PIGEON: A Framework for Private Inference of Neural Networks

Christopher Harth-Kitzerow
Technical University of Munich, BMW Group
christopher.harth-kitzerow@tum.de

Yongqin Wang
University of Southern California
yongqin@usc.edu

Rachit Rajat
University of Southern California
rrajat@usc.edu

Georg Carle
Technical University of Munich
carle@net.in.tum.de

Murali Annavaram
University of Southern California
annavara@usc.edu

Abstract

Privacy-Preserving Machine Learning is one of the most relevant use cases for Secure Multiparty Computation (MPC). While private training of large neural networks such as VGG-16 or ResNet-50 on state-of-the-art datasets such as Imagenet is still out of reach, given the performance overhead of MPC, private inference is starting to achieve practical runtimes. However, we show that in contrast to plaintext machine learning, the usage of GPU acceleration for both linear and non-linear neural network layers is actually counterproductive in PPML and leads to performance and scaling penalties. This can be observed by slow ReLU performance, high GPU memory requirements, and inefficient batch processing in state-of-the-art PPML frameworks, which hinders them from scaling to multiple images per second inference throughput and more than eight images per batch on ImageNet.

To overcome these limitations, we propose PIGEON, an open-source [1] framework for Private Inference of Neural Networks. PIGEON utilizes a novel ABG programming model that switches between Arithmetic vectorization, Bitslicing, and GPU offloading depending on the MPC-specific computation required by each layer.

Compared to the state-of-the-art PPML framework Piranha, PIGEON achieves two orders of magnitude improvements in ReLU throughput, reduces peak GPU memory utilization by one order of magnitude, and scales better with large batch size. This translates to one to two orders of magnitude improvements in throughput for large ImageNet batch sizes (e.g. 192) and more than 70% saturation of a 25 Gbit/s network.

1 Introduction

Over the past years, Machine Learning models have shown prediction capabilities that outperform human experts in various domains [3, 12, 20, 45]. Especially, Deep Learning [26], a subset of Machine Learning that uses neural networks with multiple layers, has shown to be particularly successful in image and speech recognition [18, 25], natural language processing [47], and other domains. Today, Deep Learning models are used in a wide range of applications, such as autonomous driving [14], medical diagnosis [44], and financial trading [37]. These models are trained on large datasets using powerful GPUs, which require significant computational resources. As a result, companies such as OpenAI or Anthropic train general models on large datasets and offer services to clients who want to use these models for inference on their data. This introduces a dilemma: Either companies need to reveal their proprietary model parameters to clients, or clients need to reveal their private data to the company. In practice, clients are often on the short end of the stick, as they have to send their data to company-owned servers where the model is stored securely. The sensitive nature of data used in popular deep learning applications such as images, voice recordings, and medical records makes this a serious privacy concern.

Privacy-Preserving Machine Learning (PPML) [35] aims to overcome this problem by enabling training and inference of machine learning models while keeping model parameters and input data secret. Over the last year, PPML frameworks based on Secure Multiparty Computation (MPC) such as CryptGPU [48] and Piranha [51] introduced GPU acceleration to demonstrate for the first time that PPML can be used for private inference of large convolutional neural networks. However, training these large neural networks on state-of-the-art datasets is still out of reach given the performance overhead of MPC: Based on our benchmarks in ideal network conditions, the state-of-the-art PPML framework Piranha would require more than 7 years to train ten epochs of VGG-16 on the full-size ImageNet dataset. Private Inference, on the other hand, is starting to achieve practical runtimes and throughput: Piranha and CryptGPU reduced the inference throughput of large convolutional neural networks from a few inferences per hour to a few inferences per minute.

Despite these advancements, we show in §7 that state-of-the-art frameworks are currently limited in performance and scalability. Moreover, we show that these limitations are

[1] Code Repository: https://github.com/chart21/hpmpc/

caused by evaluating the whole neural network on the GPU. While this observation might be counter-intuitive given the success of GPU-only computation in plaintext training, it is critical to identify the following differences between plaintext machine learning (ML) and PPML.

1.1 Unique Challenges in PPML

Plaintext ML generally benefits from high parallel processing power and memory bandwidth. Layers that require a large number of dot products, such as convolutional layers, contribute the most to the inference and training runtime. However, PPML faces quite different limitations. In PPML, distributed parties need to communicate with each other to jointly perform inference and training. As non-linear functions such as ReLU require multiple communication rounds and sending large chunks of messages between the parties, the non-linear layers in PPML are typically the key bottleneck for inference latency and throughput [50].

MPC Bottlenecks In realistic deployment scenarios of MPC, the parties are distributed, and bandwidths beyond a few Gbit/s can be hence considered unrealistic. A second consideration is network latency, which can be overcome from an implementation perspective only by amortizing the latency overhead over a large batch of inputs. PPML frameworks should, therefore, achieve a high level of network saturation and support high batch sizes to overcome MPC bottlenecks. In the context of this work, we consider an implementation that achieves 25 Gbit/s of network throughput to be a safe threshold to fully saturate the network bandwidth of any realistic MPC setting. Consequently, any further engineering effort should be invested in supporting large batch sizes. This opens up a dilemma: GPU-based frameworks fail to support large batch sizes due to their highly limited memory while CPU-based frameworks truggle to achieve high throughput. We observe that we can group layers in PPML into three categories, where layers within a category have similar bottlenecks and benefit from the same acceleration techniques. We distinguish between multiplication-based layers, conversion-based layers, and matrix-multiplication-based layers. The layer type indicates which MPC primitive is predominantly used to evaluate a layer and will prove helpful for accelerating its evaluation.

Multiplication-based layers In PPML, average pooling and batch normalization are characterized by independent element-wise multiplications. The key to accelerating these layers lies in the use of large register sizes one the CPU to vectorize multiple inputs of a neural network layer using SIMD instructions. We refer to this acceleration technique as arithmetic vectorization. In §4.1 we show that an efficient CPU implementation can fully saturate 25 Gbit/s when utilizing arithemtic vectorization for these layers.

Conversion-based layers In plaintext machine learning, linear layers such as convolutions and fully connected layers require more GPU memory than non-linear layers and therefore determine the peak GPU memory utilization. In PPML, the opposite is the case. Computing non-linear functions such as Softmax, ReLU, or MaxPool requires share conversion. Share conversion consists of a bit decomposition followed by evaluating a large boolean circuit, which inflates required memory. This overhead is so severe that one key contribution of Piranha [51], the state-of-the-art PPML framework, was to evaluate the boolean circuit in place and reduce GPU memory compared to CryptGPU [48]. Nevertheless, we show in §7 that compared to evaluating only linear layers on the GPU, Piranha still requires one order of magnitude higher peak GPU memory and saturates less than 5% of available network bandwidth in different settings. In §4.2 we show that an efficient CPU implementation can fully saturate any realistic network limit to compute the boolean circuit required by these layers by utilizing Bitslicing.

Matrix-Multiplication-based Layers Convolutions and Fully Connected layers are computationally demanding both in plaintext ML and PPML. As evaluating a dot product requires only communicating a single message between parties in many MPC protocols [32], these layers are most likely to be constrained by computation rather than computation. In §4.3 we show that by using cache optimizations and MPC-specific tweaks, we can saturate up to 5Gbit/s of network bandwidth using the CPU, while in §4.4 we show how to saturate more than 10 Gbit/s using the GPU.

Minimizing data movement between CPU and GPU While keeping all computations on the GPU minimizes CPU/GPU data movement in plaintext ML, the opposite is true in PPML. As parties need to communicate frequently during operations, data from GPU-accelerated layers has to be moved to the CPU and sent over the network to the other parties. Also, data that has been received from other parties needs to be moved from the CPU to the GPU. While GPU to GPU networking exists [43], these solutions require servers to be co-located in the same datacenter which is not a realistic deployment scenario for MPC. Minimizing data movement in PPML, therefore, implies maximizing CPU usage.

1.2 Our Contribution

While state-of-the-arty frameworks [48, 51] utilize a one-fits-all approach and accelerate all layers on the GPU we find that the unique challenges and bottlenecks in PPML require a more targeted approach. We propose an ABG programming model that utilizes Arithmetic vectorization, Bitslicing, and GPU acceleration depending on the PPML layer type. To switch between these techniques efficiently, PIGEON implements efficient CUDA transformations and Bitslicing conver-

sions. We demonstrate that by using the ABG programming model we can get the best of both worlds: High throughput and large batch sizes. We provide the following contributions.

1. PIGEON fully saturates a 25 Gbit/s bandwidth for layers such as Average Pooling and Batch Normalization by utilizing arithmetic vectorization on the CPU (c.f §4.1).
2. PIGEON fully saturates a 25 Gbit/s bandwidth for boolean circuits required by non-linear layers such as MaxPool, Relu, and Argmax by utilizing Bitslicing on the CPU (c.f §4.1). In §7 we show, that this design choice leads to two orders of magnitude higher ReLU throughput than Piranha [51].
3. PIGEON proposes various tweaks to saturate up to 5 Gbit/s of bandwidth for computationally intensive layers such as convolutions on the CPU (c.f. §4.3) and more than 10 Gbit/s on the GPU (c.f. §4.4).
4. PIGEON orchestrates these different acceleration techniques by offering efficient conversion between different computation domains (c.f §4.5). Along with pipelining and networking tweaks (c.f. §5), this allows PIGEON to saturate more than 70% of 25 Gbit/s over the entire end-to-end inference, thus leaving little room for further implementation-related optimizations (c.f §7).
5. By only outsourcing convolutions to the GPU PIGEON reduces peak GPU memory utilization by one order of magnitude and supports 24-96 times higher ImageNet batch sizes than Piranha on the same hardware (c.f §7).
6. PIGEON is modular and protocol-agnostic. Existing models and datasets can be imported from PyTorch directly into PIGEON and we provide implementations of semi-honest three-party computation (3PC) and malicious 4PC protocols [16] out of the box (c.f §6).

These improvements enable us to support large ImageNet batch sizes (e.g. 192) for the first time in MPC-based PPML and consistently improve Piranhas throughput for ImageNet and CIFAR-10 inferences by one to two orders of magnitude. To increase the accessibility of PPML frameworks, we also provide a CPU-only version that achieves runtimes comparable to Piranha for ImageNet inferences while utilizing only a single CPU core. When utilizing multiple cores, the CPU-only version even improves on Piranhas throughput by one to two orders of magnitude. Our results are consistent for different MPC settings and ring sizes, hence enabling us to achieve Imagenet throughput beyond one image per second in all these scenarios.

2 Related Work

Several MPC frameworks have been developed that support private inference of machine learning models. Most of these frameworks are based on additive secret sharing [11] and are typically deployed in the multi-party (2PC,3PC, or 4PC) settings. The semi-honest 3PC and malicous 4PC settings tolerating up to one corruption are of particular relevance as they are characterized by an input-independent preprocessing phase that is less expensive than the input-dependent online phase. By utilizing the outsourced computation model [10], any number of input parties can secretly share their inputs to a set of non-colluding computation nodes that carry out the 3PC or 4PC and transfer the final output shares to the parties supposed to learn the result of the computation. This model is suitable for private inference-as-a-service solutions, that similar to their plaintext equivalents handle multiple independent client inference requests in parallel.

Recently, frameworks based on Function Secret Sharing [4] have also shown impressive results, surpassing the performance of frameworks based on additive secret sharing [15,19]. However, these evaluations do not consider the end-to-end performance due to the expensive preprocessing phase of FSS. FSS-based frameworks typically fall back on assuming that all preprocessing material is provided by a trusted dealer and is already stored in the local filesystem or even RAM of each party. While the time of the preprocessing phase can be ignored for low arrival rates of private inference requests, Garimella et al. [13] show that under realistic assumptions, PPML frameworks with high preprocessing costs may have to wait for the entire preprocessing phase to finish before starting the online phase thus limiting the scalability of these approaches in practice. Given these limitations, we focus on honest-majority protocols based on additive secret sharing in this work and consider the full end-to-end performance of PPML frameworks including both the preprocessing phase and the online phase. Note that we provide an overview of models and dataset commonly used in PPML in §A.

MPC-based PPML frameworks Early work on PPML goes back as far as 2006 [2] but the first training of a deep learning model on MNIST was only achieved in 2017 by SecureML [33]. SecureML set a standard for PPML frameworks by utilizing additive secret sharing for linear layers and Yao's garbled circuit protocol [52] for non-linear layers along with efficient transformation between these sharing types. Over the years, other PPML frameworks picked up on this idea and improved PPML based on secret sharing mainly from the protocol side. ABY3 [32] focused on an honest-majority 3PC setting and proposed efficient conversion from arithmetic to boolean secret sharing and Yao's garbled circuits. The high performance and efficient share conversion in the honest-majority setting sparked the interest of several other PPML frameworks in the 3PC and 4PC settings [5,7–9,22,23,41].

CryptGPU [48] first broke the trend of optimizing PPML mainly from a protocol perspective but instead proposed solely software and hardware optimizations to improve PPML performance using GPU acceleration. This design choice led to 2-8 times performance improvements over CPU-based

frameworks for private inference of large neural networks. CryptGPU implemented a wrapper for the popular Machine Learning framework PyTorch [39] that allowed for easy integration of existing models and datasets but introduced some trade-offs by using floating point CUDA [1] kernels for fixed point computation. Piranha [51] improved on CryptGPU's performance by utilizing NVIDIA's CUTLASS [36] library in C++ which provides native integer kernels for fixed point computations. This led to a four times performance improvement over CryptGPU for private inference of VGG16.

In 2023 an SOK on PPML [35] was published that studied 53 PPML frameworks and identified Piranha as the fastest PPML framework to date. Interestingly, with its focus on software and hardware optimization but rather simple protocol design, Piranha is able to outperform PPML frameworks that utilize more efficient underlying MPC protocols than Piranha [7, 8, 23]. Also, in the SOK, Piranhas 3PC implementation of Falcon [49] achieved higher throughput on CIFAR-10 than any other cryptographic PPML framework including works that utilize Homomorphic Encryption. These prior results motivate further research into software and hardware optimizations for MPC-based PPML frameworks.

3 Background: Privacy Preserving Machine Learning based on MPC

Privacy-preserving training and inference can be implemented using MPC with a small set of primitives. Similar to other PPML frameworks [48, 51], we focus on MPC protocols based on linear secret sharing over a ring $\mathbb{Z}_{2^\ell}$. In line with existing work, we assume real numbers are approximated using Fixed-Point Arithmetic (FPA) [6, 34] and mixed circuits [8, 32, 40] are used to evaluate comparisons.

3.1 MPC Primitives

We provide an overview of secret-sharing-based MPC and the minimal set of primitives required to support PPML.

MPC Notations We use $\mathcal{P}$ to denote the set of parties and P_i to denote the ith party carrying out the computation. We use P_I to denote a party submitting inputs to $\mathcal{P}$ and P_O to denote a party receiving output from $\mathcal{P}$. Note that thanks to the outsourced computation model [10], P_I and P_O are not required to participate in the computation. We denote a linear secret share of a secret value x by $[\![x]\!]$ where x_i is the secret share held by $P_i \in P$. A linear secret sharing has the property that an individual secret reveals nothing about x but there exists a threshold t such that holding t individual shares of $[\![x]\!]$ allows to compute x using a linear combination of the shares.

Secret Sharing (Π_{Sh}) and Reconstruction (Π_{Rec}) Let x be a secret held by P_I. For each party $P_i \in \mathcal{P}$, P_I computes x_i and sends it to P_i. $\mathcal{P}$ then holds $[\![x]\!]$. To reconstruct x, each party $P_i \in \mathcal{P}$, sends x_i to P_O. P_O now holds all shares to compute x.

Addition (Π_{Add}) and Multiplication by constants (Π_{CMult}) Given public constants α, β, γ and secret-shares $[\![x]\!], [\![y]\!]$, parties can locally compute the shares of $[\![\alpha x + \beta y + \gamma]\!]$.

Multiplication (Π_{Mult}) and Matrix Multiplication (Π_{MatMul}) Given two secret-shares $[\![x]\!], [\![y]\!]$, parties can interactively compute shares of $[\![z]\!] = [\![x]\!] \cdot [\![y]\!]$. Π_{Mult} typically requires parties to send $O(1)$ ring elements between each other in one communication round. Naively evaluating a matrix multiplication with Π_{Mult}would require $O(mnk)$ local operations and exchange of $O(mnk)$ ring elements between the parties given the input dimensions $m \times k$ and $k \times n$ of two input matrices. However, optimizations [32] allow evaluating a matrix multiplication with $O(mnk)$ local operations but only $O(mn)$ ring elements.

Sign Bit Extraction (Π_{BitExt}) Given the arithmetic sharing of $x \in \mathbb{Z}_{2^\ell}$, Π_{BitExt} generates a boolean sharing of the sign bit $b \in \mathbb{Z}_{2^1}$ of x. The most common approach to evaluate this conversion includes computing a parallel prefix adder in the boolean domain, requiring $\mathcal{P}$ to exchange $O(l \cdot log_2 l)$ boolean values in $O(log_2 n)$ communication rounds. Note that exchanging $O(l \cdot log_2 l)$ boolean values is equivalent to exchanging $O(log_2 l)$ ring elements in $\mathbb{Z}_{2^\ell}$.

Bit to Arithmetic Conversion (Π_{Bit2A}) Given the boolean sharing of a shared bit $[\![\mathrm{b}]\!] \in \mathbb{Z}_{2^1}$, the protocol generates the arithmetic equivalent shares $[\![\mathrm{b}]\!] \in \mathbb{Z}_{2^\ell}$. The most common approach to evaluate this conversion requires computing an arithmetic XOR, requiring $\mathcal{P}$ to exchange $O(1)$ ring elements in $O(1)$ communication rounds.

Truncation (Π_{Trunc}) Protocols using Fixed-Point Arithmetic require truncation to prevent overflows during computation and maintain precision [6]. For a share $[\![x]\!]$, Π_{Trunc} outputs its truncated version $([\![x]\!])^t = \lfloor \frac{[\![x]\!]}{2^t} \rfloor$. Here, t denotes the number of fractional bits in the FPA representation.

3.2 Evaluating Neural Networks using MPC

The previously described MPC primitives are sufficient to evaluate common neural network layers. We denote by X^i, Y^i, and W^i the input, output, and weight matrices of the i-th Layer respectively.

Obtaining model parameters and data The parties holding the model weights and the parties holding data in plaintext locally convert their respective inputs X^0 and W to fixed point values and use Π_{SH} to secretly share them among $\mathcal{P}$.

3.2.1 Matrix-Multiplication-based Layers

Layers such as Convolutional and Fully Connected layers require parties to evaluate a large set of scalar multiplications and additions. These layers are characterized by their high computational complexity.

Convolutions and Fully Connected Layers Fully Connected layers can be evaluated with Π_{MatMul} with the input matrix X^i having a row size of 1. To evaluate convolutions, parties locally perform an im2col transformation on their shares to obtain $[\![\hat{X}^i]\!], [\![\hat{W}^i]\!]$, followed by Π_{MatMul} to obtain output matrix Y^i.

3.2.2 Conversion-based Layers

Conversion-based layers require parties to convert between arithmetic and boolean shares to extract the sign bit of a value during computation. These layers are characterized by their high communication and memory overhead caused by the large boolean circuit evaluated as part of Π_{BitExt}.

DReLU and ReLU DReLU outputs 0 for all negative values and 1 for all positive values in X^i. To compute a DReLU layer, $\mathcal{P}$ use Π_{BitExt} to extract the sign bit $[\![b^j]\!] \in \mathbb{Z}_2$ of all individual shares $[\![x^j]\!] \in [\![X^i]\!]$ They, then use Π_{Bit2A} to obtain $[\![b^j]\!] \in \mathbb{Z}_{2^\ell}$. Finally, for each $y^j \in Y^i$, they set $[\![y^j]\!] = [\![(1-b^j)]\!]$. ReLU outputs $\mathsf{max}(0, x^j)$ for each $x^j \in X^i$. To evaluate a ReLU layer, $\mathcal{P}$ compute $Y^i = DReLU(X^i) \cdot X^i$.

MaxPool and Softmax MaxPool requires parties to obtain the maximum of adjacent values in X^i. The maximum of k elements can be computed using k pair-wise max operations along a tree of height $log(k)$. The pair-wise maximum of two elements $[\![x^a]\!], [\![x^b]\!]$ can be computed as $DReLU([\![x^a]\!] - [\![x^b]\!]) \cdot ([\![x^a]\!] - [\![x^b]\!]) + [\![x^b]\!]$. During inference, Softmax can be replaced by ArgMax, since parties are only interested in the index of the maximum value to obtain the final inference prediction. To compute ArgMax, parties use a similar tree-based procedure as utilized in MaxPool. In some cases, it might even be favorable to skip the Argmax layer to reveal the probabilities of each class.

3.2.3 Multiplication-based Layers

Multiplication-based layers require parties to evaluate element-wise multiplications and additions. These layers are characterized by neither high computational nor high communication complexity.

Average Pooling : Average Pooling computes the average of adjacent values in $[\![X^i]\!]$ using a public denominator d. To avoid division, each party locally computes $\hat{d} = \frac{1}{d}$ and converts the result to FPA. The average of a vector $[\![\vec{x}]\!]$ can then be computed as $(\sum_{j=0}^{d} [\![x^j]\!]) \cdot [\![\hat{d}]\!]$ followed by truncation.

Batch Normalization Batch Normalization computes $Y^i = \frac{X^i - \mu}{\sqrt{\sigma^2+\varepsilon}} \cdot \gamma + \beta$. Note that the parameters $\mu, \sigma, \gamma, \beta$ are model parameters obtained during training, and ε is a small public constant to avoid division by zero. Thus, during inference, the party holding the model parameters locally computes $\hat{\sigma} = \gamma \cdot \frac{1}{\sqrt{\sigma^2+\varepsilon}}$ and shares it along with μ and β among the parties. Using these shares, the parties can compute output $[\![Y^i]\!] = ([\![X^i]\!] - [\![\mu]\!]) \cdot [\![\hat{\sigma}]\!] + [\![\beta]\!]$.

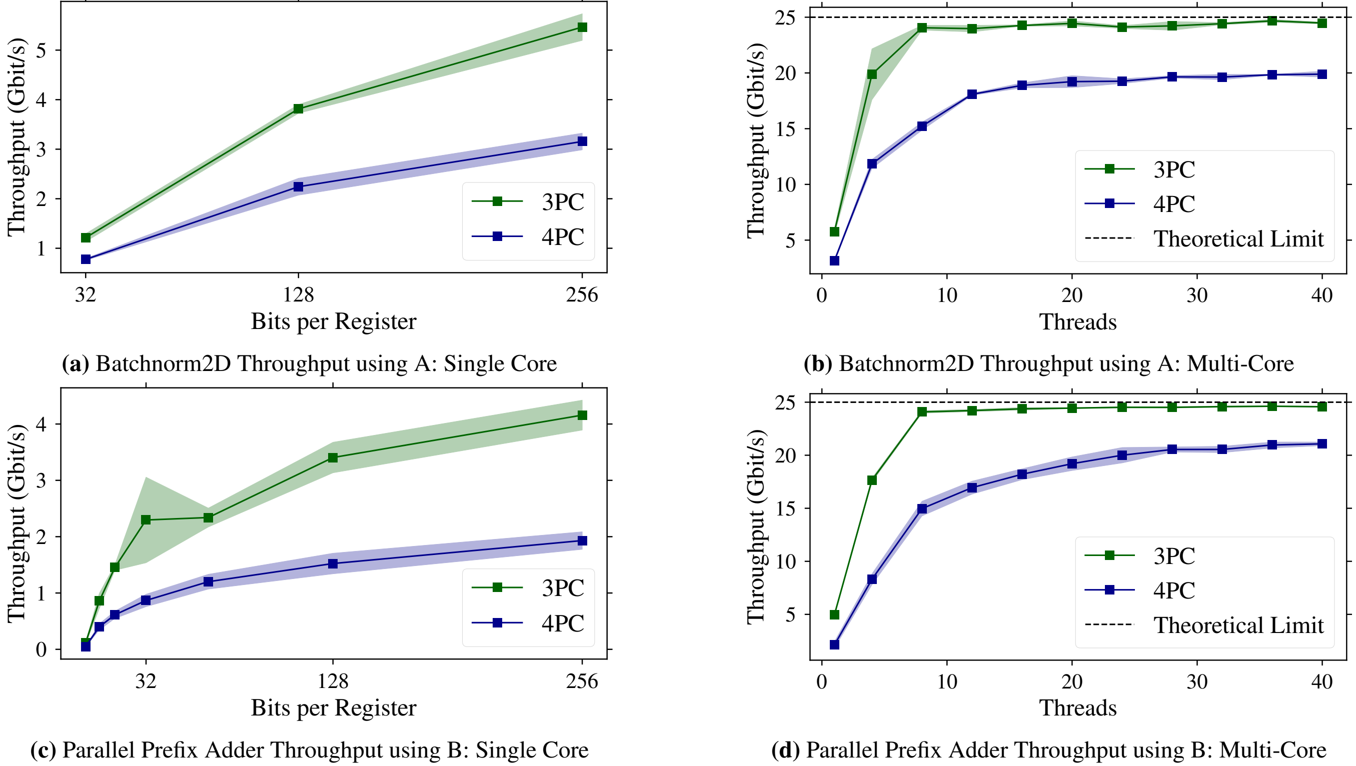
4 The ABG Programming Model

PIGEON uses a novel ABG programming model to overcome system challenges in PPML. The ABG programming model utilizes Arithmetic vectorization to accelerate multiplication-based layers, Bitslicing to accelerate conversion-based layers, and GPU offloading to accelerate matrix-multiplication-based layers. We show in this section that each of these techniques suffices to saturate most of the network even in ideal network settings with 25 Gbit/s network bandwidth. In the following sections all evaluation are based on 3 to 4 nodes equipped with a 32-Core AMD EPYC CPU and a 24GB NVIDIA L4 GPU.

4.1 Accelerating Multiplication-based Layers using Arithmetic Vectorization

Multiplication-based layers, such as Average Pooling and Batch Normalization, consist of large batches of element-wise arithmetic operations on secret shares. Modern processors often provide SIMD (single instruction, multiple data) instructions to allow a CPU to simultaneously perform the same arithmetic operations on a vector of k-bit elements. For example, x86's AVX-512 instructions can process 16 64-bit integers in parallel in one cycle, thus significantly improving the throughput of large batches of element-wise operations. PIGEON utilizes SIMD instructions like SSE, AVX-2, and AVX-512 to achieve arithmetic vectorization. In §C, Figure 6 illustrates this vectorization process.

As the thousands or even millions of independent element-wise operations are performed when evaluating multiplication-based layers, these layers are ideal candidates for arithmetic vectorization. To demonstrate this, Figure 1a shows how vectorization improves the throughput of Batch Normalization by up to five times on a single core, and Figure 1b shows how this approach scales with the number of cores. The results demonstrate that PIGEON can saturate nearly 100% of the available 25 Gbit/s network bandwidth for multiplication-based layers.



(a) Batchnorm2D Throughput using A: Single Core

(b) Batchnorm2D Throughput using A: Multi-Core

(c) Parallel Prefix Adder Throughput using B: Single Core

(d) Parallel Prefix Adder Throughput using B: Multi-Core

Figure 1: Accelerating PPML layers using Arithmetic Vectorization and Bitslicing.

4.2 Accelerating Conversion-based Layers using Bitslicing

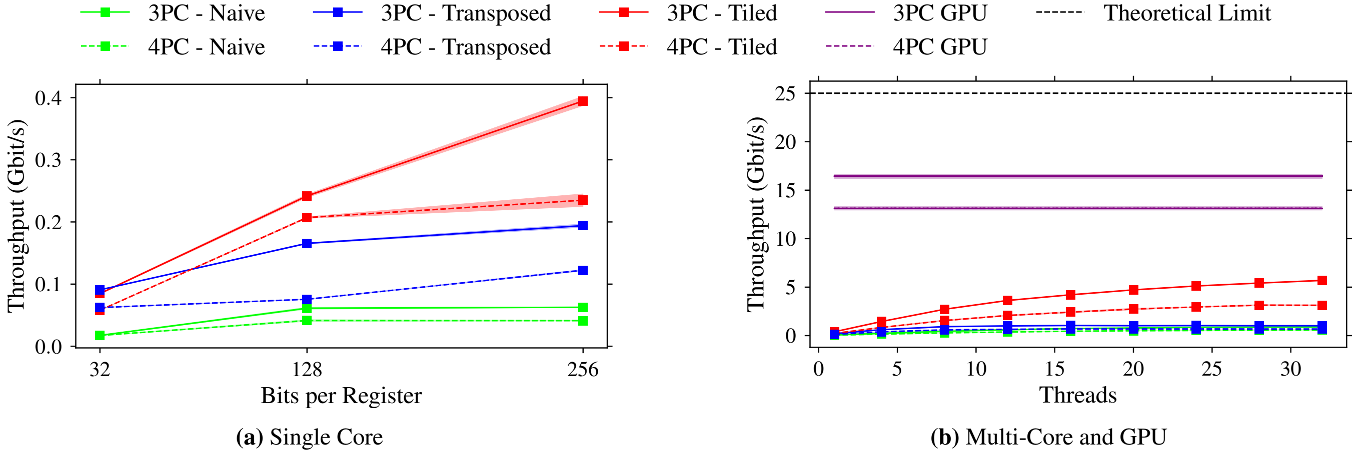
Conversion-based layers are bottlenecks by addition circuit evaluated in the boolean domain during Π_{BitExt} which consists of hundreds of boolean gates per input. To make things worse, inputs to those boolean gates are 1-bit variables, and it is wasteful to evaluate boolean gates input by input on the GPU or CPU with one instruction per gate on the smallest available register. Instead, one can utilize Bitslicing to process a batch of multiple boolean inputs packed in a large register using only a single bit-wise instruction. The key idea of Bitslicing is that computing a bit-wise logical operation on an m-bit register effectively works like m parallel boolean conjunctions, each processing a single bit [30]. For example, instead of performing a single 1-bit XOR operation across two bits, one could perform a single 32-bit bitwise XOR operation on a 32-bit register that stores a batch of individual bits. Furthermore, one can exploit hardware instruction sets such as AVX-2 to pack 256 bits and compute 256 XOR operations in parallel. In §C, Figure 6 shows how Bitslicing can replace multiple independent CPU instructions with a single one.

PIGEON supports Bitslicing using standard register sizes up to 64 bit but also larger register sizes provided by the SSE, AVX-2, and AVX-512 instruction sets. Figure 1c shows how Bitslicing improves the throughput of the Parallel Prefix Adder required by Π_{BitExt} by up to one order of magnitude on a single core, and Figure 1d shows how this throughput scales with the number of cores. The results demonstrate that PIGEON can saturate nearly 100% of the network bandwidth for conversion-based layers.

4.3 Accelerating Matrix-Multiplication based Layers using the CPU

Matrix-multiplication-based layers require parties to evaluate a large number of scalar multiplications and additions. Thanks to efficient Matrix Multiplication primitives provided by most MPC protocols, the communication complexity of these layers grows only quadratically with the input size. However, the computational complexity of matrix multiplication primitives grows cubically with the input size. Hence, it is challenging to saturate the network bandwidth for matrix-multiplication-based layers. First, we show how PIGEON's CPU-only variant evaluates matrix multiplications using only the CPU before moving to GPU accelerated approaches.

Reducing Cache Misses In general, matrix multiplications suffer from potential cache misses and inefficient memory access patterns [38]. In §D, algorithm 1 shows three different algorithms to evaluate a matrix multiplication with different memory access patterns. Line 5 of the Naive MatMult algo-



(a) Single Core

(b) Multi-Core and GPU

Figure 2: Conv2D Throughput using Vectorization, Cache Optimizations, and GPU Acceleration

Input size: $224 \times 224 \times 64$, 3×3 kernel, 1×1 stride, 1×1 padding, 64 filters

rithm shown in Algorithm 1 demonstrates this inefficiency by accessing the elements of matrix B in a row-wise fashion thus skipping over the cache line of the CPU. Line 6 of the Transposed MatMul algorithm shown in Algorithm 1 demonstrates how transposing matrix B first, can improve the memory access pattern by accessing the elements of the transposed matrix in a column-wise fashion. Finally, the Tiled MatMul algorithm shown in Algorithm 1 demonstrates how accessing all matrices in a block-wise fashion (tiles) can further improve the memory access pattern. When picking a tile size according to the cache size of the CPU, the Tiled MatMul algorithm minimizes the number of cache misses.

Reducing Redundancy Existing frameworks typically evaluate a matrix multiplication on secret shares by replacing the local multiplication and addition operators of a secure multiplication protocol Π_{Mult} with their matrix multiplication and addition counterparts. Following this approach, each party needs to iterate over the same memory locations multiple times and thus introduces redundancy. To reduce this redundancy, PIGEON instead evaluates a secure matrix multiplication by a series of local dot products where the local multiplication operator is replaced with a fused multiplication that computes the individual local operations required by Π_{Mult} before communicating in a single pass.

To illustrate the different approaches, consider a party that holds secret shares A_1, A_2 of matrix $[\![A]\!]$ and B_1, B_2 of matrix $[\![B]\!]$. Suppose the party's matrix multiplication protocol requires it to compute $T^1 = A_1B_2 + A_2B_1$ and $T^2 = A_1B_1$, where T^1 and T^2 are temporary matrices used later to compute messages and the final result. Instead of computing three individual matrix multiplications, the party can compute a single matrix multiplication but calculate $T^1[i][j] \mathrel{+}= A[i][k] \cdot B[k][j] + B[i][k] \cdot A[k][j]$ and $T^2[i][j] \mathrel{+}= A[i][k] \cdot B[k][j]$ whenever the plaintext matrix multiplication would calculate $C[i][j] \mathrel{+}= A[i][k] \cdot B[k][j]$ as part of local dot product computation.

Interleaving Communication with Computation Dissecting matrix multiplications into individual dot products also allows PIGEON to interleave communication with computation. In the traditional approach, matrices are first multiplied, then masked, and exchanged with each other to compute the final result. Thus, the communication channels are idle until all local matrix multiplications have been computed. In contrast, PIGEON can compute local dot products of the matrices and immediately mask and exchange the result element-wise with the other parties. Lines 7,8, and 16 of the Naive, Transposed, and Tiled MatMul algorithms shown in Algorithm 1 show that communication for an individual matrix element $C[i][j]$ is handled immediately after an individual dot product has been computed in contrast to waiting for the entire matrix multiplication to finish.

Figure 2 shows the throughput that the different approaches achieve for a Convolutional layer using only the CPU. For the parameters of the Convolutional layer, we use the largest Convolutional layer of VGG-16 on ImageNet. The results demonstrate that the tiled matrix multiplication approach can achieve a throughput of more than 5 Gbit/s per second when combined with PIGEON's other optimizations.

4.4 Accelerating Matrix-Multiplication based Layers using the GPU

While the earlier-mentioned CPU-based techniques suffice to saturate most network settings, scaling beyond 5Gbit/s of convolution throughput requires GPU acceleration. Similar to Piranha [51], PIGEON uses NVIDIA's CUTLASS library [36] which provides highly optimized templated CUDA kernels for

matrix multiplications and convolutions. PIGEON provides these templated kernels as standalone executables for integers ranging from 16 to 64 bits which might be of independent interest to researchers aiming to integrate these into their projects. PIGEON also supports Multi-GPU setups where each individual GPU is assigned to a separate batch of inputs during a forward pass.

Even though we cannot utilize our individual dot product optimization on the GPU, since CUDA does not support custom operators and datatypes, we can still interleave communication with computation but in a less fine-grained fashion. To do so, we split up a batch of k matrices into n batches of size k/n and interleave the computation of a batch with the communication of the previous batches. We will later show that this approach is especially suitable for our ABG programming model. Figure 2b shows that by using GPU acceleration, PIGEON can saturate more than half of our 25 Gbit/s network bandwidth for large convolutions.

4.5 Switching between Acceleration Techniques

While we have shown that each of the acceleration techniques achieves high throughput, these advantages only translate into practice if converting between techniques is efficient. PIGEON requires parties to switch between arithmetic Vectorization and Bitslicing during share conversion and between arithmetic vectorization and GPU acceleration during matrix multiplications.

Switching between Arithmetic Vectorization and Bitslicing Efficiently converting k ℓ-bit integers stored in a single register of size $r = k \cdot \ell$ from a vectorized to a bitslicded representation requires accumulating at least ℓ of these r-bit variables first. This way, we can obtain a new set of ℓ r-bit registers where register i contains the i-th bit of all k integers. Naively, this transposition would be performed bit by bit in four cycles per bit [29]. However, USUBA observed that this transposition can be optimized using a recursive matrix transposition approach proposed by Knuth [21]. Using this insight, USUBA reduced the number of cycles to transpose 512 AVX-512 variables to 0.09 cycles per bit [31] and provides an open-source version of these transpositions for various hardware architectures and block sizes.

Figure 3 shows an exemplary share conversion protocol including a transformation from arithmetic vectorization to Bitslicing. Share conversion of a value a from the arithmetic to the boolean domain as proposed by ABY3 [32] requires the parties to hold an arithmetic sharing of $[\![a]\!]^A = a_1 + a_2$ where a subset of parties $\mathcal{P}_\phi$ holds a_1 and the remaining parties $\overline{\mathcal{P}\phi}$ hold a_2. To perform the conversion, the parties create boolean sharings $[\![a_1^B]\!]$ and $[\![a_2^B]\!]$ followed by a boolean adder to obtain the final boolean sharing $[\![a]\!]^B$. The figure shows that each party can locally accumulate and transpose a vector of shares before communicating with the other parties to ensure that all parties obtain a bitsliced representation of $\vec{a}$. Protocol Π_{BitExt} follows the same procedure as Π_{A2B} but uses a carry adder in step 3 to only compute the sign bit of the addition.

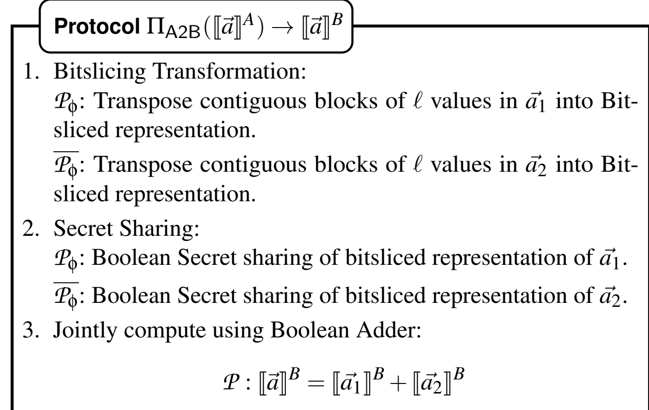


Figure 3: Vectorized Arithmetic to Binary Conversion with Bitslicing transformation

Switching between Arithmetic Vectorization and GPU Acceleration To utilize GPU acceleration for convolutions, PIGEON requires parties to convert their shares from arithmetic vectorization to a layout supported by the CUDA kernel provided by CUTLASS. Note that the raw transfer speed from the CPU to the GPU is not a concern since modern memory bus standards such as PCI 5.0 and HBM2 can transfer data at speeds that exceed any realistic network bandwidth in MPC settings by multiple order of magnitudes. Moreover, the CUDA kernel already takes care of the im2col transformation required for convolutions that expand the input matrices. Therefore, only the raw input, kernel, and output matrices need to be transferred between the CPU and the GPU.

However, we face a different challenge of converting layouts to a format that is supported by the CUDA kernel. Internally PIGEON uses PyTorch's *NCHW* layout where N is the batch size, C is the number of channels, H is the height, and W is the width of the input matrix. The CUDA kernel provided by CUTLASS instead requires an NHWC layout. Thus, to convert between *VCHW* and *VHWC*, we provide optimized CUDA kernels that closely follow NVIDIA's reference implementations for matrix transpositions.

Note that we also intend to interleave communication and computation and do not want to wait for the entire batch of *NCHWV* matrices to be processed before transferring them to the CPU and initiating communication. For this reason, PIGEON splits up the input matrix into N batches of *CHWV* matrices. As soon as the first batch has been computed, PIGEON converts it back to a *CHWV* layout and transfers it back to the CPU to initiate communication. Fully Connected layers already support different layouts and are usually small enough to be processed on the CPU. Therefore, we do not require any additional conversion for these layers.

Table 1 shows that for batch sizes of 16, all transformations required by PIGEON achieve more than 250 Gbit/s of throughput which is significantly higher than the 25 Gbit/s network bandwidth that we assume in ideal network conditions.

Table 1: Throughput of Transformations in Gbit/s

Batch Size	Vectorization	Bitslicing	GPU[a]
1 [b]	87	25	94
16	617	264	680

[a] Layout change from $VCHW$ to $VHWC$
[b] Utilizes only a single CPU core

4.6 Bringing It All Together

Finally, we show how PIGEON combines all acceleration techniques to evaluate a neural network. Arithmetic vectorization is used to accelerate secret sharing and revealing and thus the starting and end point of each inference. When evaluating a non-linear layer, r bits of ℓ vectorized inputs are packed together for Bitslicing where r is the largest register size available on a system. Each individual boolean instruction then operates on r bits in parallel. The result is then converted back to arithemtic vectorization. Similarly, to perform a convolution or matrix multiplication weights and inputs for a convolution are moved to the GPU in batches which allows interleaving communication and computation. The results are then transferred back to the CPU and loaded into vectorized variables. Table 2 shows the utilized accelerations and transformation techniques for each common neural network layer.

Table 2: Utilizing the ABG programming model in PIGEON

Layers	Acceleration	Transform
BatchNorm	Arith. Vec	-
AvgPool, Adaptive AvgPool	Arith. Vec	-
ReLU	Bitslicing	A ↔ B
MaxPool	Bitslicing	A ↔ B
Argmax	Bitslicing	A ↔ B
Convolution[a]	GPU	A ↔ G
Fully Connected Layer [a]	GPU	A ↔ G

[a] Can also be accelerated on the CPU using Arithmetic Vectorization.

5 PPML Network Optimizations

In the previous section, we showed how PIGEON utilies the ABG programming mode to accelerate each individual layers mainly from a computation perspective. In this section, we describe how PIGEON optimizes the network utilization during inference.

Interleaving Communication and Computation Interleaving communication and computation for MPC workloads is crucial to ensure that the communication channels are not idle while the parties are performing local computations. PIGEON interleaves computation and communication by starting to communicate as soon as the first elements of a layer are processed as opposed to waiting for the entire layer to be processed. As described in §4, PIGEON's CPU-based implementation communicates on a dot-product level while PIGEON's GPU-based implementation splits up a large matrix multiplication into smaller batches to interleave computation and communication. To optimize the packet size when interleaving communication and computation, PIGEON sends and receives elements in chunks using a tunable buffer size. In practice, we experienced good performance with a buffer size of around 2MB for a wide range of applications.

Another way PIGEON interleaves communication and computation is by sending and receiving data continuously in parallel to local computation. This way, all incoming and outgoing communication in PIGEON is non-blocking. Only when the main thread requires a chunk of elements from the receiving thread, it needs to wait for a condition variable to be signaled.

A third way PIGEON interleaves communication and computation is by mapping inference inputs to independent CPU processes such that processes can evaluate layers asynchronously and there is only a single spawning and joining of independent processes at the beginning and end of the inference. In combination with load balancing and pipelining described in the following paragraphs, this enables PIGEON to evaluate a computation-intensive layer in one process while another independent process saturates the network bandwidth by evaluating a communication-intensive layer.

Load Balancing and Pipelining In MPC frameworks, each node typically performs all communication and computation of a single party. However, since batches in neural networks are independent, nodes can perform multiple MPC computations in parallel where each computation uses a different party assignment. This way, a PPML framework can utilize all communication channels and hardware resources evenly across all nodes. For instance, if an MPC protocol does not utilize the communication channel between P_0 and P_1 but utilizes the communication channel between P_0 and P_2, swapping the party assignment of P_1 and P_2 for an independent batch also utilizes the otherwise idle communication channel. By default, PIGEON runs all unique player permutations in parallel which guarantees that all communication channels are utilized evenly.

Additionally, different parties in MPC often have different computation complexities. For instance, Π_{MatMul} might

require P_1 to perform two local matrix multiplication on its shares while P_2 only needs to perform one. This property inherently leads to pipelining. A node that performs batch one as P_1 may not be able to fully saturate the whole network bandwidth during the expensive convolutional layers. If the same node performs a second batch as P_2, it can utilize this idle network bandwidth for communication-intensive activations.

Interleaving MPC Phases Many MPC protocols provide a preprocessing phase and an online phase while maliciously secure protocols may additionally provide a postprocessing phase where parties compute and exchange hashes to verify the correctness of their computation. While separating these phases is beneficial in settings where preprocessing and postprocessing costs are irrelevant, it is beneficial to interleave all phases when optimizing for the total runtime of a protocol. PIGEON provides an option to separate or interleave these phases. When performing all phases sequentially, the parties essentially perform multiple forward passes that operate on the same blocks of memory. By merging all phases into a single online phase, the parties perform only a single forward pass. In this processing model, parties have an additional dependency on data from the preprocessing phase. However, in practice, the preprocessing in honest-majority protocols is much faster than the online phase and the additional dependency is unlikely to lead to a bottleneck. When interleaving all phases, PIGEON achieves 40% faster end-to-end 4PC inference runtime.

6 Software Architecture

PIGEON's software architecture can be categorized into three different modular software components: The Core components, MPC components, and Neural-network components. In this section, we describe the functionalities of each component. Figure 4 shows PIGEON's software architecture and its key features.

Core Components PIGEON's core components contain over 20,000 lines of highly optimized C++ code to offer the hardware acceleration techniques required to support the ABG programming model for different architectures. As a result, each function invoked by the higher-level components automatically utilizes the introduced acceleration techniques. On top of these, PIGEON also accelerates cryptographic instructions. Most MPC protocols require sampling shared pseudo-random numbers and in case of malicious adversaries, secure cryptographic hash functions. X86 and ARM processors have introduced instructions that accelerate the underlying cryptographic primitives such as AES and SHA. PIGEON utilizes VAES hardware instruction to generate up to 512 bits of random bits at once using AVX-512 registers. To compute cryptographic hash functions, PIGEON uses the X86 SHA hardware instruction. For ARM-based processors, PIGEON supports similiar hardware instructions and also provides architecture-independent implementations for systems that do not support any hardware acceleration for cryptographic primitives. PIGEON uses the OpenSSL library to TLS-encrypt communication between nodes.

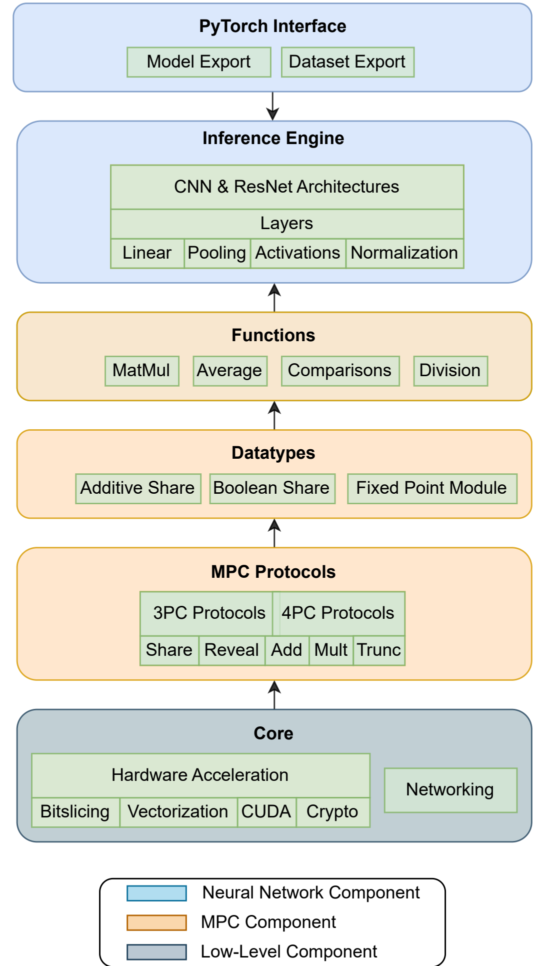


Figure 4: PIGEON's software architecture.

MPC Software Components PIGOEN's MPC software components allow users to write generic MPC functions based on high-level MPC primitives and to add new MPC protocols with a few lines of code. Functions define operations that can be implemented on top of any MPC protocol as long as the protocol supports required basic primitives. For instance, computing an average requires black-box access to the Π_{Add}, Π_{CMult}, and Π_{Trunc} primitives. Functions are based on MPC datatypes such as Additive Shares that provide a common templated interface for MPC primitives without specifying the underlying protocol. PIGEON provides implementations

of the *Trio* semi-honest 3PC and the *Quad* malicious 4PC protocols out of the box [16]. Protocols in PIGOEN require implementing the basic primitives introduced in §3. These primitives can be implemented with just a few lines of code as common local operations such as sampling shared random numbers and sending or receiving messages are implemented by PIGEON's core components.

Neural Network Software Components CryptGPU [48] innovated the user experience of PPML by providing a PyTorch wrapper that allows users to interact with PPML similarly to plaintext PyTorch. However, this design choice introduced performance overheads and workarounds to ensure compatibility with PyTorch such as representing performing matrix multiplications on fixed point numbers by using intermediate floating point representations. Piranha [51] improved the performance overhead of CryptGPU by relying solely on a C++ library with custom CUDA kernels but gave up on compatibility with common ML frameworks. PIGEON combines the advantages of both approaches by providing a PyTorch interface that allows users to export existing models and datasets to PIGEON's C++ inference engine after local processing in PyTorch.

PIGEON's C++ inference engine implements common neural network layers and their data flow during inference. The inference engine relies on arithmetic operations in a blackbox fashion by using templates. This abstraction layer enables developers to implement new neural network layers and architectures without having to understand the underlying MPC protocols that instantiate the templates with MPC primitives.

Out of the box, PIGEON supports various neural network architectures such as VGG16 [46] and ResNet50 [17] and common linear layers, pooling layers, activation functions, and batch normalization.

7 Evaluation

Given that we identified Piranha [51] as the state-of-the-art PPML framework for end-to-end private inference we compare our framework mainly to Piranha. Our test setup is based on 3-resp. 4 AWS nodes (for 3PC resp. 4PC evaluations) equipped with 24 GB Nvidia L4 GPUs on a 25 Gbit/s network and 0.3ms round-trip latency between nodes. As MPC requires distributed nodes in real-world settings, we consider this setup sufficient to stress-test whether our framework can saturate all possible real-world network bandwidths. All measurements include the total time spent on preprocessing, online phase, and verification.

7.1 Overcoming PPML Limitations

With its ABG programming model, PIGEON overcomes multiple limitations of state-of-the-art PPML frameworks such as Piranha.

GPU Memory Requirements Piranha improved over CryptGPU's memory requirements by utilizing native integer GPU kernels and using efficient in-place memory implementations However, depending on the MPC setting and ring size, Piranha still requires 2.4-7GB of GPU memory for a single-image inference of VGG-16 on Imagenet. Thus, to enable batched ImageNet inference Piranha requires High-End GPUs. In §4 we showed that by using the ABG programming model, PIGEON can accelerate most layers on the CPU while achieving nearly 100% network utilization. This insight allows PIGEON to outsource only convolutions to the GPU, thus achieving peak GPU memory utilization of only 205 MB for a single inference of VGG-16 on Imagenet.

Supporting Large Batch Sizes Private inference requires large batch sizes to overcome latency bottlenecks that are present in most MPC settings. However, despite its memory optimizations, Piranha still requires many GBs of GPU memory to evaluate state-of-the-art models. Table 3 shows that even on our 24 GB Nvidia L4 GPUs per node, we are limited to batch sizes between 2 and 8 before the framework crashes when evaluating VGG-16 on Imagenet. By minimizing the GPU memory footprint, PIGEON supports large batch sizes such as 192 for VGG-16 on Imagenet while using only 5.5 GB of peak GPU memory. On CIFAR-10, PIGEON is even able to perform inferences on the entire test dataset with a batch size of 10,000 images without running out of memory.

Table 3: Highest supported[a] inference batch size of Piranha on 24GB NVIDIA L4 GPUs (VGG16, ImageNet)

Setting	32 bit		64 bit	
	3PC	4PC	3PC	4PC
Maximum Batch Size	8	4	2	2
Peak GPU Memory (GB)	11.44	9.36	7.23	10.82

[a] Next power of two throws runtime error.

Scaling While Piranha effectively accelerates inferences with a batch size of one, its throughput even decreases when increasing the batch size: Table 4 shows that when gradually increasing the batch size from 1 to 8 in the 3PC setting with a 32bit ring size, Pirnaha's throughput drops consistently until reaching a 25 % performance penalty for higher batch sizes.

Table 4: Piranha inference throughput in images per second on VGG16, ImageNet, 3PC, 32-bit

Batch Size	1	2	4	8
Inferences per second	0.29	0.25	0.22	0.21

Table 5: Layer-wise Benchmark: Imagenet, Batchsize 192, 3PC, 32bit

Model	Layer	GB[c]	Runtime (s)		Gbps	
			PIGEON CPU	PIGEON GPU	PIGEON CPU	PIGEON GPU
VGG-16	LINEAR	0.01	0.62 ± 0.02	0.59 ± 0.02	0.09 ± 0.00	0.10 ± 0.00
	FLATTEN	0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
	AVGPOOL2D	0.39	0.15 ± 0.01	0.15 ± 0.02	**21.33** ± 1.59	20.36 ± 3.23
	ACTIVATION	45.30	12.29 ± 0.56	16.16 ± 0.43	**29.50** ± 1.34[b]	22.42 ± 0.60
	CONV2D	10.40	29.80 ± 0.48	9.14 ± 0.57	2.79 ± 0.04	**9.10** ± 0.57
	Total	56.11	40.32 ± 0.83	25.10 ± 0.33	11.13 ± 0.23	**17.88** ± 0.24
ResNet152	LINEAR	0.00[a]	0.04 ± 0.00	0.13 ± 0.03	0.17 ± 0.02	0.05 ± 0.01
	ADAPTIVEAVGPOOL2D	0.00[a]	0.00 ± 0.00	0.00 ± 0.00	6.08 ± 0.80	4.91 ± 0.77
	FLATTEN	0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
	AVGPOOL2D	0.05	0.04 ± 0.00	0.04 ± 0.00	10.49 ± 0.45	9.52 ± 0.96
	ACTIVATION	70.39	17.87 ± 0.38	24.38 ± 2.15	**31.51** ± 0.68[b]	23.10 ± 2.04
	BATCHNORM2D	34.64	8.67 ± 0.65	12.22 ± 0.70	**31.98** ± 2.38[b]	22.69 ± 1.30
	CONV2D	17.32	23.79 ± 0.40	13.29 ± 0.52	5.83 ± 0.10	**10.43** ± 0.40
	Total	122.40	48.41 ± 0.66	47.39 ± 0.73	20.23 ± 0.28	**20.66 ± 0.32**

[a] Communication is greater 0 but less than 0.01GB
[b] Layer benefits from previous layer's idle communication channels due to inter-batch pipelelining
[c] Communication per party in GB

Table 5 shows that by using its inter-batch concurrency model and ABG programming model, PIGEON achieves 17.88 Gbit/s throughput in the 3PC setting for a batch size of 192 when evaluating VGG-16 on Imagenet. This constitutes an improvement of over 20 times compared to PIGEON's single inference throughput.

ReLU Performance Piranha's throughput is severely bottlenecked by ReLU. ReLU layers require multiple communication rounds between the parties thus necessitating frequent data movements from between the CPU and the GPU to interact with the system's network socket. Between these communication rounds, the GPU is only used for small bursts of computation. While Piranha improved over CryptGPU to accelerate ReLU on the GPU, we find that it is still far from saturating our network, even with high batch sizes and large input sizes. For instance, in the 4PC, 64-bit setting, Piranha saturates less than 2% of our available network bandwidth. We conclude that the GPU is not the right hardware to fully accelerate ReLUs for MPC. PIGEON instead utilizes Bitlsicing and multiple CPU cores to saturate more than 90% of our network. In §B, Figure 5 shows that this leads to around one order of magnitude improvement for single inferences and two orders of magnitude improvement for batched inferences. Figure 5d shows our largest improvement: Our batch-wise evaluation of 192 ReLU layers in the 4PC, 64-bit setting closely matches Piranhas runtime when evaluating a single ReLU layer.

7.2 Benchmark

We benchmark PIGEON and Piranha's inference performance in the 3PC and 4PC settings with different ring sizes. According to common practice, we replace MaxPooling layers with AveragePooling layers which are more MPC-friendly. Throughput measurements report Piranha's best-performing batch size for each model which almost exclusively is a batch size of one due to Piranhas earlier mentioned scaling limitations. We limit PIGEON's batch size to 192. While PIGEON supports higher batch sizes for most models and datasets on our hardware, it might be unrealistic to assume that a real-world setting would require processing more than 192 inputs in parallel.

Throughput Table 6 shows the throughput the two frameworks achieve when evaluating different models and datasets using a ring size of 32 bits. In §B, Table 7 contains similar results for ring sizes of 64 bits. The results show that PIGEON consistently outperforms Piranha by one to two orders of magnitude. These performance improvements can be mainly attributed to PIGEON's efficient ReLU implementation which showed similar performance improvements in Figure 5. When comparing the PPML throughput to plaintext PyTorch inference on ImageNet, PIGEON reduces the overhead from more than three orders of magnitude to two.

Network Saturation Given the large performance improvement of PIGEON over Piranha, we investigate how close PIGEON's throughput comes to the network limit of 25 Gbit/s.

Table 6: Throughput (Images per second) 32bit

Setting	Framework	CIFAR-10			ImageNet	
		AlexNet	ResNet50	VGG16	ResNet50	VGG16
3PC	Piranha	24.57 ± 0.06	3.10 ± 0.01	10.86 ± 0.01	1.03 ± 0.00	0.21 ± 0.00
	PIGEON CPU	**1409.30** ± 90.70	**247.72** ± 10.89	**208.70** ± 3.51	**8.37** ± 0.15	4.76 ± 0.11
	PIGEON GPU	48.81 ± 3.53	36.25 ± 0.80	42.37 ± 1.79	7.88 ± 0.12	**7.65** ± 0.12
4PC	Piranha	9.19 ± 0.01	1.02 ± 0.00	4.19 ± 0.00	-[a]	0.08 ± 0.00
	PIGEON CPU	**1034.11** ± 55.13	**171.03** ± 5.42	**122.78** ± 4.74	5.70 ± 0.14	3.24 ± 0.16
	PIGEON GPU	45.94 ± 2.53	31.68 ± 0.91	37.38 ± 1.26	**6.05** ± 0.18	**6.12** ± 0.09

[a] Rutime error

Table 5 contains the layer-wise and total runtimes with resulting network saturation that PIGEON achieves. The total runtime measures the entire forward pass of the CNN. The individual runtimes are measured independently by the runtime of the slowest process multiplied by the number of processes. We find that this way of reporting the throughput of individual layers closely matches the total runtime of the forward pass.

The results show that fully connected layers only contribute an insignificant percentage to the runtime of a CNN given their small size. For pooling and activations, PIGEON is able to saturate more than 20 Gbit/s of the available network bandwidth with the exception of the average pooling layers in ResNet-152 due to their small input size. For convolutional layers, PIGEON GPU achieves around 10 Gbit/s of throughput while PIGEON CPU achieves around 2-3 times lower throughput. However, this gap is closed by PIGEON's inter-batch pipelining and load balancing as described in §5. The table shows that the activation and batch nomalization layers which typically appear directly after a convolutional layer even exceed the network bandwidth of 25 Gbit/s for PIGEON CPU based on our measurements. This of course does not mean that at a certain point in time, the network is oversaturated but rather that the fraction of processes computing an activation can exploit that another fraction of processes is still stuck in the compute-intensive convolutional layers. As a result, even the slowest of n total processes is still able to utilize more than $25/n$ Gbit/s of the network bandwidth when evaluating an activation due to the asynchronous network utilization of processes.

In total, PIGEON GPU is able to saturate more than 70% of the network bandwidth which implies limited room for further improvements from an engineering perspective. As expected, achieving high throughput on VGG-16 is more challenging than on ResNet-152 due to the higher computation complexity of VGG-16's convolutional layers.

7.3 Making PPML more accessible

In order to support ImageNet inference with large batch sizes, existing GPU frameworks require High-End GPUs. While CPU-only frameworks such as FALCON [49] exist, they achieve less than one order of magnitude of throughput for batched inference of large neural networks than their GPU-only alternatives [51]. PIGEON addresses these limitations by providing GPU acceleration with low memory requirements and a high throughput CPU-only implementation.

PIGEON CPU As GPU hours are expensive, offering fast CPU-only implementations can make PPML more accessible. Additionally, CPU-based frameworks have the potential to support higher batch sizes as system memory is typically larger than GPU memory. Along with our GPU implementation, we provide a CPU-only implementation that achieves respectable convolution throughput by utilizing the techniques described in §4 such as cache tiling. Table 6 shows that even PIGEON CPU achieves one to two orders of magnitude higher throughput than Piranha. For models with small convolution sizes including all models evaluated on CIFAR-10, PIGEON CPU even outperforms PIGEON GPU.

PIGEON with limited compute resources While we showed that by using server-grade hardware PIGEON can outperform Piranha by one to two orders of magnitude, we also show that PIGEON is able to achieve state-of-the-art performance while utilizing only a few compute resources. In §B, Tables 8 and 9 contain the runtime of PIGEON compared to Piranha while restricting PIGEON to utilize only a single CPU core for local computation and a batch size of 1. The table shows that PIGEON CPU and PIGEON GPU provide comparable runtimes to Piranha under these restrictions. In this setting, PIGEON GPU only requires 205 MB of GPU memory. These results imply that PIGEON can enable fast private ImageNet inference even on low-end hardware.

Modular Design and PyTorch Interface Finally, PIGEON may also make PPML more accessible by enabling users to import existing models and datasets from PyTorch with a single command while PIGEON's modular design allows developers of different domains to extend either the neural network, MPC, or low-level software components of PIGEON.

References

[1] CUDA libraries documentation. https://docs.nvidia.com/cuda-libraries/index.html.

[2] Mauro Barni, Claudio Orlandi, and Alessandro Piva. A privacy-preserving protocol for neural-network-based computation. In *Proceedings of the 8th workshop on Multimedia and security*, pages 146–151, 2006.

[3] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*, 2019.

[4] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 337–367. Springer, 2015.

[5] Megha Byali, Harsh Chaudhari, Arpita Patra, and Ajith Suresh. Flash: Fast and robust framework for privacy-preserving machine learning. *Cryptology ePrint Archive*, 2019.

[6] Octavian Catrina and Amitabh Saxena. Secure Computation with Fixed-Point Numbers. In *FC*, 2010.

[7] Harsh Chaudhari, Ashish Choudhury, Arpita Patra, and Ajith Suresh. Astra: high throughput 3pc over rings with application to secure prediction. In *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop*, pages 81–92, 2019.

[8] Harsh Chaudhari, Rahul Rachuri, and Ajith Suresh. Trident: Efficient 4pc framework for privacy preserving machine learning. *arXiv preprint arXiv:1912.02631*, 2019.

[9] Anders PK Dalskov, Daniel Escudero, and Marcel Keller. Fantastic four: Honest-majority four-party secure computation with malicious security. In *USENIX Security Symposium*, pages 2183–2200, 2021.

[10] Ivan Damgård, Kasper Damgård, Kurt Nielsen, Peter Sebastian Nordholt, and Tomas Toft. Confidential benchmarking based on multiparty computation. In *International Conference on Financial Cryptography and Data Security*, pages 169–187. Springer, 2016.

[11] Daniel Escudero. An introduction to secret-sharing-based secure multiparty computation. *Cryptology ePrint Archive*, 2022.

[12] Andre Esteva, Brett Kuprel, Roberto A Novoa, Justin Ko, Susan M Swetter, Helen M Blau, and Sebastian Thrun. Dermatologist-level classification of skin cancer with deep neural networks. *nature*, 542(7639):115–118, 2017.

[13] Karthik Garimella, Zahra Ghodsi, Nandan Kumar Jha, Siddharth Garg, and Brandon Reagen. Characterizing and optimizing end-to-end systems for private inference. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 89–104, 2023.

[14] Sorin Grigorescu, Bogdan Trasnea, Tiberiu Cocias, and Gigel Macesanu. A survey of deep learning techniques for autonomous driving. *Journal of field robotics*, 37(3):362–386, 2020.

[15] Kanav Gupta, Neha Jawalkar, Ananta Mukherjee, Nishanth Chandran, Divya Gupta, Ashish Panwar, and Rahul Sharma. Sigma: Secure gpt inference with function secret sharing. *Cryptology ePrint Archive*, 2023.

[16] Christopher Harth-Kitzerow, Ajith Suresh, Yonqing Wang, Hossein Yalame, Georg Carle, and Murali Annavaram. High-throughput secure multiparty computation with an honest majority in various network settings, 2024.

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

[18] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal processing magazine*, 29(6):82–97, 2012.

[19] Neha Jawalkar, Kanav Gupta, Arkaprava Basu, Nishanth Chandran, Divya Gupta, and Rahul Sharma. Orca: Fss-based secure training and inference with gpus. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 63–63. IEEE Computer Society, 2023.

[20] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, et al. Highly accurate protein structure prediction with alphafold. *nature*, 596(7873):583–589, 2021.

[21] Donald Ervin Knuth. *The art of computer programming*, volume 3. Pearson Education, 1997.

[22] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. SWIFT: Super-fast and Robust Privacy-Preserving Machine Learning. In *USENIX Security Symposium*, pages 2651–2668, 2021.

[23] Nishat Koti, Arpita Patra, Rahul Rachuri, and Ajith Suresh. Tetrad: Actively secure 4pc for secure training and inference. *arXiv preprint arXiv:2106.02850*, 2021.

[24] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, Toronto, Ontario, 2009.

[25] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *NeurIPS*, pages 1106–1114, 2012.

[26] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[27] Yann LeCun, Bernhard E. Boser, John S. Denker, Donnie Henderson, Richard E. Howard, Wayne E. Hubbard, and Lawrence D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Comput.*, 1(4):541–551, 1989.

[28] Yann LeCun, Corinna Cortes, and Christopher J. Burges. The MNIST database of handwritten digits. `http://yann.lecun.com/exdb/mnist`, 2010. Accessed: July 19, 2024.

[29] Darius Mercadier. *Usuba, Optimizing Bitslicing Compiler*. PhD thesis, Sorbonne université, 2020.

[30] Darius Mercadier and Pierre-Évariste Dagand. Usuba: high-throughput and constant-time ciphers, by construction. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 157–173, 2019.

[31] Darius Mercadier, Pierre-Évariste Dagand, Lionel Lacassagne, and Gilles Muller. Usuba: optimizing & trustworthy bitslicing compiler. In *Proceedings of the 2018 4th Workshop on Programming Models for SIMD/Vector Processing*, pages 1–8, 2018.

[32] Payman Mohassel and Peter Rindal. Aby3: A mixed protocol framework for machine learning. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 35–52, 2018.

[33] Payman Mohassel and Yupeng Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *IEEE Symposium on Security and Privacy*, pages 19–38, 2017.

[34] Payman Mohassel and Yupeng Zhang. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *IEEE S&P*, 2017.

[35] Lucien KL Ng and Sherman SM Chow. Sok: Cryptographic neural-network computation. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 497–514. IEEE, 2023.

[36] NVIDIA Corporation. Cutlass: Cuda templates for linear algebra subroutines. `https://github.com/NVIDIA/cutlass`. Accessed: July 19, 2024.

[37] Ahmet Murat Ozbayoglu, Mehmet Ugur Gudelek, and Omer Berat Sezer. Deep learning for financial applications: A survey. *Applied soft computing*, 93:106384, 2020.

[38] Neungsoo Park, Bo Hong, and Viktor K Prasanna. Tiling, block data layout, and memory hierarchy performance. *IEEE Transactions on Parallel and Distributed Systems*, 14(7):640–654, 2003.

[39] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *NeurIPS*, pages 8024–8035, 2019.

[40] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. {ABY2. 0}: Improved {Mixed-Protocol} secure {Two-Party} computation. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2165–2182, 2021.

[41] Arpita Patra and Ajith Suresh. Blaze: blazing fast privacy-preserving machine learning. *arXiv preprint arXiv:2005.09042*, 2020.

[42] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115:211–252, 2015.

[43] Gilad Shainer, Ali Ayoub, Pak Lui, Tong Liu, Michael Kagan, Christian R Trott, Greg Scantlen, and Paul S Crozier. The development of mellanox/nvidia gpudirect over infiniband. a new model for gpu to gpu communications. *Computer Science-Research and Development*, 26:267–273, 2011.

[44] Dinggang Shen, Guorong Wu, and Heung-Il Suk. Deep learning in medical image analysis. *Annual review of biomedical engineering*, 19(1):221–248, 2017.

[45] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of

go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

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

[47] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27, 2014.

[48] Sijun Tan, Brian Knott, Yuan Tian, and David J Wu. Cryptgpu: Fast privacy-preserving machine learning on the gpu. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1021–1038. IEEE, 2021.

[49] Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. FALCON: honest-majority maliciously secure framework for private deep learning. *Proc. Priv. Enhancing Technol.*, 2021, 2021.

[50] Yongqin Wang, G. Edward Suh, Wenjie Xiong, Benjamin Lefaudeux, Brian Knott, Murali Annavaram, and Hsien-Hsin S. Lee. Characterization of mpc-based private inference for transformer-based models. In *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 187–197, 2022.

[51] Jean-Luc Watson, Sameer Wagh, and Raluca Ada Popa. Piranha: A {GPU} platform for secure computation. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 827–844, 2022.

[52] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *27th annual symposium on foundations of computer science (Sfcs 1986)*, pages 162–167. IEEE, 1986.

A Benchmark models and datasets in PPML

Most PPML frameworks evaluate their performance using convolutional neural networks (CNNs) on image datasets. The following datasets are commonly used for this purpose:

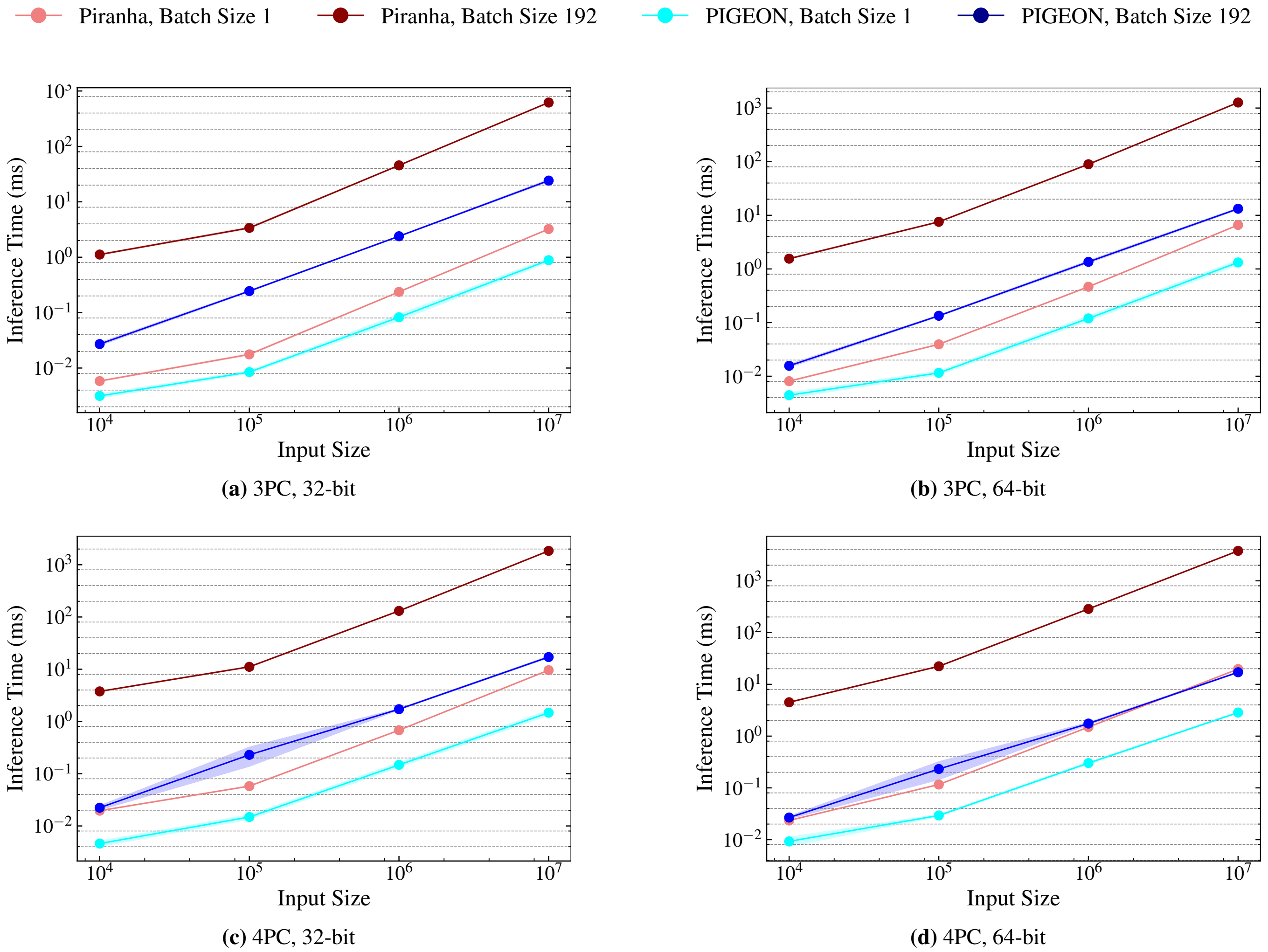
- MNIST [28]: A small-scale dataset consisting of 60,000 grayscale images and $28 \times 28 \times 1$ pixels per image.
- CIFAR-10 [24]: A dataset consisting of 60,000 color images and $32 \times 32 \times 3$ pixels per image.
- Imagenet [42]: A large-scale dataset with over 14 million color images and $224 \times 224 \times 3$ pixels per image.

The following neural network architectures are commonly used for benchmarking PPML frameworks:

- LeNet [27] and AlexNet [25]: Historically important, small CNNs with two and five convolutional layers respectively.
- VGG-16 [46]: A CNN with 16 convolutional layers and a large number of operations per layer.
- ResNet-18/50/101/152 [17]: Deep CNNs with 18 to 152 layers but fewer operations per layer than VGG-16.

Out of these different models and datasets, only VGG-16 and the different ResNet architectures can be considered state-of-the-art machine learning models based on their large number of trainable parameters, while only Imagenet can be considered a state-of-the-art dataset based on its number of pixels per image. In 2021, CryptGPU [48] was the first MPC-based PPML framework to achieve private inference of VGG-16 and ResNet architectures on the full-size Imagenet dataset.

B Additional Benchmark Results



(a) 3PC, 32-bit

(b) 3PC, 64-bit

(c) 4PC, 32-bit

(d) 4PC, 64-bit

Figure 5: ReLU Inference Time with different Bitlengths

Minor ticks represent a 2 times increase, major ticks represent a 10 times increase.

Table 7: Throughput (Images per second) 64bit

Setting	Framework	CIFAR-10			ImageNet	
		AlexNet	ResNet50	VGG16	ResNet18	VGG16
3PC	Piranha	17.99 ± 0.07	1.50 ± 0.00	5.08 ± 0.01	0.79 ± 0.01	0.11 ± 0.00
	PIGEON CPU	**941.51** ± 14.69	**81.97** ± 1.72	**76.85** ± 1.23	**3.51** ± 0.06	**1.77** ± 0.02
4PC	Piranha	4.19 ± 0.00	0.52 ± 0.00	1.86 ± 0.00	-[a]	0.04 ± 0.00
	PIGEON CPU	**681.68** ± 17.11	**80.81** ± 0.89	**75.92** ± 1.41	**2.75** ± 0.03	**1.40** ± 0.04

[a] Rutime error

Table 8: Single-Core Runtime (Seconds) 32bit

Setting	Framework	CIFAR-10				ImageNet	
		AlexNet	ResNet18	ResNet50	VGG16	ResNet18	VGG16
3PC	Piranha[b]	0.07 ± 0.00	**0.22** ± 0.00	**0.94** ± 0.00	**0.16** ± 0.00	**0.97** ± 0.00	3.50 ± 0.02
	PIGEON CPU	**0.05** ± 0.11	1.50 ± 0.20	2.86 ± 0.28	0.28 ± 0.00	1.50 ± 0.10	6.34 ± 0.04
	PIGEON GPU	0.48 ± 0.13	1.55 ± 0.03	3.29 ± 0.12	0.60 ± 0.01	1.28 ± 0.05	**1.58** ± 0.10
4PC	Piranha[b]	0.18 ± 0.00	**0.52** ± 0.01	**2.09** ± 0.01	**0.44** ± 0.00	-[a]	9.91 ± 0.01
	PIGEON CPU	**0.08** ± 0.11	1.22 ± 0.00	2.84 ± 0.02	0.51 ± 0.02	3.45 ± 0.09	16.24 ± 0.17
	PIGEON GPU	0.52 ± 0.13	1.60 ± 0.02	3.25 ± 0.00	0.66 ± 0.02	**2.02** ± 0.14	**2.97** ± 0.20

[a] Rutime error
[b] The single-core restriction only applies to PIGEON. We do not restrict Piranha's CPU or GPU usage.

Table 9: Single-Core Runtime (Seconds) 64bit

Setting	Framework	CIFAR-10				ImageNet	
		AlexNet	ResNet18	ResNet50	VGG16	ResNet18	VGG16
3PC	Piranha[b]	**0.11** ± 0.00	**0.26** ± 0.00	1.50 ± 0.00	**0.27** ± 0.00	**1.61** ± 0.00	8.37 ± 0.04
	PIGEON CPU	0.21 ± 0.13	0.66 ± 0.01	**1.37** ± 0.02	0.45 ± 0.00	3.19 ± 0.16	15.66 ± 0.04
	PIGEON GPU	0.67 ± 0.06	1.10 ± 0.01	1.88 ± 0.03	0.74 ± 0.00	1.72 ± 0.15	**2.61** ± 0.19
4PC	Piranha[b]	[a]	**0.64** ± 0.00	-[a]	**0.74** ± 0.00	-[a]	23.01 ± 0.02
	PIGEON CPU	0.16 ± 0.09	0.77 ± 0.00	**1.55** ± 0.00	0.96 ± 0.02	7.39 ± 0.10	40.22 ± 0.06
	PIGEON GPU	0.64 ± 0.13	1.20 ± 0.00	2.07 ± 0.01	0.85 ± 0.02	**2.62** ± 0.10	**4.54** ± 0.16

[a] Rutime error
[b] The single-core restriction only applies to PIGEON. We do not restrict Piranha's CPU or GPU usage.

C Bitslicing and Vectorization illustrated

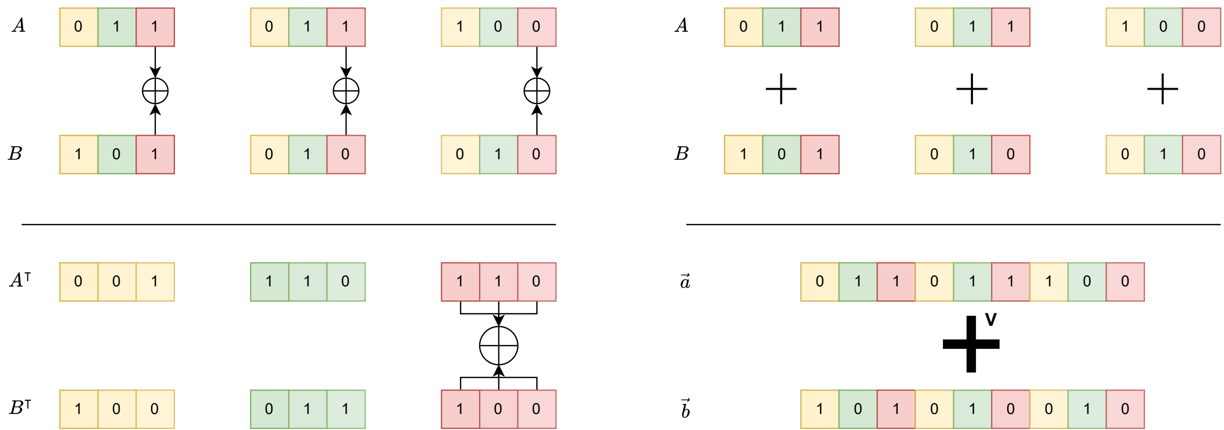


Figure 6: Bitslicing (left) and Vectorization (right) of independent integers replace multiple independent CPU instructions with a single one

D Matrix Multiplication Algorithms

Algorithm 1 Matrix Multiplication

Require: Matrices A of size $M \times K$ and B of size $K \times N$
Ensure: Matrix C of size $M \times N$, initalized with zeros

Naive MatMul

```
1:  for i ← 0 to M − 1 do
2:      for j ← 0 to N − 1 do
3:          temp ← 0
4:          for k ← 0 to K − 1 do
5:              temp += A[i][k] ×̂ B[k][j]
6:          end for
7:          C[i][j] ← temp
8:          Communicate C[i][j]
9:      end for
10: end for
```

Transposed MatMul

```
1:  b̂ ← transpose(B)
2:  for i ← 0 to M − 1 do
3:      for j ← 0 to N − 1 do
4:          temp ← 0
5:          for k ← 0 to K − 1 do
6:              temp += A[i][k] × b̂[j][k]
7:          end for
8:          C[i][j] ← temp
9:          Communicate C[i][j]
10:     end for
11: end for
```

Tiled MatMul

```
1:  b̂ ← transpose(B)
2:  for i ← 0; i < M; i += TILE_SIZE do
3:      for j ← 0; j < N; j += TILE_SIZE do
4:          for k ← 0; k < K; k += TILE_SIZE do
5:              for ii ← i; ii < min(i + TILE_SIZE, M); ii += 1 do
6:                  for jj ← j; jj < min(j + TILE_SIZE, N); jj += 1 do
7:                      temp ← 0
8:                      for kk ← k; kk < min(k + TILE_SIZE, K); kk += 1 do
9:                          temp += A[ii][kk] × b̂[jj][kk]
10:                     end for
11:                     C[ii][jj] ← C[ii][jj] + temp
12:                 end for
13:             end for
14:             for ii ← i; ii < min(i + TILE_SIZE, M); ii += 1 do
15:                 for jj ← j; jj < min(j + TILE_SIZE, N); jj += 1 do
16:                     Communicate C[ii][jj]
17:                 end for
18:             end for
19:         end for
20:     end for
21: end for
```

Note: Operator $\hat{\times}$ denotes fused multiplication on the entire secret share as described in the text.