mathsf{C}}, z))$ to get share $\langle \mathsf{wrap}\rangle_q^{\mathsf{S}}$ and $\langle \mathsf{wrap}\rangle_q^{\mathsf{C}}$, resp.
10: S: $\langle x \le y\rangle_q^{\mathsf{S}} \leftarrow -\lfloor \tau/2^\ell \rfloor - (1 - \langle \alpha \le \beta\rangle_q^{\mathsf{S}}) + \langle \mathsf{wrap}\rangle_q^{\mathsf{S}}$
11: C: $\langle x \le y\rangle_q^{\mathsf{C}} \leftarrow \lfloor z/2^\ell \rfloor + \langle \alpha \le \beta\rangle_q^{\mathsf{C}} + \langle \mathsf{wrap}\rangle_q^{\mathsf{C}}$

For ℓ-bit inputs, our protocol transfers $64\ell + 112$ bits in the online phase, while a GC approach takes at least 384ℓ (for oblivious transfers). For instance, for a plaintext size of $\ell = 20$, we can reduce the online communication cost by 81.8%.

3.5 GPU-Friendly Wrap-Around Protocol

In the finite field $\mathbb{Z}_q$ over which our protocols mostly operate, we need to deal with the wrap-around issue, *i.e.*, for a secret s, its additively masked value $z = s + \tau \bmod q$ may equal to $s + \tau - q$ because $s + \tau > q$. Our protocol's output usually involves an additional z/d term, *e.g.*, $z/2^\ell$ in Line 11 of Protocol 2, where $d < q$ is a public divisor. To ensure correctness, we need to offset the $-q/d$ term as if wrap-around does not happen.

We propose GPU-Wrap (Protocol 3), our GPU-friendly wrap-around handling protocol, to produce the shares $\langle \mathsf{wrap}\rangle$ that can offset $-q/d$. Namely, we want $z/d - \tau/d + \mathsf{wrap} \approx s/d$. As observed by Veugen [25], we can assume $s < 2^{\ell+1} < (q-1)/2$ is always in the "first half" of $[0, q-1]$, and wrap-around happens if and only if τ is in the "second half," *i.e.*, $\tau \in [(q-1)/2, q)$, and z is wrapped to the first half, *i.e.*, $z = s + \tau \bmod q = s + \tau - q \in [0, (q-1)/2)$. In other words, given public q and d, GPU-Wrap computes

$$\mathsf{wrap} = f_\tau(z) = (\tau \ge (q-1)/2) \cdot (z < (q-1)/2) \cdot \lfloor q/d \rfloor$$

which is an offline-known linear function for the online input z of C if S randomly picks $\tau \in \mathbb{Z}_q$ offline.

To extend DGK to handle probably negative inputs, Veugen [25] argues that, in addition to the above wrap-around offset, it should take $\hat{\alpha} = \alpha - q \bmod 2^\ell$ instead of α to handle the wrap-around error. We do not adopt this trick in GPU-DGK because it takes extra computational and communication costs. Instead, we impose a constraint that $q = 1 \bmod 2^\ell$ throughout our framework, so $\mathsf{DGK}(\alpha, \beta) \neq \mathsf{DGK}(\hat{\alpha}, \beta)$ only occurs when $\alpha = \beta$, implying $x = y$, but it is fine since the result merely serves for $\max(x, y)$. This constraint is specifically beneficial for us, and it seems no related works did it before. Appendix B proves this constraint makes GPU-DGK correct.

Protocol 3 GPU-friendly Wrap-around Handling Protocol

Offline Input (S|C)	τ, pk_{AHE_q}	sk_{AHE_q}
Online Input (S|C)		z
Output (S|C)	$\langle \mathsf{wrap}\rangle_q^{\mathsf{S}}$	$\langle \mathsf{wrap}\rangle_q^{\mathsf{C}}$
Constraints	$\log_2(q) < 23$, $q \equiv 1 \bmod 2^\ell$	

1: **procedure** GPU-Wrap$^{\mathsf{off}}(\tau, \emptyset)$
2: C generates $\langle u\rangle_q^{\mathsf{C}} \in \mathbb{Z}_q^k$ and sends $[\langle u\rangle_q^{\mathsf{C}}]_q$ to S
3: S computes $\mathsf{mult}_d \leftarrow (\tau > (q-1)/2) \cdot \lfloor q/2^\ell \rfloor$
4: S generates a random $\langle \mathsf{wrap}\rangle_q^{\mathsf{S}} \in \mathbb{Z}_q$
5: S sends $[\langle \mathsf{wrap}\rangle_q^{\mathsf{C}}]_q \leftarrow \mathsf{mult}_d \cdot [\langle u\rangle_q^{\mathsf{C}}]_q - \langle \mathsf{wrap}\rangle_q^{\mathsf{S}}$ to C
6: S, C has every values stored in $\mathsf{pre}^{\mathsf{S}}$ or $\mathsf{pre}^{\mathsf{C}}$, resp.
7: **procedure** GPU-Wrap$^{\mathsf{on}}(\mathsf{pre}^{\mathsf{S}}, (\mathsf{pre}^{\mathsf{C}}, z))$
8: C sends $\langle u\rangle_q^{\mathsf{S}} \leftarrow (z < (q-1)/2) - \langle u\rangle_q^{\mathsf{C}}$ to S
9: S: $\langle \mathsf{wrap}\rangle_q^{\mathsf{S}} \leftarrow \mathsf{mult}_d \cdot \langle u\rangle_q^{\mathsf{S}} - \langle \mathsf{wrap}\rangle_q^{\mathsf{S}}$

3.6 GPU-Friendly Secure Comparison Layers

Secure Max Computation and ReLU Layers. As $\max(x, y) = (x \le y) \cdot (y - x) + x$, we compute $\langle \max(x, y)\rangle = \langle x \le y\rangle \cdot (\langle y\rangle - \langle x\rangle) + \langle x\rangle$ with $\langle x \le y\rangle$ output by GPU-DGK, where share multiplication can be done efficiently online by GPU with Beaver's trick. $\mathsf{ReLU}(x)$ is computing $\max(x, 0)$.

Maxpool Layers. Maxpool can use $\max()$ in a binary-tree style, *e.g.*, $\max(\max(\boldsymbol{x}_0, \boldsymbol{x}_1), \max(\boldsymbol{x}_2, \boldsymbol{x}_3))$, where $\boldsymbol{x}_i$ are in the vector form. For n inputs with window size w, the number of comparisons is $n \cdot (1 - 2^{-\lceil \log_2(w) \rceil})$, and we need to invoke our GPU-DGK for $\lceil \log_2(w) \rceil$ rounds. To reduce the invocations of $\max()$, we apply the maxpool layer before the ReLU layer as in Falcon [13] when they are next to each other.

3.7 Inference from SWALP-trained Networks

SWALP's (De)quantization. SWALP [28] quantizes the values of input $\boldsymbol{x}$ (from queries or previous layers) and weight $\boldsymbol{w}$ of linear layers f. It also dequantizes the output values. The boldface type here emphasizes that the inputs can be operated as a set or a tensor. Let bit be the number of bits in fixed-point computation. It defines a quantization function $\mathsf{Q}(\boldsymbol{x}_f)$ that outputs $x_\mathsf{Q} = \mathsf{clip}(\lfloor \boldsymbol{x}_f \cdot 2^{-\exp_x + \mathsf{bit} - 2} \rceil)$, where $\mathsf{clip}(a) = \min(\max(a, -2^{\mathsf{bit}-1}), 2^{\mathsf{bit}-1})$,

$\mathsf{exp}_x = \lfloor(\log_2 \circ \max \circ \mathsf{abs})(\{x_{f,i}\}_i)\rfloor$ is an auxiliary output of an integer indicating the *highest magnitude* among the values in $\boldsymbol{x}_\mathsf{Q}$, and $\lfloor\cdot\rceil$ is *stochastic rounding* [9]. The quantization for the weight $\mathsf{Q}(\boldsymbol{w}_f)$ is also defined similarly. The resulting output $\boldsymbol{y}_\mathsf{Q} = f(\boldsymbol{x}_\mathsf{Q}; \boldsymbol{w}_\mathsf{Q})$ is then dequantized accordingly via $\boldsymbol{y}_f = \mathsf{DeQ}(\boldsymbol{y}_\mathsf{Q}; \mathsf{exp}_x, \mathsf{exp}_w)$, defined to be $\boldsymbol{y}_\mathsf{Q} \cdot 2^{\mathsf{exp}_x + \mathsf{exp}_w - 2\cdot\mathsf{bit}+4}$.

Turning a SWALP-trained model for oblivious inference is challenging because we operate secret shares in $\mathbb{Z}_q$ with (linear) homomorphism, but (de)quantization is *non-linear*.

3.7.1 Precomputing the Maximum

We observe that once the training is done, the maximum value in the weight is fixed, so does exp_w. So we can precompute exp_w for each linear layer. Meanwhile, a trained network has more or less learned the distribution of the input and intermediate data, *i.e.*, x, and thus we can sample x to compute exp_x. So the inference phase can use exp_x and exp_w derived from training, and treat exp_x and exp_w as learnable parameters.

3.7.2 Fusing (De)quantization into Truncation

Suppose, for a linear layer with quantization parameters exp_x and exp_w, y is its quantized output. We want to dequantize it, pass it through (a maxpool layer and) a ReLU layer, and quantize it for the next linear layer with quantization parameters exp_y. GForce does these by fusing the dequantization (DeQ) with the quantization (Q). Theorem 1 proves that this leads to the same result when the non-linear layers between the linear layers are comparison-based ReLU and MaxPool.

Theorem 1 (Fusing (De)quantization)**.** $\mathsf{Q} \circ f_\mathsf{CMP} \circ \mathsf{DeQ}(\boldsymbol{y}) = \mathsf{clip}(\lfloor f_\mathsf{CMP}(\boldsymbol{y})/d)\rceil)$ or $\mathsf{clip}(\lfloor f_\mathsf{CMP}(\boldsymbol{y}) \cdot d)\rceil)$ for some $d \in \mathbb{Z}$, where $f_\mathsf{CMP} = \mathsf{ReLU} \circ \mathsf{MaxPool}$ (or ReLU as an easier case), $\mathsf{ReLU}(\boldsymbol{x}) = \max(\boldsymbol{x}, 0)$, and $\mathsf{MaxPool}(\boldsymbol{x}) = \max(\{\boldsymbol{x}_i\}_i)$.

Proof. We have $(\mathsf{Q} \circ f_\mathsf{CMP} \circ \mathsf{DeQ})(\boldsymbol{y}) = \mathsf{clip}(\lfloor 2^{-\mathsf{exp}_y + \mathsf{bit} - 2} \cdot \max(\{2^{\mathsf{exp}_x + \mathsf{exp}_w - 2\cdot\mathsf{bit}+4} \cdot \boldsymbol{y}_i\}_i, 0)\rceil)$ since $f_\mathsf{CMP}(\{\boldsymbol{x}_i\}_i) = \max(\{\boldsymbol{x}_i\}_i, 0)$, which can be fused into $\mathsf{clip}(\lfloor 2^\mathsf{shift} \cdot f_\mathsf{CMP}(\boldsymbol{y})\rceil)$, where $\mathsf{shift} = \mathsf{exp}_x + \mathsf{exp}_w - \mathsf{exp}_y - \mathsf{bit} + 2$ as $c\max(a,b) = \max(ca, cb)$ for $c > 0$. Depending on the sign of shift, the fused (de)quantization becomes division/multiplication. □

3.7.3 Stochastic Rounding and Truncation Layers

Secure division is not easy even for a public divisor. Some prior works (*e.g.*, [18]) directly divide each share by a (public) divisor d, even for wrapped-around $\langle s\rangle = \{-\tau, s+\tau-q\}$, *i.e.*, $\{\lfloor -\tau/d\rfloor, \lfloor (s+\tau-q)/d\rfloor\}$. The reconstruction is thus incorrect: $\lfloor -\tau/d\rfloor + \lfloor (s+\tau-q)/d\rfloor \approx \lfloor (s-q)/d\rfloor \neq \lfloor s/d\rfloor$.

For (floor) division, we modify a DGK-based approach [25] (on AHE ciphertexts). Our protocol works over secret shares (with the wrap-around protocol) without running the entire DGK explicitly. It also "implicitly" performs stochastic rounding on the output. Our division protocol could incur errors to the output values, but the error distribution of the division is close to the value distribution of *stochastic rounding*:

$$\lfloor s\rceil = \begin{cases} \lfloor s\rfloor + 1, & \text{with probability } s - \lfloor s\rfloor, \\ \lfloor s\rfloor, & \text{with probability } 1 - (s - \lfloor s\rfloor). \end{cases}$$

For our protocol to perform division and stochastic rounding at once, S computes $\langle\lfloor s/d\rceil\rangle^\mathsf{S} \leftarrow -\lfloor\tau/d\rfloor + \langle\mathsf{wrap}\rangle^\mathsf{S}_q$, where τ is a pre-drawn additive mask for s, and C computes $\langle\lfloor s/d\rceil\rangle^\mathsf{C} \leftarrow \lfloor z/d\rfloor + \langle\mathsf{wrap}\rangle^\mathsf{C}_q$, where $\langle\mathsf{wrap}\rangle$ is corresponding to z and the divisor d. Like other GPU-friendly secure online/offline protocols, the server can take advantage of its prior knowledge on the randomness τ in the offline phase. The ideas above result in GPU-Trun (Protocol 4) for SRT layers.

Protocol 4 GPU-friendly Truncation Protocol

Offline Input ($\mathsf{S}\vert\mathsf{C}$)	d, pk_{AHE_q}	d, sk_{AHE_q}
Online Input ($\mathsf{S}\vert\mathsf{C}$)	$\langle s\rangle^\mathsf{S}_q$	$\langle s\rangle^\mathsf{C}_q$
Output ($\mathsf{S}\vert\mathsf{C}$)	$\langle\lfloor s/d\rceil\rangle^\mathsf{S}_q$	$\langle\lfloor s/d\rceil\rangle^\mathsf{C}_q$
Constraints	$\log_2(q) < 23,\ 0 \le s, d < 2^\ell$ $\ell \le \lfloor\log_2(q)\rfloor - 2,\ q \equiv 1 \bmod 2^\ell$	

procedure $\mathsf{GPU\text{-}Trun}^\mathsf{off}$
  S picks $r \in \mathbb{Z}^k_q$, sets $\mathsf{mult}_d \leftarrow (r > (q-1)/2) \cdot \lfloor q/d\rfloor$
  S and C run $\mathsf{GPU\text{-}Wrap}^\mathsf{off}(r)$
  S, C has every values stored in pre^S or pre^C, resp.
procedure $\mathsf{GPU\text{-}Trun}^\mathsf{on}((\mathsf{pre}^\mathsf{S}, \langle s\rangle^\mathsf{S}_q), (\mathsf{pre}^\mathsf{C}, \langle s\rangle^\mathsf{C}_q))$
  S computes $\langle z\rangle^\mathsf{S}_q \leftarrow \langle s\rangle^\mathsf{S}_q + r$ and sends it to C
  C reconstructs $z = s + r \bmod q = \langle z\rangle^\mathsf{S}_q + \langle s\rangle^\mathsf{C}_q$
  S, C gets $\langle\mathsf{wrap}\rangle^\mathsf{S}_q, \langle\mathsf{wrap}\rangle^\mathsf{C}_q \leftarrow \mathsf{GPU\text{-}Wrap}^\mathsf{on}(z)$, resp.
  S: $\langle\lfloor s/d\rceil\rangle^\mathsf{S}_q \leftarrow -\lfloor r/d\rfloor + \langle\mathsf{wrap}\rangle^\mathsf{S}_q$
  C: $\langle\lfloor s/d\rceil\rangle^\mathsf{C}_q \leftarrow \lfloor z/d\rfloor + \langle\mathsf{wrap}\rangle^\mathsf{C}_q$

Theorem 2. The secret value underlying the output of GForce's SRT layers (or GPU-Trun, *i.e.*, Protocol 4) on input s and a divisor d approximates stochastically rounded $\lfloor s/d\rceil$.

Proof. We analyze the value distribution of $\langle s/d\rangle$ when there is no wrap-around. In this proof, we let v_d be the remainder of a variable v ($v \in \{q, \tau, s\}$) with respect to a divisor d. (One may consider v_d as v in $\mathbb{Z}_d$.) The result of the reconstruction is:

$$\lfloor z/d\rceil - \lfloor\tau/d\rceil = \begin{cases} \lfloor s\rfloor + 1, & \text{if } s_d + \tau_d \ge d, \\ \lfloor s\rfloor, & \text{if } s_d + \tau_d < d. \end{cases}$$

As τ is uniformly sampled from $[0, q-1]$, its distribution is $p(\tau_d) = 1/(q_d + d)$ if $\tau_d \ge q_d$ and $p(\tau_d) = 2/(q_d + d)$ if $\tau_d < q_d$. Since we have the constraint that $q_d = 1$, we can assume τ_d is uniformly distributed when $d \gg 1$. (Based on our experimental results, d is usually in $\{2^9, 2^{10}, 2^{11}\}$, meaning the deviation from a uniform distribution is very small.)

Also, as $s_d + \tau_d \geq d \iff \tau_d/d \geq 1 - (s/d - \lfloor s/d \rfloor)$, we can conclude that when $d \gg 1$,

$$\lfloor \frac{z}{d} \rceil - \lfloor \frac{\tau}{d} \rceil = \begin{cases} \lfloor s \rfloor + 1, & \text{with prob. } s_d/d - \lfloor s_d/d \rfloor, \\ \lfloor s \rfloor, & \text{with prob. } 1 - (s_d/d - \lfloor s_d/d \rfloor), \end{cases}$$

which is identical to the distribution of stochastic rounding $\lfloor x/d \rceil$. If it wraps around, $s + r \bmod q = s + \tau - q$. So,

$$\lfloor \frac{z}{d} \rfloor - \lfloor \frac{\tau}{d} \rfloor - \lfloor \frac{q}{d} \rfloor = \begin{cases} \lfloor s \rfloor + 1, & \text{if } s_d + \tau_d - q_d \geq d, \\ \lfloor s \rfloor, & \text{if } 0 < s_d + \tau_d - q_d < d, \\ \lfloor s \rfloor - 1, & \text{if } s_d + \tau_d - q_d < 0. \end{cases}$$

When $d \gg 1$, $q_d/d \approx 0$, so this distribution is very close to the distribution when wrap-around does not happen, and thus it is also close to the distribution of stochastic rounding. □

Truncation Approaches Comparison. Delphi [18] directly truncates the least significant bits without any wrap-around handling. Much plaintext space is wasted to avoid error because the error probability is proportional to the ratio of values hidden in additive SS to the size of the plaintext space. We will empirically show in Section 4.1 that such truncation renders the inference useless due to the tight bit-width.

Delphi [18] picks a plaintext space of 32 bits. It is enough to prevent overflow during linear computations on GPU since the plaintext multipliers are small. However, it is too large for additive SS multiplication on GPU because it requires at least 64-bit bit-width, while GPU can only work with 52 bits for optimized performance. Also, adopting 32-bit bit-width instead of our choice of 22-bit would increase GPU-DGK's computation and communication costs by ∼45%.

Another idea of using the original DGK [25] is to deterministically round up the divided values. We will show by experiments in Section 4.1 that it is orders-of-magnitude slower than our truncation protocol, let alone the extra procedures and bit-width needed (Section 3.5) to prevent off-by-one errors.

4 Experimental Evaluation

Experimental Platform. Following the LAN setting of Gazelle [11] (AWS Virtual Machines (VMs) in us-east-1a) in spirit, our experiments ran on 2 Google Cloud VMs located in the same region (asia-east-1c). They are equipped with Nvidia V100 GPU and run Ubuntu 18.04 LTS. Each has 52GB RAM and 8 virtual Intel Xeon (Skylake) CPUs at 2GHz.

We report the mean of 10 experiment repetitions and provide the standard deviation in $[\cdot]$ if the measurement may be affected by randomness, *e.g.*, runtime and inference accuracy.

Cryptographic Implementations. We code GForce in C++ (compiled by GCC 8.0) and Python 3.6. We marshal network communication and GPU operations via PyTorch 1.3.1 and CUDA 10.0. We assume the bit length of all data, *i.e.*, the input, the intermediate values, and the weights, is 18, except we set $\ell = 20$ for a fair comparison with Gazelle in benchmarking ReLU and maxpool (Tables 4-5). We set bit $= 8$ for 8-bit fixed-point representation in quantization (see Section 3.7).

We use Microsoft SEAL (release 3.3.2)'s BFV-FHE [8] as AHE. The plaintext space for the neural networks is defined by $q = 7340033$. The degree of encryption polynomials (*i.e.*, the number of plaintext slots in a ciphertext) is 16384, and the coefficients modulus of the polynomials is of 438 bits. The ciphertext size is 32MB, which is amortized to 2048 bit for each data entry. We picked the recommended parameters for SEAL to ensure 128-bit security. In the bit-wise comparison of the DGK protocol (GPU-DGK), we also pick the same set of parameters for SEAL except we set $p = 65537$.

BFV-FHE relies on the hardness of the learning-with-error problem. By itself, it does not support circuit privacy because the noise embedded into the ciphertexts may allow the sk holder to infer some partial information about the input plaintexts. To hide S's private input to AHE for linear functions, we adopted noise flooding [1] with 330-bit smudging noise; namely, S adds encryption of 0 with 330-bit noise to each ciphertext before sending it to C. Appendix C.2.3 discusses why this magnitude of the noise is enough for circuit privacy.

Comparing to Prior Arts' Experiments. Gazelle [11], Falcon [13], and Delphi [18] are our major competitors.

Gazelle's implementation is criticized [18,21] for its choice of AHE parameters, which may not ensure circuit privacy. Falcon's choice suffers from the same issue. Changing the parameters will worsen their performance and require re-evaluation. To their advantage, we rely on their figures as is. Non-linear layers are not affected by AHE, and we reproduced their experiments on our Google Cloud VMs. For ReLU (Table 4), our reproduced results are slightly worse than what were reported. For maxpool (Table 5), ours got slightly better.

Falcon did not release their code, and we failed to compile the code of Delphi, so we only quote the figures from their papers. We will give inline remarks on the comparison fairness.

4.1 Comparison-based Layers

As one of our contributions, we demonstrate the performance of our ReLU and maxpool implementations using our GPU-friendly protocol. We chose Gazelle [11] and Falcon [13] for comparison due to their similar paradigm for this part: the server and the client interact to get secret shares of the layer inputs and collaboratively compute the shared outputs. Delphi uses GC for non-linear computation and is not compared.

ReLU Layers. As in Table 4, for 10000-element inputs, we outperform Gazelle by 9× and Falcon by 11× in the online runtime and by at least 8× in the online communication cost.

#input	Framework	ℓ	Comp. (ms)		Comm. (MB)	
			Offline	Online	Offline	Online
10000	Gazelle	20	771 [10.28]	146.77 [4.52]	54.35	16.79
	Falcon	30	361.70	179.60	67.40	15.01
	GForce	20	18426 [82.42]	**16.37** [1.87]	269.54	**1.87**
2^{17}	Gazelle	20	9378 [50.00]	1754 [20.33]	712.35	220
	Falcon	30	*4740	*2354	*883.42	*196.74
	GForce	20	134632 [443.30]	**65.13** [7.05]	2125	**24.5**

Note: ℓ is the bit length of the input in plaintext. $[\cdot]$ denotes the standard deviation. Figures with * are based on estimation. Falcon's figures are quoted from its paper. Gazelle's figures come from our reproduced experiments.

Table 4: (Non-approximated) ReLU Layer Benchmarks

#input	Framework	ℓ	Comp. (ms)		Comm. (MB)	
			Offline	Online	Offline	Online
10000	Gazelle	20	485.60 [8.18]	115.6 [6.45]	38.99	14.27
	Falcon	30	365.50	181.90	68.20	15.02
	GForce	20	30807 [97.59]	**20.11** [0.96]	534.54	**1.40**
40000	Gazelle	20	1828 [17.57]	397.8 [14.25]	155.98	57.08
	Falcon	30	*1462	*727.6	*272.80	*60.08
	GForce	20	43739 [147.70]	**25.88** [2.38]	799.65	**5.61**
2^{18}	Gazelle	20	13681 [95.69]	2950 [42.99]	1022	374.00
	Falcon	30	*9580	*4768	*1787	*393.74
	GForce	20	195783 [644.99]	**88.02** [10.74]	3185.5	**36.75**

Note: ℓ is the bit length of the input in plaintext. Figures with * are based on estimation. $[\cdot]$ denotes the standard deviation. Comp.: Computation; Comm.: Communication.

Table 5: Maxpool (2×2) Layers Benchmarks

Falcon only provided their runtime for 1000 or 10000 inputs. The latter grows up by a ratio of $9.6\times$ over the former, so we treat their fixed runtime cost amortized. According to our estimation of their performance on 2^{17} inputs[3] by linearly scaling its runtime for 10000 inputs, we outperform Gazelle by $27\times$ and Falcon by $36\times$ in the online runtime. The larger speed-up ratio indicates that GForce can handle a large batch of inputs better than prior arts. We also outperform by at least $7\times$ in the online communication cost.

Maxpool Layers. Table 5 shows the runtime and communication costs of maxpool layers of window size 2×2. For 10000 inputs, we outperform Gazelle and Falcon by $6\times$ and $9\times$ for the online runtime and by $10\times$ and $11\times$ for the online communication cost, respectively. Falcon did not provide the figures for 40000 and 2^{18} inputs, so we estimate by scaling its runtime linearly, similar to the case for ReLU. For 40000-inputs, we reduce the online runtime of Gazelle and Falcon by $15\times$ and $28\times$, respectively. For input size up to 2^{18}, we reduce the online runtime by $34\times$ and $54\times$, respectively.

While we just quote the figure from the paper of Falcon [13], we believe its performance would not change dramatically since then, since it also adopts GC for non-linear layer as Gazelle, and Gazelle's performance reproduced in our platform is similar to the reported figures. Note also that the main technical contribution of Falcon lies in its linear layer. The baseline is, our figures are order-of-magnitude better.

[3] Gazelle's implementation is too memory-consuming. We failed to benchmark it on a larger batch for ReLU but managed to do 2^{18} for maxpool.

Dataset	Architecture	Stocha. (**GForce**)	Nearest	Floor	Naïve
CIFAR-10	A-MT	90.82% [0.069%]	90.88%	90.08%	10.62%
	VGG-16	93.22% [0.076%]	93.11%	83.92%	10.06%
CIFAR-100	VGG-16	72.83% [0.075%]	73.14%	64.83%	1.03%

$[\cdot]$ denotes the standard deviation: Nearest/Floor is deterministic. Naïve's are omitted.

Table 6: Accuracy of Different Rounding Methods

SRT Layers. Figure 5 shows the online runtime of the truncation layers. It illustrates a pattern similar to other non-linear layers in that the fixed cost dominates the runtime for small input size. Nevertheless, truncation layers can finish the computation in less than 10ms for small inputs (whose size is less than 10^5). Compared to the layers built on top of DGK, truncation layers are faster by an order of magnitude.

Table 6 shows that our SRT layers are both efficient and accurate. We implement several rounding methods in our truncation layers and test them with CIFAR-10/100 datasets over A-MT and VGG architectures (see Section 4.2 for their description). Stochastic rounding can attain an accuracy similar to the nearest rounding. Nearest rounding and floor rounding could be realized by DGK, but with the runtime increased by an order of magnitude. Naïvely truncating the least significant bits of each additive share is adopted by Delphi [18], but it would make the model almost useless. We suspect the tight plaintext space is a reason (our 22-bit vs. Delphi's 32-bit, resulting in a $2^{10}\times$ increase in the error probability).

While the usage of clip() in SWALP's quantization (mentioned in Section 3.7) appears to be helpful, our experimental results in Tables 7-8 show that it has a very mild impact on the accuracy: SWALP [28] reports their VGG-16 can attain 93.3% and 73.3% accuracy on CIARF-10/100, respectively, while the accuracy of GForce over VGG-16 without clip() drops by less than 0.5pp. We suspect that SWALP-trained neural networks have already optimized the parameters so that the intermediate values rarely exceed the range.

Runtime w.r.t. Input Sizes. Figure 5 sheds light on how the online runtimes of our comparison-based layers grow with the input size. For small ($<10^5$) input sizes, they grow very slowly, indicating the runtime is dominated by the fixed costs, including the constant latency of transferring data between CPU and GPU and over the network. For larger input sizes, the runtime grows linearly. It also explains why we outperform Falcon and Gazelle the most when the input size is large. The maxpool layers have shorter online runtime than ReLU layers and the basic DGK protocol for the same input size because the total number of comparisons that maxpool layers invoke is

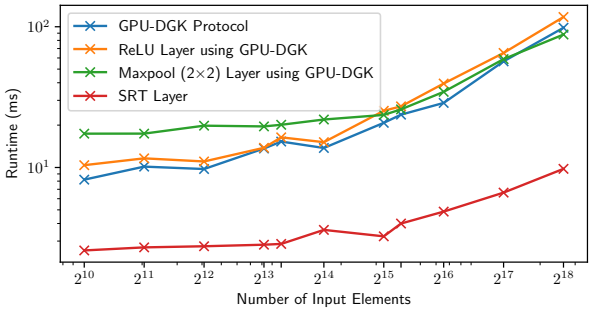


Figure 5: GForce's Online Runtime of Non-linear Layers

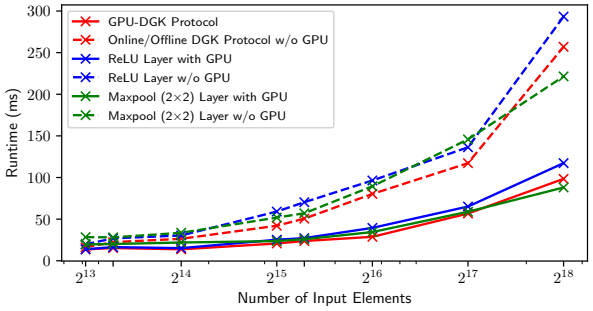


Figure 6: Non-linear Layers Online Runtime, with or without GPU

less than their input size. Section 3.6 explained why maxpool layers require less than n comparison for n inputs.

Gain from GPU. To see how GPU contributes to the lower online runtime, we run GPU-DGK, ReLU layers, and maxpool layers with only CPU. Figure 6 shows that when GPU is not used, the online runtime of our protocols (dash lines) still remain in milliseconds level but are much higher than their GPU-enabled counterparts (solid lines). The gap becomes wider as the input size increases, which aligns with the goal of GForce to efficiently process DNNs for complicated tasks.

4.2 Oblivious Inference

Datasets. CIFAR-10 contains 10 classes of 32×32 colorful images. It has 50,000 training images and 10,000 testing images, each labeled with a class. CIFAR-100 has the same number of colorful images, but they belong to 100 classes, which is harder for classification since each class has less training images, and classifiers need to learn more classes. They are popular benchmarks for (plaintext) neural networks[4].

Neural Networks. The neural network architecture is central to the runtime and accuracy of inference. Among the lists in Tables 7-8, ResNet-32/18 [10], used by Delphi [18] and

Architecture	Framework	Accuracy	Comp. (ms)		Comm. (MB)	
			Offline	Online	Offline	Online
A	MiniONN	81.61%	472000	72000	3046	6226
	Gazelle	-	9340	3560	940	296
	Falcon	81.61%	7200	2880	265	1459
	Delphi	83.33%	41900	380	159	7.5
	Delphi	87.21%	44444	640	247	11
	Delphi	87.77%	101904	7742	3319	281
A-MT	**GForce**	90.82% [0.069%]	249304 [567.25]	**147.26** [5.21]	4698	31.43
BC5	XONN	88.00%		123940		41
BatN-CNN	SHE	92.54%		2258000		160
ResNet-18	SHE	94.62%		12041000		160
VGG-16	**GForce**	**93.12%** [0.076%]	900007 [14106]	352.75 [5.41]	19195	50.46

[·] denotes the standard deviation.

Table 7: CIFAR-10 Benchmarks for Cryptographic Frameworks

Arch.	Framework	Accuracy	Comp. (ms)		Comm. (MB)	
			Offline	Online	Offline	Online
ResNet-32	Delphi	65.77%	109873	2600	1397	74
	Delphi	67.81%	178227	14200	6296	373
VGG-16	**GForce**	**72.83%** [0.075%]	849565 [3171]	**350.10** [10.51]	19197	**50.47**

Table 8: CIFAR-100 Benchmarks for Cryptographic Frameworks

SHE [15], has the best accuracy in plaintext inference, while ResNet-32 is slightly better. VGG-16 [22], used by us, seconds in the plaintext accuracy. An early work MiniONN [14] proposed Architecture A [14, Figure 13], but without reporting its accuracy. We implement a slightly modified version A-MT that replaces all meanpool layers by maxpool layers and inserts truncation layers between linear layers. The accuracy of A-MT on CIFAR-10 is slightly shy to VGG-16.

Neural networks with a higher accuracy incur a longer runtime in general. A and A-MT have the shortest runtime. SHE [15] adopts a convolutional neural network (BatN-CNN in Table 7) [5] with a similar composition as A-MT, except it adopts batch normalization layers instead of truncation layers, and some of its convolution layers have more output channels, meaning that it is more computationally intensive.

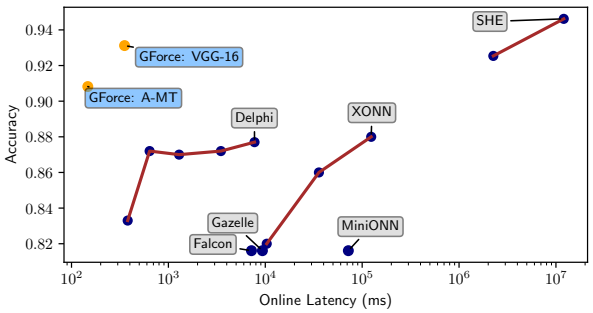
Prior arts also modify architectures to better fit with cryptographic tools. XONN [21] binarizes [20] VGG and prunes out unimportant weights in convolution layers [16] to reduce computational cost. Table 7 reports the one with the highest accuracy (BC5) while more are reported in Figure 7. Delphi [18] also tunes architectures by replacing some ReLU layers by their quadratic approximation. The measured accuracy and performance of Delphi are reported in Tables 7 and 8.

Notably, ResNet-18/32 runs faster than VGG-16 in plaintext[5]. Still, our VGG-16 outperforms Delphi's ResNet-32 in both the accuracy and online runtime for oblivious inference.

VGG-16. We implement VGG-16 [22] for CIFAR-10/100 and train it with SWALP. VGG-16 has 16 convolution lay-

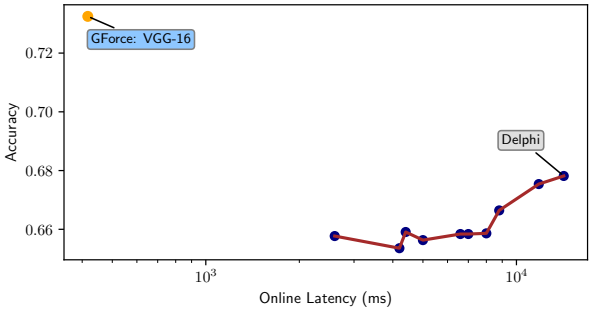
[4]More than 100 machine-learning papers compete for higher accuracy on them (https://paperswithcode.com/sota/image-classification-on-cifar-10 and https://paperswithcode.com/sota/image-classification-on-cifar-100). CIFAR datasets are arguably harder than MNIST evaluated in prior works [14].

[5]This result is from https://github.com/jcjohnson/cnn-benchmarks. We adopt VGG-16 since SWALP provides off-the-shelf training code [28] for it. Also, VGG-16's convolution layers are easier to implement since its stride = 1. It does not mean GForce cannot realize ResNet.



Note: The closer to the upper *left* corner the better

Figure 7: Accuracy and Online Latency on CIFAR-10



Note: The closer to the upper *left* corner the better

Figure 8: Accuracy and Online Latency on CIFAR-100

ers and 3 fully-connected layers and has widespread use in medical diagnosis. Combining with SWALP, our VGG-16 attains 93.12% accuracy on CIFAR-10 (Table 7). Our accuracy outperforms almost all other cryptographic solutions [11, 13, 14, 18, 21], except SHE [15]'s ResNet-18 [10]. However, SHE's ResNet-18 performs impractically slow (taking more than 3 hours, the slowest among all other solutions). Figure 7 compares both the accuracy and latency.

Delphi [18] trained several neural networks that trade accuracy for performance. Our latency is lower than Delphi's, and our most accurate DNN can attain an accuracy higher by at least 5pp than the best of Delphi [18]. Figure 8 plots all the reported accuracy-runtime data. We have a higher accuracy and shorter online runtime than all Delphi's neural networks.

CIFAR-100. To further examine GForce on handling complicated tasks, we test it on CIFAR-100 with 100 classes, 600 images each. Running on VGG-16, we achieve 72.83% accuracy in 350ms. Compared to Delphi [18]'s ResNet-32, our VGG-16 is at least 5pp more accurate. Compared to even the fastest DNN of Delphi [18], GForce is still 6.23× faster.

Comparison with Delphi [18]. We quoted Delphi [18]'s runtime from its paper, which would be lower if it ran in a LAN setting. (Their experiments run two VMs located in different regions of AWS with >20ms network delay.) Table 8 shows that GForce is an order of magnitude faster than Delphi. We reckon that we still have an edge even if its runtime would be halved. Note that accuracy is also our goal.

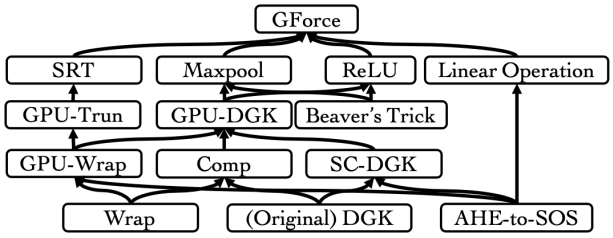


Figure 9: Dependency Graph of Protocols (and their Security Proof)

Comparison based on Existing Architecture (A). Without the learned parameters of architecture A, we cannot guarantee GForce's plaintext is large enough to provide the same accuracy. Instead, we produce a trained DNN via SWALP with a similar architecture A-MT, which attains 90.82% accuracy on CIFAR-10. Compared with MiniONN, Gazelle, and Falcon, GForce attains the shortest online latency and reduces it by 489×, 24×, and 20×, respectively. Figure 7 further illustrates GForce's improvement on CIFAR-10 over other frameworks.

5 Security Analysis

5.1 Threat Model and Protection Scope

We consider probabilistic polynomial-time (PPT) honest-but-curious adversary that controls the communication and either the server or the client. GForce protects the most sensitive information of the network except its architecture and hyper-parameters, which are costly to hide. Specifically, GForce hides the learnable parameters and the kernel size of convolution layers from the client, the query's inputs and outputs from the server, and all the intermediate results of non-output layers from both parties. However, it leaks about DNN's architecture Archi, such as the intermediate outputs' size, the type of each layer, and the window size of pooling layers.

All in all, we have the same privacy guarantee as previous works [11, 13, 14], modulo our unique SRT layers. Each of the n_{T} SRT layers has a divisor parameter d_i, which is always a power-of-2 and within $[2^0, 2^{20}]$. Quantitatively, it means $\log_2(21) \approx 4.4$ bits of information, whereas the weights of the n_{L} linear layers (denoted by $\{M_i\}_{i\in[1:n_{\mathsf{L}}]}$), which GForce can protect from the client, carry at least kilobytes or even megabytes of information. While there seem no inference attacks exploiting such divisors, it may deserve closer scrutiny.

5.2 Overview of Security

GForce composes of many cryptographic protocols, and each can be derived from other sub-protocols. Figure 9 shows their

dependency with arrows from the building blocks to the higher protocols. Following this graph and relying on other proofs, we have the following theorem on the security of GForce.

Theorem 3. GForce's oblivious inference, as a composition of protocols SOS, GPU-DGK, and GPU-Trun over different neural-network layers (third row of Figure 1), is secure:

- A corrupted server's view can be generated by a PPT simulator $\mathsf{Sim}_\mathsf{S}(\mathsf{Archi}, \{d_i\}_{i\in[1:n_\mathsf{T}]}, \{M_i\}_{i\in[1:n_\mathsf{L}]})$.
- A corrupted client's view can be generated by a PPT simulator $\mathsf{Sim}_\mathsf{C}((x, \mathsf{sk}_\mathsf{AHE}), \mathsf{out}, (\mathsf{Archi}, \{d_i\}_{i\in[1:n_\mathsf{T}]}))$, where x is the query and $\mathsf{out} = \mathsf{DNN}(x)$ is the query result.

For all protocols, the simulators of both kinds (for a corrupted client or a corrupted server) also take the following inputs implicitly, which include the description of the cryptographic groups used (*e.g.*, the security parameter), the dimensional information (*e.g.*, $\mathbb{Z}_q^k$ in AHE-to-SOS transformed protocol or $\mathbb{Z}_q$ in pure-AHE protocols), and public key pk_AHE.

The above spells out the relevant parts of $\mathsf{DNN}(\cdot)$ required for Sim_S, the simulator for the corrupted server. Note that the server is run by the model owner and is supposed to know $\{M_i\}$. For brevity, we suppose it is the client who gets the final output out. For many sub-protocols, out will be secret-shared across the server and the client, which can be simulated easily.

Our AHE-to-SOS transformation plays a central role in GForce for deriving many of its sub-protocols. The security proof of AHE-to-SOS transformation is in Appendix D.1.

We prove the security of SC-DGK and GPU-DGK (Protocols 1-2) in Appendices D.2-D.4. We only state the security guarantees of GPU-Wrap and GPU-Trun (Protocols 3-4) but postpone their proof to our full version (which is straightforward given the security of additive SS and AHE).

6 Complementing the Other Frameworks

High-Throughput HE Implementations. We aim for online performance, so we did not optimize for the HE-dominated offline phase. One can employ a more efficient HE implementation (*e.g.*, those used by Falcon/Gazelle) with a more compact encoding or has been optimized for GPU (*e.g.*, as used by HCNN [2]). We can also integrate GForce with HE compilers that aim for high inference throughput (*e.g.*, [6]).

Integration with Delphi [18]. Adopting our GPU-friendly comparison protocols can improve Delphi's performance for its maxpool layers and the remaining ReLU layers.

Oblivious Decision-Tree Inference. For a decision tree, inference proceeds to the left child if the query satisfies the predicate of a node; right otherwise. Tai *et al.* [23] proposed the first approach solely based on AHE that does not need to pad a sparse tree. Their path-cost trick has been utilized in a few subsequent works. In essence, the server runs DGK protocol for each node to produce an AHE-encrypted comparison result bit and adds up these bits for each possible path, which can be readily replaced by our protocols instead.

7 More on Related Works

Gazelle [11] and Falcon [13] use GC for non-linear layers, which heavily relies on AES-NI on CPU for a decent performance [4], with no GPU-friendly counterpart. They also propose a compact encoding to speed up operations of leveled-homomorphic encryption [8]. Falcon [13] aims to improve the linear computations of Gazelle by a Fourier transform-based approach. The best result (by Falcon) takes >2.88s for a CIFAR-10 recognition at <81.62% accuracy.

XONN [21] restricts inference to binarized neural networks (BNN) with confined ($\{-1,1\}$) weights in linear layers and only binary activation functions. It thus manages to use only GC (except for the first layer), which reduces the communication rounds and the total (offline + online) runtime to 5.79s for CIFAR-10 image classification at 81% accuracy (*cf.*, Falcon's 7.22s for ∼81.5%). However, using BNN requires a wider neural network to maintain the accuracy, leading to a longer latency. In particular, for 88% accuracy, it takes ∼2 minutes.

HCNN [2] and Plaid-HE [6] adopt GPU-optimized AHE implementation, but it can only handle non-linear layers with approximation, sacrificing accuracy. Also, the overhead due to AHE is still large. Our AHE-to-SOS approach remains beneficial for moving the AHE-related operations offline and supporting common non-linear layers without approximation.

Using GPU is also a relatively new idea for SGX-based frameworks. Slalom [24] securely outsources linear operation from SGX to untrusted GPU for inference. Goten [19] solves the challenges in supporting private learning left by Slalom.

8 Conclusion

GForce is an efficient oblivious inference protocol that works over a low-precision integer domain while maintaining high accuracy. For this, we adopt SWALP [28] and formulate stochastic rounding and truncation layers that fuse multiple operations SWALP needs for efficiency and accuracy. We also propose cryptographic protocols for leveraging GPU parallelism even for non-linear layers, which reduce the online latency and communication cost by orders of magnitude. These are validated by our evaluation comparing prior frameworks.

We hope that this work can inspire further research of machine-learning experts to devise new algorithms compatible with finite fields used in cryptography and stimulate cryptographers to propose more GPU-friendly protocols.

With a secret-sharing-based design, it seems promising to explore if GForce can be extended to secure outsourced inference [17] or training, say, by using 3 non-colluding servers.

References

[1] Gilad Asharov, Abhishek Jain, Adriana López-Alt, Eran Tromer, Vinod Vaikuntanathan, and Daniel Wichs. Multiparty computation with low communication, computation and interaction via threshold FHE. In *EUROCRYPT*, pages 483–501, 2012.

[2] Ahmad Al Badawi, Bharadwaj Veeravalli, Jie Lin, Xiao Nan, Kazuaki Matsumura, and Khin Mi Mi Aung. Multi-GPU design and performance evaluation of homomorphic encryption on GPU clusters. *IEEE Trans. Parallel Distributed Syst.*, 32(2):379–391, 2021.

[3] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *CRYPTO*, pages 420–432, 1991.

[4] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient garbling from a fixed-key blockcipher. In *IEEE Symposium on Security and Privacy*, 2013.

[5] Hervé Chabanne, Amaury de Wargny, Jonathan Milgram, Constance Morel, and Emmanuel Prouff. Privacy-preserving classification on deep neural network. Cryptology ePrint 2017/035, 2017. Also presented at Real World Crypto Symposium.

[6] Huili Chen, Rosario Cammarota, Felipe Valencia, and Francesco Regazzoni. PlaidML-HE: Acceleration of deep learning kernels to compute on encrypted data. In *Intl' Conf. on Computer Design*, pages 333–336, 2019.

[7] Ivan Damgård, Martin Geisler, and Mikkel Krøigaard. Efficient and secure comparison for on-line auctions. In *ACISP*, pages 416–430, 2007.

[8] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint, 2012/144, 2012.

[9] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *ICML*, pages 1737–1746, 2015.

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, pages 770–778, 2016.

[11] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In *Usenix Security*, pages 1651–1669, 2018.

[12] Laine Kim. Simple Encrypted Arithmetic Library 2.3.1.

[13] Shaohua Li, Kaiping Xue, Bin Zhu, Chenkai Ding, Xindi Gao, David S. L. Wei, and Tao Wan. FALCON: A Fourier transform based approach for fast and secure convolutional neural network predictions. In *CVPR*, pages 8702–8711, 2020.

[14] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. Oblivious neural network predictions via MiniONN transformations. In *CCS*, pages 619–631, 2017.

[15] Qian Lou and Lei Jiang. SHE: A fast and accurate deep neural network for encrypted data. In *NeurIPS*, pages 10035–10043, 2019.

[16] Jian-Hao Luo, Hao Zhang, Hong-Yu Zhou, Chen-Wei Xie, Jianxin Wu, and Weiyao Lin. ThiNet: Pruning CNN filters for a thinner net. *IEEE Trans. Pattern Anal. Mach. Intell.*, 41(10):2525–2538, 2019.

[17] Jack P. K. Ma, Raymond K. H. Tai, Yongjun Zhao, and Sherman S. M. Chow. Let's stride blindfolded in a forest: Sublinear multi-client decision trees evaluation. In *NDSS*. Internet Society, 2021.

[18] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. Delphi: A cryptographic inference service for neural networks. In *USENIX Security*, pages 2505–2522, 2020.

[19] Lucien K. L. Ng, Sherman S. M. Chow, Anna P. Y. Woo, Donald P. H. Wong, and Yongjun Zhao. Goten: GPU-Outsourcing Trusted Execution of Neural Network Training. In *AAAI Conf. on Artificial Intelligence*, 2021.

[20] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. XNOR-Net: Imagenet classification using binary convolutional neural networks. In *ECCV Part IV*, pages 525–542, 2016.

[21] M. Sadegh Riazi, Mohammad Samragh, Hao Chen, Kim Laine, Kristin E. Lauter, and Farinaz Koushanfar. XONN: XNOR-based oblivious deep neural network inference. In *USENIX Security*, pages 1501–1518, 2019.

[22] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2015.

[23] Raymond K. H. Tai, Jack P. K. Ma, Yongjun Zhao, and Sherman S. M. Chow. Privacy-preserving decision trees evaluation via linear functions. In *ESORICS Part II*, pages 494–512, 2017.

[24] Florian Tramèr and Dan Boneh. Slalom: Fast, verifiable and private execution of neural networks in trusted hardware. In *ICLR*, 2019.

[25] Thijs Veugen. Encrypted integer division and secure comparison. *Int. J. Appl. Cryptogr.*, 3(2):166–180, 2014.

Protocol 5 DGK Comparison with Private Inputs (Review)

Input (S|C)	$0 \le \alpha < 2^\ell$, pk_{AHE}	$0 \le \beta < 2^\ell$, sk_{AHE}
Output (S|C)	$\langle \alpha \le \beta \rangle_2^{\mathsf{S}}$	$\langle \alpha \le \beta \rangle_2^{\mathsf{C}}$
Constraint	$\ell \le \lfloor \log_2(q) \rfloor - 2$	

procedure DGK(α, β)
  C sends $[\beta_i]$'s to S where $\beta_{\ell-1} \cdots \beta_0 \leftarrow \beta$ and $\beta_{-1} \leftarrow 0$
  S decomposes $\alpha_{\ell-1} \cdots \alpha_0 \leftarrow \alpha$ and sets $\alpha_{-1} \leftarrow 0$
  S: $\langle \alpha \le \beta \rangle_q^{\mathsf{S}} \leftarrow 1 \oplus \delta^{\mathsf{S}}$, $a \leftarrow 1 - 2 \cdot \delta^{\mathsf{S}}$, where $\delta^{\mathsf{S}} \in \mathbb{Z}_q^k$
  for $i \leftarrow \{\ell - 1, \ldots, -1\}$ **do**
    S: $a \leftarrow \delta^{\mathsf{S}}$ if $i = -1$
    S: $[\alpha_i \oplus \beta_i] \leftarrow (1 - 2\alpha_i)[\beta_i] + [\alpha_i]$
    S: $[b_i] \leftarrow [a + \alpha_i] - [\beta_i] + 3 \cdot \sum_{j=i+1}^{\ell-1} [\alpha_j \oplus \beta_j]$
    S blinds $[b_i] \leftarrow r_i \cdot [b_i]$ with random $r_i \in \mathbb{Z}_q^*$
  S shuffles $[b_i]$'s and sends them to C so C can decrypt
  C: $\langle \alpha \le \beta \rangle_q^{\mathsf{C}} \leftarrow 1$ if any decrypted b_i is 0; otherwise 0.

[26] Thijs Veugen. Correction to "Improving the DGK comparison protocol". Cryptology ePrint, 2018/1100, 2018.

[27] Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. FALCON: Honest-majority maliciously secure framework for private deep learning. *PoPETs*, 2021(1):187–207, 2021.

[28] Guandao Yang, Tianyi Zhang, Polina Kirichenko, Junwen Bai, Andrew Gordon Wilson, and Christopher De Sa. SWALP : Stochastic weight averaging in low precision training. In *ICML*, pages 7015–7024, 2019.

A Review of DGK and Basic Wrap-around

For the sake of completeness, we review four protocols here.

Protocol 5 describes the original DGK protocol [7], which uses AHE and only supports comparisons of positive integers.

Protocol 6 modifies an improved version of DGK [26] such that it can compare additively-shared negative integers. However, after our modification for making it GPU-friendly, the results might become wrong in the corner case that the inputs are equal. We thus prove its correctness in Appendix B.

Protocol 7 detects wrap-around when additive shares are divided by a public divisor and compensates for the wrapped values to maintain the correctness. This idea was proposed by Veugen [25] and adopted by another work of Veugen [26].

Protocol 8 reviews how to perform multiplication over additive secret shares using Beaver's trick [3].

Protocol 6 Comparison over AHE, with $\mathbb{Z}_q$ output (Review)

Input (S|C)	$[x]$, $[y]$, and pk_{AHE_q}	sk_{AHE_q}
Output (S|C)	$\langle x \le y \rangle_q^{\mathsf{S}}$	$\langle x \le y \rangle_q^{\mathsf{C}}$
Constraints	$-2^{\ell-1} < x, y < 2^{\ell-1}$ $\ell \le \lfloor \log_2(q) \rfloor - 2$, $q \equiv 1 \bmod 2^\ell$	

procedure Comp($[x], [y]$)
  S computes $\alpha \leftarrow \tau \bmod 2^\ell$ where $\tau \in \mathbb{Z}_q$
  S computes $[z] \leftarrow [y] - [x] + [2^\ell + \tau]$ and sends it to C
  C decrypts $[z]$ and computes $\beta = z \bmod 2^\ell$
  S, C gets $\langle \alpha > \beta \rangle_2^{\mathsf{S}}, \langle \alpha > \beta \rangle_2^{\mathsf{C}} \leftarrow$ DGK(α, β), resp.
  S, C gets $\langle \mathsf{wrap} \rangle_q^{\mathsf{S}}, \langle \mathsf{wrap} \rangle_q^{\mathsf{C}} \leftarrow \mathsf{Wrap}(r, z)$, resp.
  S: $\langle x \le y \rangle_q^{\mathsf{S}} \leftarrow -\lfloor \tau / 2^\ell \rfloor - (1 - \langle \alpha \le \beta \rangle_2^{\mathsf{S}}) + \langle \mathsf{wrap} \rangle_q^{\mathsf{S}}$
  C: $\langle x \le y \rangle_q^{\mathsf{C}} \leftarrow \lfloor z / 2^\ell \rfloor + \langle \alpha \le \beta \rangle_2^{\mathsf{C}} + \langle \mathsf{wrap} \rangle_q^{\mathsf{C}}$

Protocol 7 Wrap-around Handling (Basic version)

Input (S|C)	$\tau \in \mathbb{Z}_q$, d, pk_{AHE}	$z \in \mathbb{Z}_q$, sk_{AHE}
Output (S|C)	$\langle \mathsf{wrap} \rangle_q^{\mathsf{S}}$	$\langle \mathsf{wrap} \rangle_q^{\mathsf{C}}$

procedure Wrap(τ, z, d)
  C sets $u \leftarrow (z < (q-1)/2)$ and sends $[u]$ to S
  S computes $[\mathsf{wrap}] \leftarrow (\tau \ge (q-1)/2) \cdot \lfloor q/d \rfloor \cdot [u]$
  S generates a random $\langle \mathsf{wrap} \rangle_q^{\mathsf{S}} \in \mathbb{Z}_q$
  S sends $[\langle \mathsf{wrap} \rangle_q^{\mathsf{C}}] \leftarrow [\mathsf{wrap}] - [\langle \mathsf{wrap} \rangle_q^{\mathsf{S}}]$ to C
  C obtains $\langle \mathsf{wrap} \rangle_q^{\mathsf{C}}$ via decrypting $[\langle \mathsf{wrap} \rangle_q^{\mathsf{C}}]$

Protocol 8 Beaver's Trick for Share Multiplication (Review)

Offline Input (S|C)	pk_{AHE}	sk_{AHE}
Online Input (S|C)	$\langle a \rangle^{\mathsf{S}}, \langle b \rangle^{\mathsf{S}}$	$\langle a \rangle^{\mathsf{C}}, \langle b \rangle^{\mathsf{C}}$
Output	$\langle a \cdot b \rangle^{\mathsf{S}}$	$\langle a \cdot b \rangle^{\mathsf{C}}$
Constraints	$a, b \in \mathbb{Z}_q$	

procedure ShareMul$^{\mathsf{off}}$
  S picks $\langle u \rangle^{\mathsf{S}}, \langle u \rangle^{\mathsf{S}}, l \in \mathbb{Z}_q$ and sets $\langle z \rangle^{\mathsf{S}} \leftarrow \langle u \rangle^{\mathsf{S}} \langle v \rangle^{\mathsf{S}} - l$
  C picks $\langle u \rangle^{\mathsf{C}}, \langle v \rangle^{\mathsf{C}} \in \mathbb{Z}_q$ and sends $[\langle u \rangle^{\mathsf{C}}], [\langle v \rangle^{\mathsf{C}}]$ to S
  S sends $[\tau] \leftarrow [\langle u \rangle^{\mathsf{C}}] \cdot \langle v \rangle^{\mathsf{S}} + \langle u \rangle^{\mathsf{S}} \cdot [\langle v \rangle^{\mathsf{C}}] + [l]$ to C
  C decrypts $[\tau]$ and sets $\langle z \rangle^{\mathsf{C}} \leftarrow \tau + \langle u \rangle^{\mathsf{C}} \cdot \langle v \rangle^{\mathsf{C}}$
  S, C has every values stored in $\mathsf{pre}^{\mathsf{S}}$ or $\mathsf{pre}^{\mathsf{C}}$, resp.
procedure ShareMul$^{\mathsf{on}}$($\{\mathsf{pre}^{\mathsf{role}}, \langle a \rangle^{\mathsf{role}}, \langle b \rangle^{\mathsf{role}}\}_{\mathsf{role} \in \{\mathsf{S},\mathsf{C}\}}$)
  S sends $\langle e \rangle^{\mathsf{S}} \leftarrow \langle a \rangle^{\mathsf{S}} - \langle u \rangle^{\mathsf{S}}$, $\langle f \rangle^{\mathsf{S}} \leftarrow \langle b \rangle^{\mathsf{S}} - \langle v \rangle^{\mathsf{S}}$ to C
  C sends $\langle e \rangle^{\mathsf{C}} \leftarrow \langle a \rangle^{\mathsf{C}} - \langle u \rangle^{\mathsf{C}}$, $\langle f \rangle^{\mathsf{C}} \leftarrow \langle b \rangle^{\mathsf{C}} - \langle v \rangle^{\mathsf{C}}$ to S
  S and C reconstruct e and f
  S sets $\langle a \cdot b \rangle^{\mathsf{S}} \leftarrow \langle a \rangle^{\mathsf{S}} \cdot f + e \cdot \langle b \rangle^{\mathsf{S}} + \langle z \rangle^{\mathsf{S}}$
  C sets $\langle a \cdot b \rangle^{\mathsf{C}} \leftarrow \langle a \rangle^{\mathsf{C}} \cdot f + e \cdot \langle b \rangle^{\mathsf{C}} - e \cdot f + \langle z \rangle^{\mathsf{C}}$

B Correctness of GPU-DGK

GPU-DGK is an online/offline transformation of Comp (Protocol 6), so we focus on proving the correctness of Comp.

Inside Comp, DGK (Protocol 5) is the original comparison protocol [7], and thus its correctness has been proved. Veugen [26] also provided another correctness analysis.

What is left is to prove that our extension of DGK for

probably negative inputs is correct when we do follow Veugen [25]'s modification (as mentioned in Section 3.5).

Comp outputs $\langle x \le y \rangle^{\mathsf{S}}$ and $\langle x \le y \rangle^{\mathsf{C}}$, and we prove that the reconstructed results $(x \le y)'$ equals $(x \le y)$ except when $x = y$. We unroll $(x \le y)' = \langle x \le y \rangle^{\mathsf{S}} + \langle x < y \rangle^{\mathsf{C}} = \lfloor z/2^\ell \rfloor - \lfloor \tau/2^\ell \rfloor - \langle \alpha > \beta \rangle_2 + \mathsf{wrap}$. In this proof, we use $(expr)_{2^\ell}$ to denote the evaluation of an arithmetic expression $expr$ modulo 2^ℓ, *i.e.*, $expr \bmod 2^\ell$.

We first consider the case that z does not wrap around, *i.e.*, $z = y - x + 2^\ell + \tau < q$. We can omit wrap in $(x \le y)'$ as it equals to 0. Also, be reminded that $\lfloor (a+b)/2^\ell \rfloor = \lfloor (a_{2^\ell} + b_{2^\ell})/2^\ell \rfloor + \lfloor a/2^\ell \rfloor + \lfloor b/2^\ell \rfloor$ for any integer a and b. We have $(x \le y)' = \lfloor (y - x + \tau + 2^\ell)/2^\ell \rfloor - \lfloor \tau/2^\ell \rfloor - (\alpha > \beta) = \lfloor (y - x + \tau_{2^\ell})/2^\ell \rfloor + \lfloor \tau/2^\ell \rfloor - \lfloor \tau/2^\ell \rfloor + (1 - (\alpha > \beta)) = \lfloor (y - x + \tau_{2^\ell})/2^\ell \rfloor + (\alpha \le \beta)$. We denote $\lfloor (y - x + \tau_{2^\ell})/2^\ell \rfloor$ by Δ, so $(x \le y)' = \Delta + (\alpha \le \beta)$.

Since $x, y \in (-2^{\ell-1}, 2^{\ell-1})$, we have $-2^\ell + 1 < y - x + \tau_{2^\ell} < 2 \cdot 2^\ell$. Thus, $\Delta = 0$ or 1. For $\Delta = 0$, we have $y - x + \tau_{2^\ell} \in [0, 2^\ell)$. Thus, there is no further wrap around (over $\mathbb{Z}_{2^\ell}$) for $\beta = y - x + \tau_{2^\ell}$. As a result, $\alpha \le \beta \iff \tau_{2^\ell} \le y - x + \tau_{2^\ell} \iff x \le y$. It means $(x \le y)' = (\alpha \le \beta) = (x \le y)$

For $\Delta = 1$, we have $2^\ell \le y - x + \tau_{2^\ell}$. Hence, $x < y$ because $0 < 2^\ell - \tau_{2^\ell} \le y - x$. Also, β can further be rewritten as $y - x + \tau_{2^\ell} - 2^\ell$, where 2^ℓ is the offset due to the wrap around. As a result, we have $\beta = \tau_{2^\ell} + (y - x - 2^\ell) < \tau_{2^\ell} = \alpha$ because $y - x < 2^\ell$. Thus, $(\alpha \le \beta) = 0$ and $(x \le y)' = \Delta + (\alpha \le \beta) = 1$, matching the inferred fact that $x < y$.

Now, we assume z wraps around and prove that $(x \le y)' \ne (x \le y) \iff x = y$. z wrapping around implies $z = y - x + 2^\ell + \tau - q$. Then $(x \le y)' = \lfloor z/2^\ell \rfloor - \lfloor \tau/2^\ell \rfloor - \langle \alpha > \beta \rangle_2 + \mathsf{wrap} = \lfloor (y - x + 2^\ell + \tau - q)/2^\ell \rfloor - \lfloor \tau/2^\ell \rfloor - \langle \alpha > \beta \rangle_2 + \lfloor q/d \rfloor = (\lfloor (y - x + \tau_{2^\ell} + q_{2^\ell})/2^\ell \rfloor + 1 + \lfloor \tau/2^\ell \rfloor - \lfloor q/2^\ell \rfloor) - \lfloor \tau/2^\ell \rfloor - \langle \alpha > \beta \rangle_2 + \lfloor q/d \rfloor = \lfloor (y - x + \tau_{2^\ell} + q_{2^\ell})/2^\ell \rfloor + \langle \alpha \le \beta \rangle_2$. Recall that we have imposed a constraint $q_{2^\ell} = 1$ and that $(\alpha \le \beta) = (r_{2^\ell} \le (y - x + \tau_{2^\ell} + q_{2^\ell})_{2^\ell})$. We can rewrite $(x \le y)' = \lfloor ((y+1) - x + \tau_{2^\ell})/2^\ell \rfloor + \langle r_{2^\ell} \le ((y+1) - x + \tau_{2^\ell})_{2^\ell} \rangle_2$. This is equivalent to $(x \le (y+1))'$ with no wrap around, whose correctness has been proven above. Hence, $(x \le y)' = (x \le (y+1))$, which is wrong only when $x = y$.

C Security of Cryptographic Building Blocks

C.1 Additive Secret Sharing

$(2,2)$-additive SS leaks no information of the secret (except its domain). Formally, there exists a PPT simulator Sim such that $\langle m \rangle^{\mathsf{role}} \approx \mathsf{Sim}(\mathbb{Z}_p^n)$, for any secret $m \in \mathbb{Z}_p^n$ and $\mathsf{role} \in \{\mathsf{S}, \mathsf{C}\}$.

C.2 Additive Homomorphic Encryption

C.2.1 Semantic Security and Circuit Privacy

AHE schemes possess the following security properties.

Semantic security requires that any PPT adversary cannot distinguish the plaintext of a ciphertext. More formally, $(pk, \mathsf{AHE.Enc}(pk, m_0)) \approx_{\mathsf{c}} (pk, \mathsf{AHE.Enc}(pk, m_1))$ for any messages m_0 and m_1. The views are distributed over the choices of public key pk and the random coins of AHE.Enc.

Circuit privacy requires that any PPT adversary, even holding the secret key of AHE, cannot learn anything about the homomorphic operations f performed on an AHE ciphertext except what can be inferred by the message, *i.e.*, $f(m)$. In other words, there exists a PPT simulator Sim such that $(sk, pk, \{m_i\}_{i \in [1:n]}, f(\{m_i\}_{i \in [1:n]}), \{\mathsf{ct}_i\}_{1 \in [1:n]}, \mathsf{ct}') \approx_{\mathsf{c}} \mathsf{Sim}(sk, pk, \{m_i\}_{1 \in [1:n]}, f(\{m_i\}_{1 \in [1:n]}))$, where $\{\mathsf{ct}_i = \mathsf{AHE.Enc}(pk, m_i)\}_{i \in [1:n]}$, $\mathsf{ct}' = \mathsf{AHE.Eval}(pk, \{m_i\}_{i \in [1:n]}, f))$, $n < \mathsf{poly}(\lambda)$, and f is a linear function.

C.2.2 Noise Flooding

Our implementation adopts BFV-FHE [8] as the AHE scheme for offline preprocessing. However, the "textbook" BFV-FHE scheme does not preserve circuit privacy because the secret key holder may be able to extract information about the homomorphic operations performed on a ciphertext. Very roughly, the culprit is the noise e it uses to hide the plaintext. The corresponding defense is *noise flooding* [1]. Before S sends the ciphertext to C for decryption, we add an extra huge "smudging" noise in the ciphertext to smudge out the distribution of the original output noise $f(e)$. By the smudging lemma [1], the smudging noise in any two ciphertexts produced by the same protocol will have a statistical distance of $2^{-\lambda}$ if the smudging noise is λ bit larger than the original output noise.

Although the exact parameters in the linear function f are secret of S and they may be uncertain before the actual execution of the protocol, we still can evaluate B_{eval}, the bound of the noise's magnitude $|f(e)|$. Then, we define the bound of the smudging noise to be $B_{\mathsf{smud}} \ge 2^\lambda \cdot B_{\mathsf{eval}}$. The smudging noise $e^{\mathsf{smud}} = (e_1^{\mathsf{smud}}, e_2^{\mathsf{smud}})$ are uniformly sampled from $[-B_{\mathsf{smud}}, B_{\mathsf{smud}}]^2$. The smudged ciphertext is $f(c) + e^{\mathsf{smud}}$.

C.2.3 An Estimation of the Noise Bound

Our implementation has two sets of (n, p) where n is the degree of the ciphertext polynomial and $\mathbb{Z}_p$ is the underlying field. We use $(16384, 65537)$ in SC-DGK since it computes at the bit level and $(16384, 7340033)$ for the rest (*e.g.*, GPU-DGK) of the offline phase (which is denoted by $\mathbb{Z}_q$ here). According to the manual of SEAL [12], a (loose) noise bound of a newly encrypted ciphertext is $B^{\mathsf{Enc}} = np(p + 336/\sqrt{2\pi})$. Their corresponding B^{Enc} is of 44 bits and 51 bits.

SEAL's manual [12] suggests the noise bound after these operations is $B^{\mathsf{Eval}} = B^{\mathsf{Enc}} \cdot knp$, where k is the number of addition over the ciphertexts. For our DGK bit-comparison protocol, k roughly equals the bit-length $\ell \approx 20$. Hence, B^{Eval} is about 78 bits. The largest k occurs in our VGG-16's fully-connected layer, which is up to $k \approx 2^{10}$, and the corresponding

B^{Eval} is about 98 bits.

For 128-bit security, as suggested by the smudging lemma [1], we add extra 128 bits to the noise bound B^{Eval} for smudging noise, meaning that B^{Smud} should be at least 206 for DGK bit-comparison and 226 for the other protocols. For 128-bit security and $n = 16384$, we follow SEAL's recommendation to pick a 438-bit coefficient modulus. B^{Smud} should be smaller than it to prevent incorrect decryption. Hence, we set B^{Smud} to be 330 bits to leave a safety margin for security while avoiding B^{Smud} being too large for correct decryption.[6]

D Security Proofs for Our Protocols

We use the simulation-based security definition for two-party computation. Our goal is to exhibit a PPT simulator $\mathsf{Sim}_{\mathsf{role}}$ for party $\mathsf{role} \in \{\mathsf{S}, \mathsf{C}\}$ taking its private input $\mathsf{in}_{\mathsf{role}}$, its private output $\mathsf{out}_{\mathsf{role}}$, and the leakage $\mathsf{leak}_{\mathsf{role}}$ it could learn from the protocol to be proven, which can generate a view computationally indistinguishable from $\mathsf{View}_{\mathsf{role}}$, its view in a real protocol invocation, *i.e.*, $\mathsf{View}_{\mathsf{role}} \approx_c \mathsf{Sim}(\mathsf{in}_{\mathsf{role}}, \mathsf{out}_{\mathsf{role}}, \mathsf{leak}_{\mathsf{role}})$.

Due to the page limit, we only show the security proof of our AHE-to-SOS transformation for its central role in GForce, which leads to the proof for the GForce as a whole. For other protocols, we mostly only highlight the intuition for the simulation or rely on the security arguments of the original protocols in the respective papers.

D.1 Security of AHE-to-SOS Transformation

Theorem 4. The protocol obtained from our AHE-to-SOS transformation remains secure against a semi-honest server or client in that it does not leak more than the original protocol.

Security Proof against a Corrupted Client. The simulator $\mathsf{Sim}(\mathsf{in}_{\mathsf{C}}, \mathsf{out}_{\mathsf{C}}, \mathsf{leak}_{\mathsf{C}})$ can be constructed with leakage $\mathsf{leak}_{\mathsf{C}}$ being empty. Namely, the private input in_{C} of C is χ^{C} and its private output $\langle f(\chi)\rangle^{\mathsf{C}}$, which comes from decrypting the only protocol message $[\langle f(\chi)\rangle^{\mathsf{C}}]$ it receives from S, can be simulated by randomly picking a secret share $\langle f(\chi)\rangle^{\mathsf{C}}$ from an appropriate domain and encrypting it under $\mathsf{pk}_{\mathsf{AHE}}$.

We remark that this is a sub-protocol in which the client would probably be interacting with server S that takes the private output $\mathsf{out}_{\mathsf{S}}$ of the server as an input in a subsequent step. In this case, eventually, we need $\mathsf{out}_{\mathsf{S}}$ to simulate the subsequent view of the client (otherwise, the private output to the client is just a random value). This can be easily simulated by using the knowledge of $f(\chi)$ by $\mathsf{out}_{\mathsf{S}} = f(\chi) - \langle f(\chi)\rangle^{\mathsf{C}}$.

[6] Our security model assumes the client to be semi-honest. When C is malicious, noise flooding may provide less protection than expected. A malicious client may pick an initial noise larger than the protocol specified for encryption, making our estimation on the noise bound too small. The smudging noise cannot provide sufficient obfuscation in this case.

Security Proof against a Corrupted Server. The simulator $\mathsf{Sim}(\mathsf{in}_{\mathsf{S}}, \mathsf{out}_{\mathsf{S}}, \mathsf{leak}_{\mathsf{S}})$ can be constructed with leakage $\mathsf{leak}_{\mathsf{S}}$ being empty. Suppose the private server input is $\langle\chi\rangle^{\mathsf{S}}$, and the server randomness is r^{S}. S sees two protocol messages. The first one $[r^{\mathsf{C}}]$ can be simulated by encryption of a dummy plaintext (*e.g.*, 0) of the same size, which remains indistinguishable to S since S does not have the decryption key. The second protocol message can be easily simulated by randomly picking an element Y from an appropriate domain.

The corresponding private output of the client can be simulated given the knowledge of $f(\cdot)$ and $f(\chi)$ by $\mathsf{out}_{\mathsf{C}} = f(\chi) - f(\langle\chi\rangle^{\mathsf{S}} + Y) + r^{\mathsf{S}}$ since the simulated view based on Y and the server randomness r^{S} will make the server computes $\mathsf{out}_{\mathsf{S}} = f(\langle\chi\rangle^{\mathsf{S}} + Y) - r^{\mathsf{S}}$.

D.2 Security of SC-DGK, DGK, and Comp

Theorem 5. The vanilla DGK, Protocol 5, and its SOS version, Protocol 1, are secure against a semi-honest PPT corrupted server or client that learns nothing more than its input.

The proof of Protocol 5 can be found in the original paper [7]. Since Lines 5-9 of Protocol 5 can be aggregated into a linear function, we can use our AHE-to-SOS transformation to produce Protocol 1, which is secure by Theorem 4.

Theorem 6. Protocol 6 is secure against a semi-honest PPT corrupt server or client that learns nothing more than its input.

Protocol 6 slightly modifies the existing protocol of Veugen [26] and can be proven secure in a similar manner.

D.3 Security of GPU-Wrap and GPU-Trun

Theorem 7. GPU-Wrap and GPU-Trun (Protocols 3-4) are secure against any semi-honest, computationally bounded, corrupted server or client, *i.e.*, either one cannot learn anything other than its input and the corresponding protocol output.

We have PPT simulators $\mathsf{Sim\text{-}Wrap}_{\mathsf{S}}()$, $\mathsf{Sim\text{-}Wrap}_{\mathsf{C}}()$, $\mathsf{Sim\text{-}Trun}_{\mathsf{S}}(d)$, $\mathsf{Sim\text{-}Trun}_{\mathsf{C}}(d)$, which simulate the respective view of the server and of the client in GPU-Wrap and GPU-Trun, respectively, where d is the (public) divisor for truncation. The proof is deferred to the full version.

D.4 Security of GPU-DGK

Theorem 8. GPU-DGK (Protocol 2), as the SOS version of Comp (Protocol 6), is secure against a semi-honest PPT corrupt server or client that learns nothing more than its input.

We first note that Comp (Protocol 6) can be viewed as the secure computation of a linear function defined over the private server input $([x], [y])$ because both of its sub-protocols DGK() and Wrap() can be expressed as a linear function.

Protocol 2 is almost an AHE-to-SOS transformed version of Protocol 6. The difference is that Protocol 2 takes additive SS as inputs, *i.e.*, $\langle x \rangle$ and $\langle y \rangle$. We can still view the private function $f()$ of S, which is the private input of S, as expressed in the form of $\langle x \rangle^{\mathsf{S}}$ and $\langle y \rangle^{\mathsf{S}}$ since the only operations over them in Protocol 2 are, again, linear computations.

More specifically, the computation of z, which is to be sent to C in an encryption form originally, is now sent as an additive SS that C can recover the original value of z. Since Protocol 6 is secure, we can use its simulator to create the view, including z and the client shares of the two sub-protocols for a corrupted client. Similarly, the view for a corrupted server can also be simulated. With these simulators and the security of our AHE-to-SOS transformation, the resulting Protocol 2 is secure for computing the same linear function as its "underlying" Protocol 6.

D.5 Security of GForce

Security Proof of GForce against a Corrupted Server. We prove by hybrid games that the simulated view is indistinguishable from the server's view, which additionally includes the additive SS of all intermediate values of the underlying protocols.

- Hyb_0: We start from the real-world protocol and assume that the simulator $\mathsf{Sim}_{\mathsf{S}}$ knows S's view.
- Hyb_1: The simulator does not receive the (additive SS of) outputs of all SRT layers, but it generates them by involving the simulator of GPU-Trun with the known divisor $\{d_i\}$ as inputs.
- Hyb_2: The simulator does not receive the (additive SS of) outputs of all ReLU and maxpool layers but involves the simulators of Beaver's trick [3] and GPU-DGK to generate them.
- Hyb_4: The simulator does not receive the (additive SS of) outputs of all linear layers. It constructs the linear functions with $\{M_i\}$ and calls the simulator of the AHE-to-SOS transformation to provide the additive SS output for the linear layers.
- Hyb_5: The simulator has simulated most layers except the input layer. The view originated from the computation of the input layers is an additive SS of the input, and the simulator replaces the SS with a random value from $\mathbb{Z}_q$. Now, the view originated from the interactive computation of GForce for all layers can be simulated without knowing the query x and the result out.

Security Proof of GForce against a Corrupted Client. We prove by hybrid games that the simulated view is indistinguishable from the client's view, which additionally includes the additive SS of all intermediate values of the underlying protocols.

- Hyb_0: We start from the real-world protocol and assume that the simulator $\mathsf{Sim}_{\mathsf{C}}$ knows C's view.
- Hyb_1: The simulator does not receive the (additive SS of) outputs of all SRT, ReLU, and maxpool layers. Instead, it invokes the simulators of GPU-DGK and Beaver's trick to generate the views for ReLU and maxpool layers as Hyb_2 in the security proof for a corrupted server. For the SRT layers, it provides $\{d_i\}$ to the simulator for GPU-Trun, similar to Hyb_1 in the security proof for a corrupted server, to generate the view.
- Hyb_2: The simulator does not receive the (additive SS of) outputs of all linear layers. It treats the linear layers as linear functions and calls the simulator of our AHE-to-SOS transformation, which does not need the input of $\{M_i\}$, to generate the resulting additive SS.