usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

Security Symposium

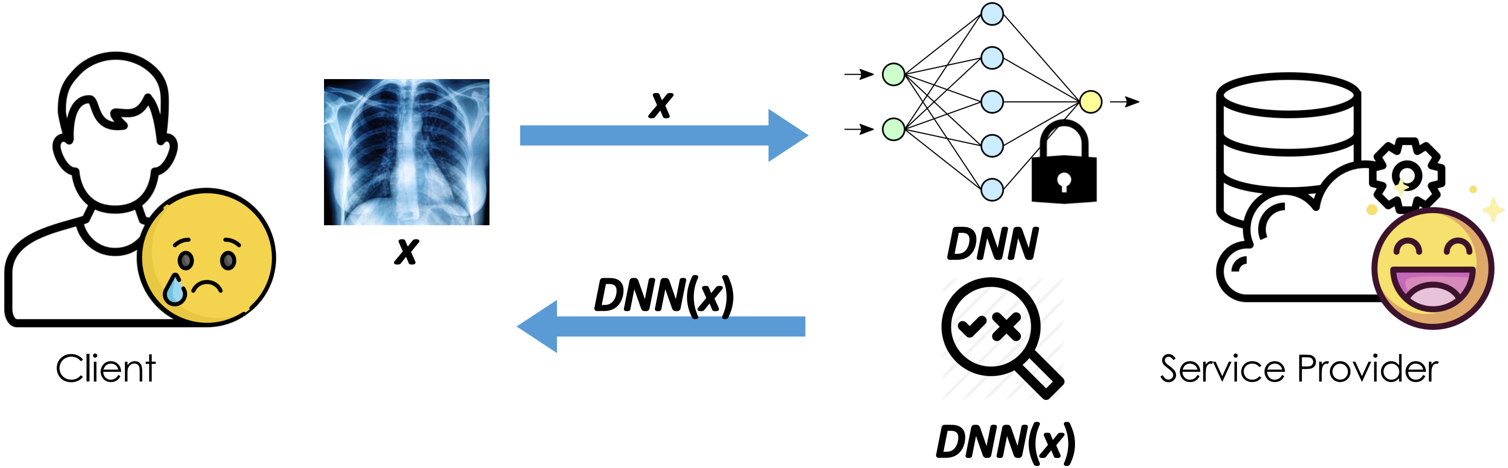
GForce: GPU-Friendly Oblivious and Rapid Neural Network Inference

Lucien K. L. Ng and Sherman S. M. Chow

Department of Information Engineering
Chinese University of Hong Kong (CUHK), Hong Kong

Query Privacy in NN Inference

- Queries in inference can be sensitive
    - Social applications, Medical image analysis, Computer vision, …
- The "natural" way will leak them to the server



x

x

DNN

DNN(x)

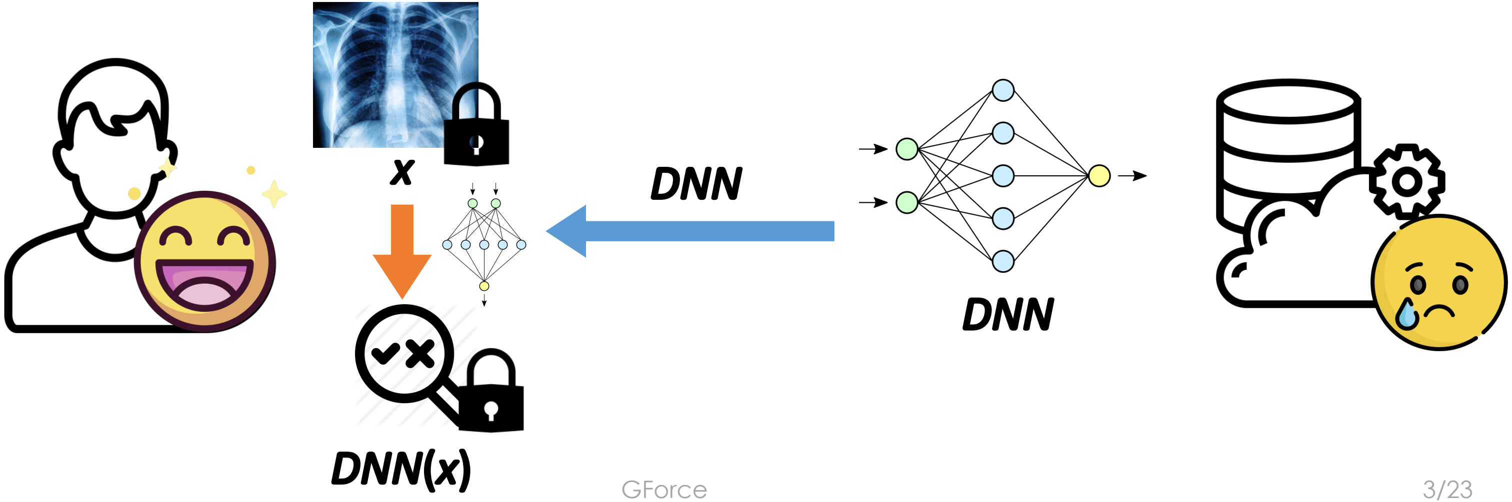
DNN(x)

Client

Service Provider

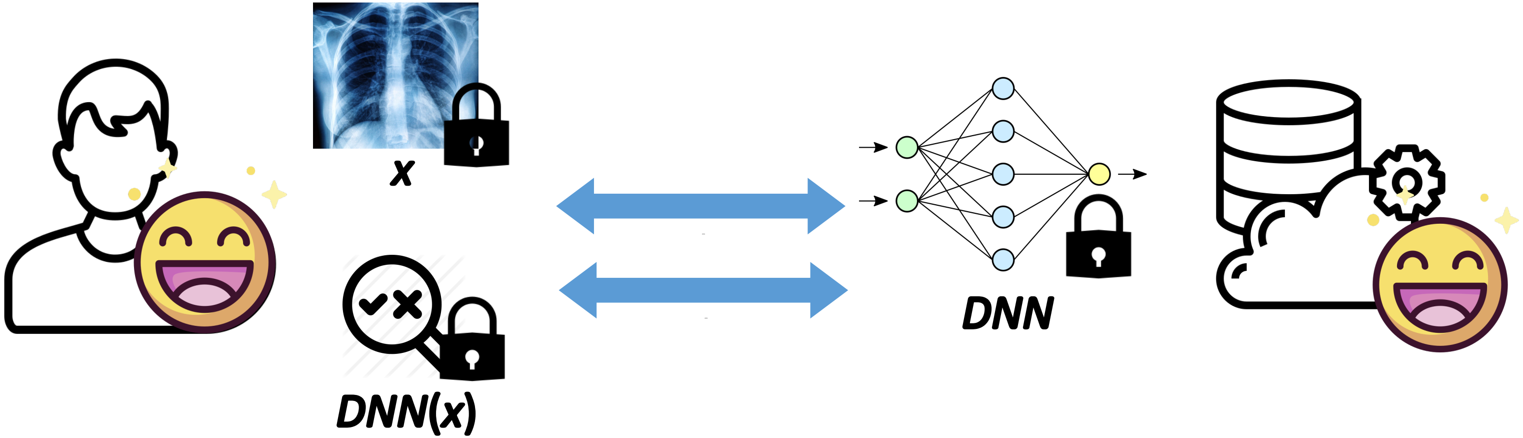
Revealing the model to all clients?

- Local inference well protects the client
  - The model itself is an intellectual property
  - One may reverse-engineer the model to recover training data



Oblivious NN Inference

- The client can learn *DNN(x)* but not *DNN*
- The server cannot learn anything about *x*



GForce

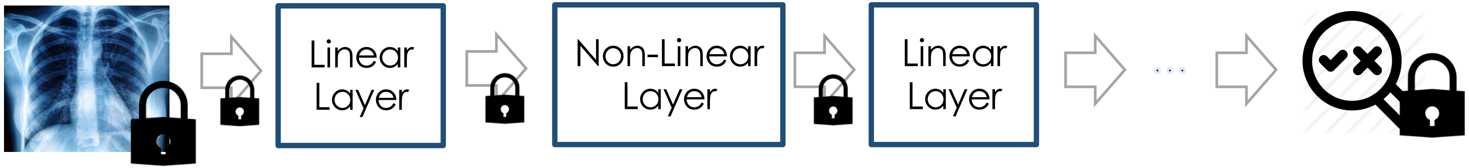
- Oblivious, rapid, and accurate NN Inference

- GForce attains **~73%** in **0.4s** (the *first* for *purely*-crypto solutions)
  - (*e.g.*, no trusted execution environment, no non-colluding server)
  - over CIFAR-100: Image dataset consisting of 100 classes
  - Delphi (prior best [USS20]): **~68%** in **14s** (or ~66% in 2.6s)

- *Spoiler Alert*:
  - I: Make (non-linear) Crypto GPU-friendly
    - "GPU-DGK"
  - II: Tackle the (notorious) issue of Accuracy vs. Bitwidth
    - "SRT" for "SWALP"

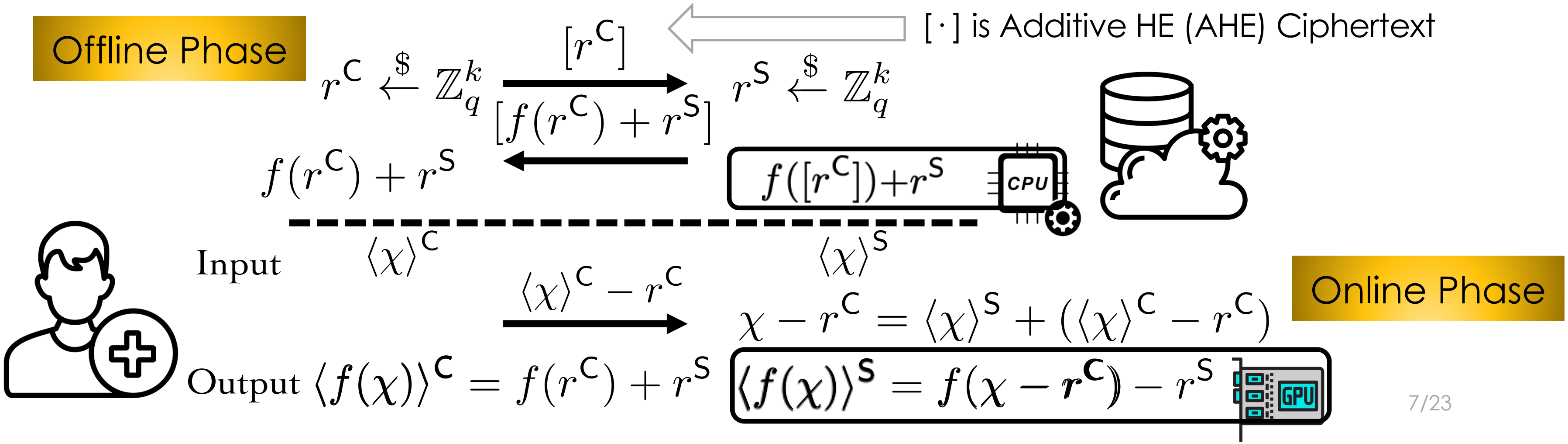
Basic: Dividing a NN

- Treat linear layers and non-linear layers *differently*
  - non-linear: *e.g.*, ReLU, Maxpool
  - linear: *e.g.*, Convolution, Matrix Multiplication



Secure On-/Offline Share Comp.

- To compute a linear function *f*: $f(x) = f(x-r) + f(r)$
  - Offline pre-compute $f(r)$ with (slow) *Homomorphic Encryption* (HE)
  - Online compute $f(x-r)$ in GPU in a batch of *k* (100× faster than CPU)
  - $(x-r, r)$ are like Additive *Secret Share* (SS) of *x*: $\langle x\rangle^{S} + \langle x\rangle^{C} = x \pmod{q}$



Offline Phase

[·] is Additive HE (AHE) Ciphertext

$r^{C} \xleftarrow{\$} \mathbb{Z}_q^k \xrightarrow{[r^{C}]} r^{S} \xleftarrow{\$} \mathbb{Z}_q^k$

$f(r^{C}) + r^{S} \xleftarrow{[f(r^{C}) + r^{S}]} \boxed{f([r^{C}])+r^{S}}$

Input $\langle\chi\rangle^{C}$ $\langle\chi\rangle^{S}$

$\xrightarrow{\langle\chi\rangle^{C} - r^{C}}$ $\chi - r^{C} = \langle\chi\rangle^{S} + (\langle\chi\rangle^{C} - r^{C})$

Online Phase

Output $\langle f(\chi)\rangle^{C} = f(r^{C}) + r^{S}$ $\boxed{\langle f(\chi)\rangle^{S} = f(\chi - r^{C}) - r^{S}}$

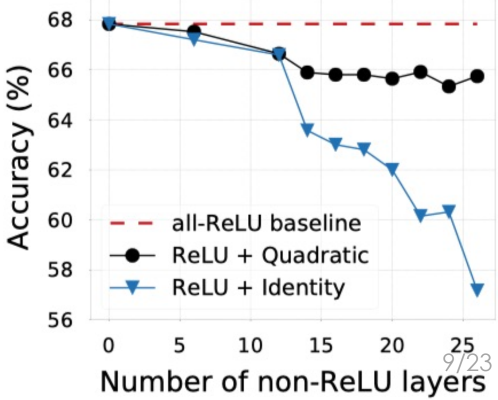
Linear Layers by SOS

- *Secure On-/Offline Share Comp.* (SOS) suits linear layers
  - e.g., used by the prior art Delphi [USS20]
- Operation of a linear layer: $y = W \otimes x$
  - y: output; x: inputs; W: weight (e.g., kernel in a conv. layer)
- The linear layers can be treated as a linear function f_W
  - $f_W(x) = W \otimes x$
  - apply SOS to f_W
- Can we call SOS for non-linear layers?

GPU for Non-Linear Layers?

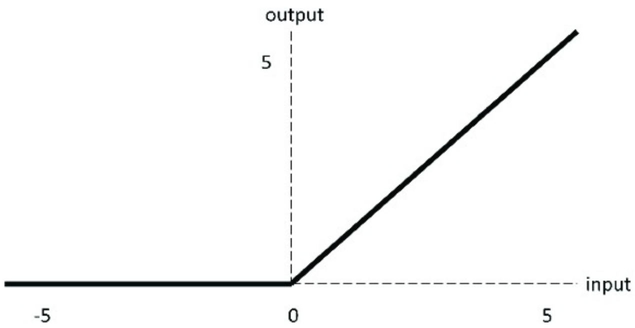
- Non-linear layers need slow *garbled circuit* (GC)
- Delphi replaces some ReLU by quadratic *approximation*
  - Computing x^2 is fast with additive SS and Beaver’s trick

- Problem 1: Approximation ➔ Worse Accuracy
- Problem 2: Maxpool is still using *slow* GC
  - Maxpool: another popular non-linear layer



I: GPU for Non-Linear Layers!

- *Comparison* $(x \leq y)$ is a fundamental operation
  - $\mathrm{ReLU}(x) = \mathrm{Max}(x, 0)$
  - $\mathrm{Maxpool}(\{x\}_{0..3}) = \mathrm{Max}(x_0, x_1, x_2, x_3)$
    - *e.g.*, for a pooling window of size 4
  - $\mathrm{Max}(x, y) = (x \leq y) \cdot (y-x) + x$



ReLU(*x*)

2	2	7	3
9	4	6	1
8	5	2	4
3	1	2	6

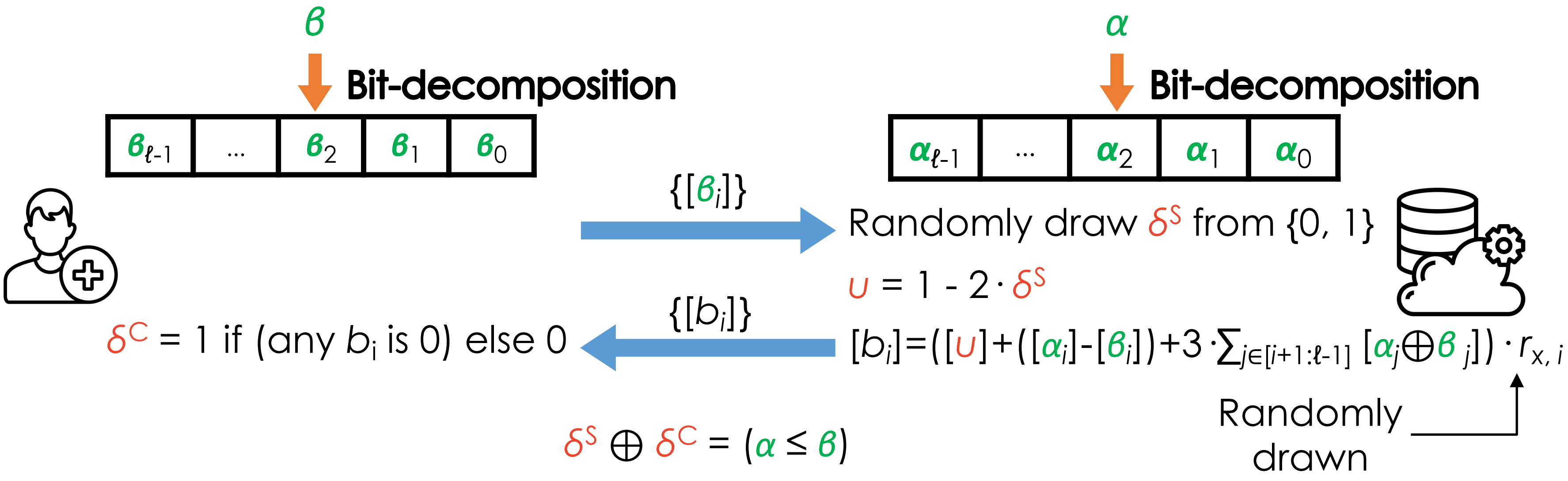
Filter: (2 × 2)
Stride: (2, 2)

9	7
8	6

Maxpool(*x*)

Recap: DGK Protocol

- DGK uses AHE for Comparison
- Each input α or β and get an additive SS of $(\alpha \leq \beta)$



Randomly draw δ^S from {0, 1}

$u = 1 - 2 \cdot \delta^S$

$[b_i]=([u]+([\alpha_i]-[\beta_i])+3 \cdot \sum_{j\in[i+1:\ell-1]} [\alpha_j \oplus \beta_j]) \cdot r_{x,i}$

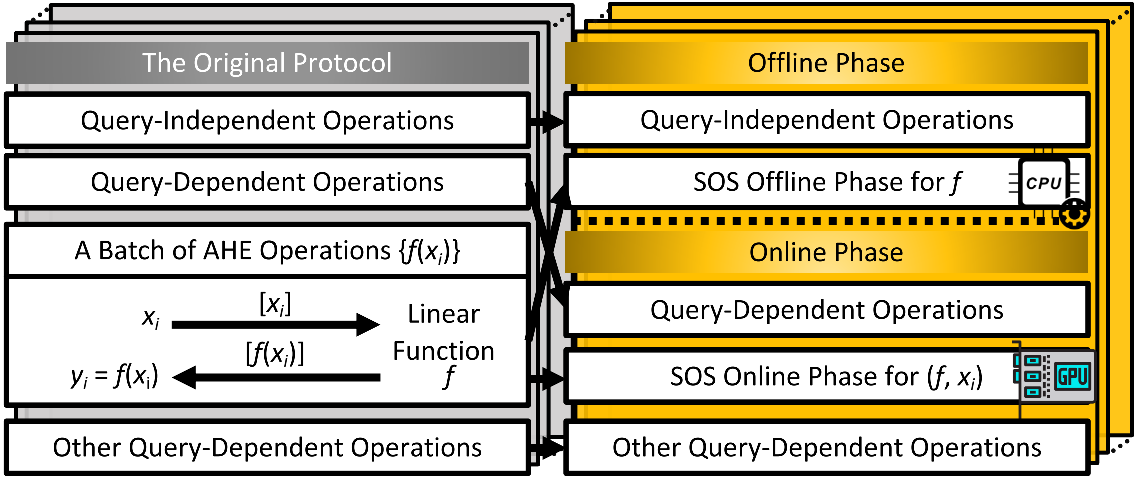
Randomly drawn

$\delta^C = 1$ if (any b_i is 0) else 0

$\delta^S \oplus \delta^C = (\alpha \leq \beta)$

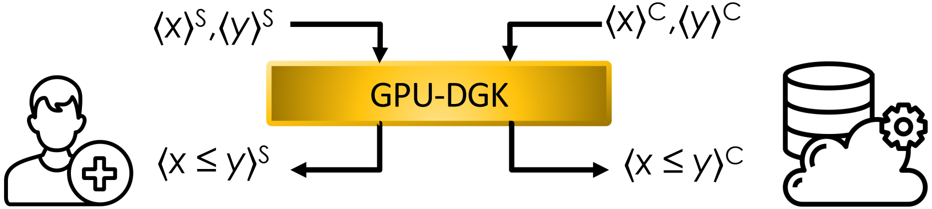
AHE-to-SOS

- Observation: SOS is appliable to many AHE Protocols
- Non-linear “becomes” linear!
- Batch many instances to fully utilize GPU in online phase



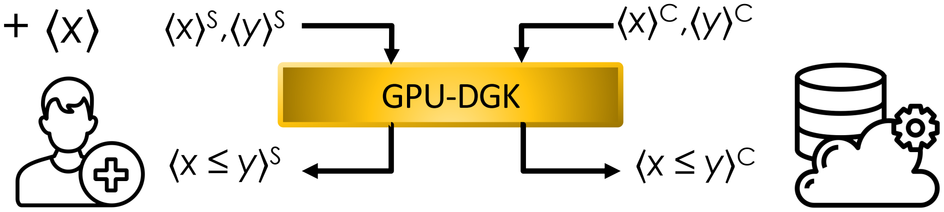
GPU-DGK = AHE-to-SOS + DGK

- Transform the core AHE steps into linear functions
  - $dgk_{i, u, \alpha, r}(\beta) = (u + \alpha_i - \beta_i + 3 \cdot xor_{i, \alpha}(\beta)) \cdot r_{x, i}$ (*xor*() defined in the paper)
  - *i* is the bit position, *u* and *r* are server's randomness
  - but α, β is the ***online*** input of the server/client
- Server can't know/precompute $dgk_{\alpha}()$ in the offline phase
- We devise a trick to "let the server know" α offline
- by deriving β from α and the actual online inputs x and y
  - (More detail in our paper)



GPU-DGK for Non-Linear Layers

- $\langle \mathrm{Max}(x, y)\rangle = \langle x \leq y\rangle \cdot (\langle y\rangle - \langle x\rangle) + \langle x\rangle$
  - Notation: $\langle x\rangle = \{\langle x\rangle^S, \langle x\rangle^C\}$
  - $\mathrm{Max}(x, y) = (x \leq y) \cdot (y-x) + x$
- Max ➔ ReLU and Maxpool
- Better (Online) Performance w/o (GC) approx.!



non-approximate garble circuit approach ([USS18])

Framework	ReLU	Speedup	Maxpool	Speedup
Gazelle	1754.00ms	-	2950.00ms	-
GForce	65.15ms	**27×**	99.02ms	**34**×

Number of input elements = 2^{17}

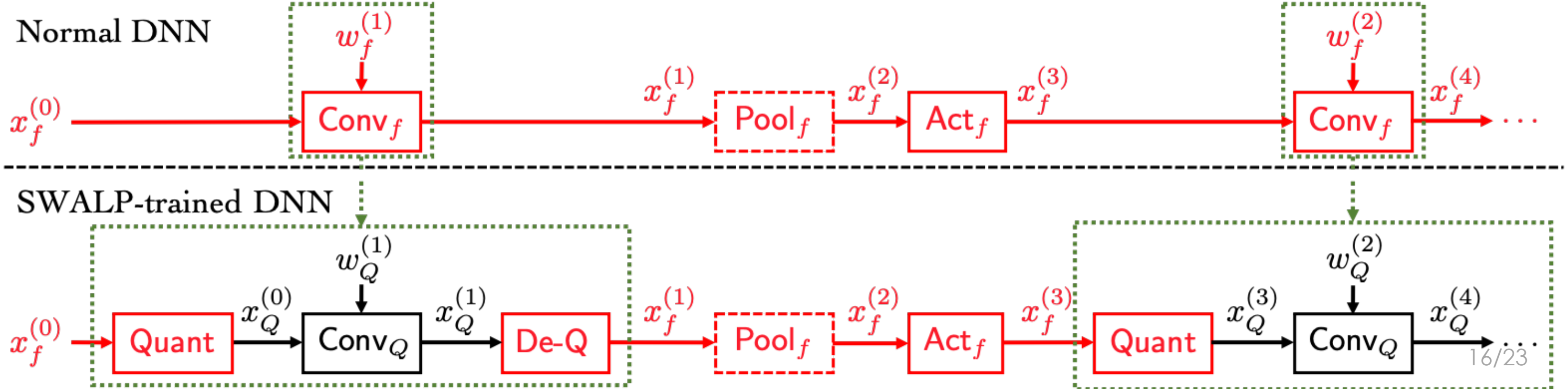
II: Accuracy vs. Bitwidth

- AHE/Additive SS: Operating in $\mathbf{Z}_q$ (integers)
  - Parameters are mostly floating points, w/ *highly dynamic* ranges
    - from 2^{-127} to 2^{127}
  - Need *high-bitwidth* integers to simulate floating points
    - may need integers with 255(=127 + 127 +1) bitwidth

- Small $\mathbf{Z}_q$ (low bitwidth) ➔ Worse Accuracy
  - Error in conversion between floating points and integers
- Large $\mathbf{Z}_q$ (high bitwidth) ➔ Worse Performance
  - GC: *larger* circuit
  - DGK: *more* “bit comparison”: $[b_i]=[a_i]+([x_i]-[y_i])+3\sum_{j\in[i+1:\ell-1]}[x_j\oplus y_j]$
  - GPU has *limited bitwidth* for efficient computation over integers

(De-)Quantizing Linear Layers

- Quantize the NN using SWALP [ICML19]
  - **S**tochastic **W**eight **A**veraging in **L**ow-**P**recision Training
  - almost as good as floating
- Quant(): find maximum ➔ scale up/down ➔ round to int.
- De-Q(): scale up/down



Issues in adopting SWALP

- How to find the maximum (securely and efficiently)?
- How to represent floating points after dequantization?

- How to scale down?
  - Naive division over additive SS *ruins* low-bitwidth NNs
- How to do rounding?
- Experiments over VGG-16 shows:

Dataset	Rounding w/ Proper Scale Down	Naive Division
CIFAR-10	**93.22%**	10.06%
CIFAR-100	**72.83%**	1.03%

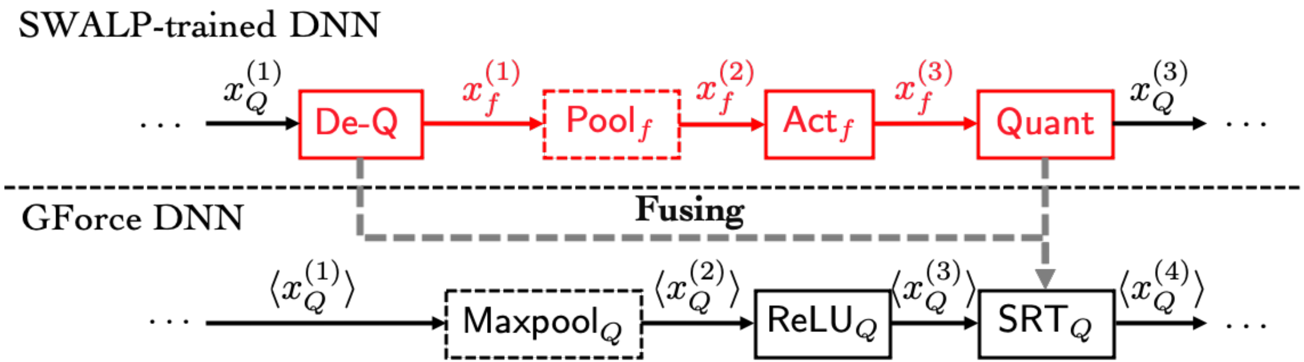
Precomputation & Fusing

- Finding the Maximum: Precompute using training data

- Fusing (De)quantization into just a *division*!
  - De-Q o ReLU o Maxpool o Quant
    = (ReLU o Maxpool) / **d**
  - *d* is computed with the precomputed maximum
  - No floating points now

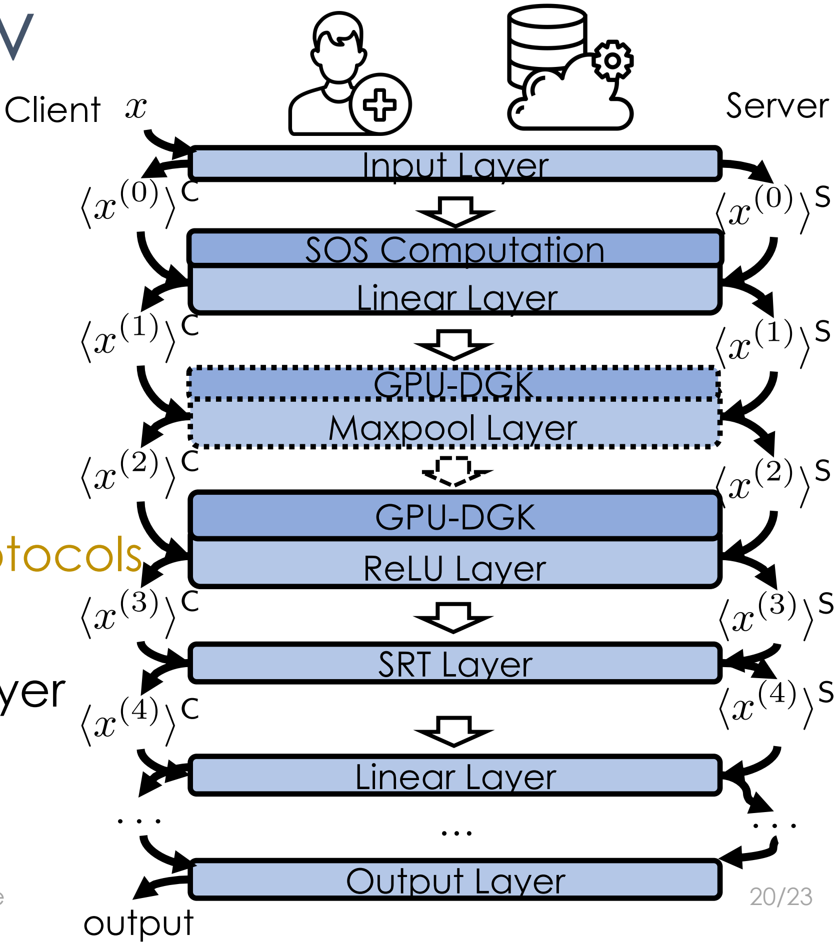
Stochastic **R**ounding **T**runcation

- We form a new *SRT layer* (also utilizing AHE-to-SOS) that
- performs stochastic rounding
- corrects the error in naive division/truncation (“for free”)
    - (More detail in our paper)



End-to-End Workflow

- Setup:
  - Training a NN with SWALP
  - Precompute $\{d_i\}$ for SRT Layers
- Inference:
  - Offline computation with AHE
  - Online: Run our *GPU-friendly* protocols
  - We make *all* layers GPU-friendly
  - They jointly run them layer-by-layer



Client x

Server

Input Layer

$\langle x^{(0)} \rangle^C$ $\langle x^{(0)} \rangle^S$

SOS Computation

Linear Layer

$\langle x^{(1)} \rangle^C$ $\langle x^{(1)} \rangle^S$

GPU-DGK

Maxpool Layer

$\langle x^{(2)} \rangle^C$ $\langle x^{(2)} \rangle^S$

GPU-DGK

ReLU Layer

$\langle x^{(3)} \rangle^C$ $\langle x^{(3)} \rangle^S$

SRT Layer

$\langle x^{(4)} \rangle^C$ $\langle x^{(4)} \rangle^S$

Linear Layer

$\cdots$

Output Layer

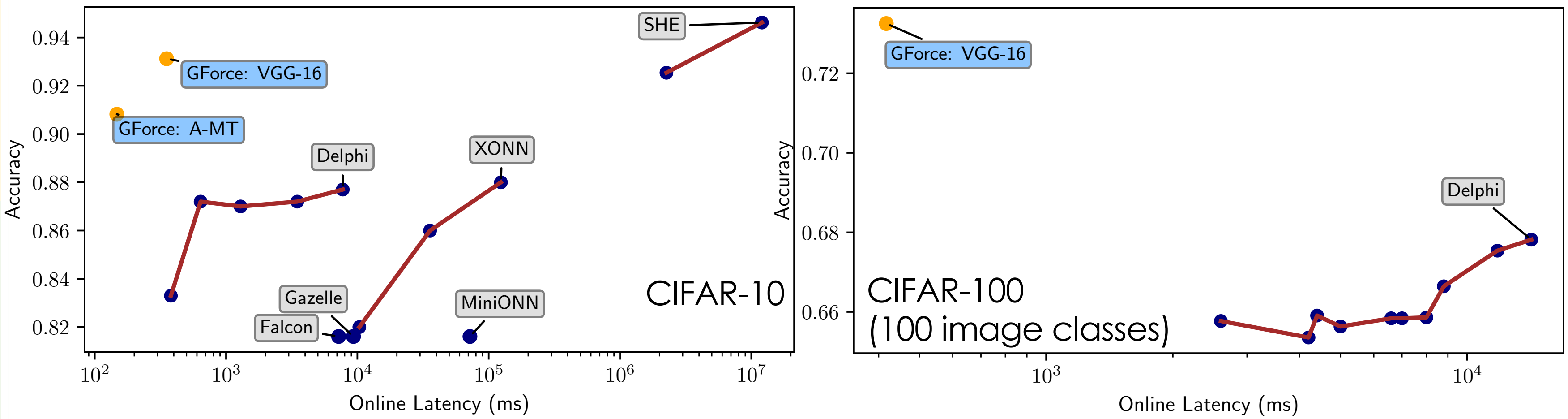
output

Security Analysis

- GForce assumes *semi-honest* client and server
- The client learns
  - *DNN*(x), the query result
- The server learns
  - $\{M_i\}$, the weight (and bias) in linear layers
- Common knowledge/leakage:
  - *DNN* architecture
  - $\{d_i\}$ in SRT Layers (~4 bits for each layer)

Overall Accuracy and Latency

- Shortest (Online) Latency: (CIFAR-10/100: 150/350ms)
- Highest Accuracy in CIFAR-100 (73% vs. 68% of Delphi)



GPU: Nvidia V100 16GB
CPU: Intel Xeon (Skylake) CPUs at 2GHz

Network: Google Cloud (8Gbps & <5ms latency)

Final Remarks

- Utilizing GPU for the *entire* model
- Further applications:
  - Integrating with Delphi
  - Oblivious Decision-Tree Inference (vs. *SS*-based? [NDSS21])
- Code: github.com/Lucieno/gforce-public
  - `SEAL` w/ noise flooding (for AHE) and `PyTorch` (for GPU & NN)
- Also see our GPU-friendly work for training [AAAI21]
  - **G**PU-**O**utsourcing **T**rusted **E**xecution of **N**eural Network Training
- Contact: `{luciengkl, sherman}@ie.cuhk.edu.hk`