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
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 \( x\): \( \langle x \rangle^S + \langle x \rangle^C = x \mod q \)

\[
\begin{align*}
\text{Offline Phase} \\
\begin{align*}
& r^C \gets \mathbb{Z}^k_q \\
& [f(r^C) + r^S] \xrightarrow{\text{CPU}} \\
& f(r^C) + r^S \\
\end{align*}
\end{align*}
\]

Input \( \langle x \rangle^C \)

\[
\begin{align*}
\langle x \rangle^C - r^C \xrightarrow{\text{CPU}} \\
\langle x \rangle^C - r^C \xrightarrow{\text{GPU}} \\
\end{align*}
\]

Output \( \langle f(x) \rangle^C = f(r^C) + r^S \)

\[
\begin{align*}
\langle x \rangle^S & = \langle x \rangle^S + \langle (x)^C - r^C \rangle^C \\
\langle f(x) \rangle^S & = f(x-r^C) - r^S
\end{align*}
\]
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 $\rightarrow$ Worse Accuracy
- Problem 2: Maxpool is still using slow GC
  - Maxpool: another popular non-linear layer
### Comparison

$(x \leq y)$ is a fundamental operation

- $\text{ReLU}(x) = \text{Max}(x, 0)$
- $\text{Maxpool}([x]_{0..3}) = \text{Max}(x_0, x_1, x_2, x_3)$
  - e.g., for a pooling window of size 4
- $\text{Max}(x, y) = (x \leq y) \cdot (y-x) + x$

---

**GPU for Non-Linear Layers!**

$I: \text{GPU for Non-Linear Layers!}$

<table>
<thead>
<tr>
<th>Filter: $(2 \times 2)$</th>
<th>Stride: $(2, 2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Filter Diagram" /></td>
<td><img src="image2" alt="Stride Diagram" /></td>
</tr>
</tbody>
</table>

---

**GForce**
Recap: DGK Protocol

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

\[
\delta^C = 1 \text{ if (any } b_i \text{ is 0) else 0}
\]

\[
\delta^S \oplus \delta^C = (\alpha \leq \beta)
\]
AHE-to-SOS

- Observation: SOS is applicable 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
  - \( dgk_{i, u, \alpha, r}(\beta) = (u + \alpha_i - \beta_i + 3 \cdot \text{xor}_{i, \alpha}(\beta)) \cdot r_{x, i} \) (\text{xor}() defined in the paper)
  - \( i \) is the bit position, \( u \) and \( r \) are server’s randomness
  - but \( \alpha, \beta \) is the \textit{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” \( \alpha \) offline
- by deriving \( \beta \) from \( \alpha \) and the actual online inputs \( x \) and \( y \)
  - (More detail in our paper)
GPU-DGK for Non-Linear Layers

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

<table>
<thead>
<tr>
<th>Framework</th>
<th>ReLU</th>
<th>Speedup</th>
<th>Maxpool</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gazelle</td>
<td>1754.00ms</td>
<td>-</td>
<td>2950.00ms</td>
<td>-</td>
</tr>
<tr>
<td><strong>GForce</strong></td>
<td>65.15ms</td>
<td><strong>27x</strong></td>
<td>99.02ms</td>
<td><strong>34x</strong></td>
</tr>
</tbody>
</table>

Number of input elements = \(2^{17}\)

non-approximate garble circuit approach ([USS18])
II: Accuracy vs. Bitwidth

- AHE/Additive SS: Operating in $\mathbb{Z}_q$ (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 $\mathbb{Z}_q$ (low bitwidth) $\Rightarrow$ Worse Accuracy
  - Error in conversion between floating points and integers

- Large $\mathbb{Z}_q$ (high bitwidth) $\Rightarrow$ 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]
  - Stochastic Weight Averaging in Low-Precision Training
  - almost as good as floating
- Quant(): find maximum ➔ scale up/down ➔ round to int.
- De-Q(): scale up/down

Normal DNN

```
\[ x_f^{(0)} \rightarrow \text{Conv}_f \rightarrow x_f^{(1)} \rightarrow \text{Pool}_f \rightarrow x_f^{(2)} \rightarrow \text{Act}_f \rightarrow x_f^{(3)} \rightarrow \text{Conv}_f \rightarrow x_f^{(4)} \rightarrow \ldots \]
```

SWALP-trained DNN

```
\[ x_f^{(0)} \rightarrow \text{Quant} \rightarrow x_Q^{(0)} \rightarrow \text{Conv}_Q \rightarrow x_Q^{(1)} \rightarrow \text{De-Q} \rightarrow x_f^{(1)} \rightarrow \text{Pool}_f \rightarrow x_f^{(2)} \rightarrow \text{Act}_f \rightarrow x_f^{(3)} \rightarrow \text{Quant} \rightarrow x_Q^{(3)} \rightarrow \text{Conv}_Q \rightarrow x_Q^{(4)} \rightarrow \ldots \]
```

\[
\text{SWALP-trained DNN, with weights scaled down by a factor of 16/23.}
\]
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:

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Rounding w/ Proper Scale Down</th>
<th>Naive Division</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIFAR-10</td>
<td>93.22%</td>
<td>10.06%</td>
</tr>
<tr>
<td>CIFAR-100</td>
<td>72.83%</td>
<td>1.03%</td>
</tr>
</tbody>
</table>
Precomputation & Fusing

- Finding the Maximum: Precompute using training data

- Fusing (De)quantization into just a division!
  - $\text{De-Q o ReLU o Maxpool o Quant} = (\text{ReLU o Maxpool}) / d$
  - $d$ is computed with the precomputed maximum
  - No floating points now
Stochastic Rounding Truncation

- 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

Inference:
- Offline computation with AHE
- Online: Run our GPU-friendly protocols
- We make all layers GPU-friendly
- They jointly run them layer-by-layer
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]
  - GPU-Outsourcing Trusted Execution of Neural Network Training
- Contact: {luciengkl, sherman}@ie.cuhk.edu.hk