PIGEON: A High Throughput Framework for Private Inference of Neural Networks using Secure Multiparty Computation

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 Soutern 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 (PPML) 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, GPU-based MPC frameworks are starting to achieve practical runtimes for private inference. However, we show that, unlike plaintext machine learning, using GPU acceleration for both linear (e.g., convolutions) and non-linear neural network layers (e.g., ReLU) is actually counterproductive in PPML.: While GPUs effectively accelerate linear layers compared to CPU-based MPC implementations, the MPC circuits required to evaluate non-linear layers introduce memory overhead and frequent data movement between the GPU and the CPU to handle network communication. This results in slow ReLU performance and high GPU memory requirements in state-of-the-art GPU-based PPML frameworks, hindering 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 employs a novel ABG programming model that switches between Arithmetic Vectorization and Bitslicing on the CPU for non-linear layers depending on the MPC-specific computation required while offloading linear layers to the GPU.

Compared to the state-of-the-art PPML framework Piranha, PIGEON improves ReLU throughput by two orders of magnitude, reduces peak GPU memory utilization by one order of magnitude, and scales better with large batch sizes. 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.

Keywords

PPML, MPC, Secure Inference

[1] Our framework is integrated into HPMPC: https://github.com/chart21/hpmpc/

Proceedings on Privacy Enhancing Technologies 2025(3), 88–105

https://doi.org/10.56553/popets-2025-0090

1 Introduction

Machine learning models have shown prediction capabilities that match human performance in various domains [4, 13, 21, 48]. Deep neural networks (DNNs) [27], a class of machine learning models with multiple layers, have proven particularly successful in image and speech recognition [19, 26], natural language processing [50], autonomous driving [15], medical diagnosis [47], and financial trading [41]. 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 creates a dilemma: Either companies must disclose their proprietary model parameters to clients, or clients must expose their private data to the company. In practice, clients are often at a disadvantage, as they must send their data to company-owned servers where the model is securely stored. 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) [38] aims to overcome this problem by enabling training and inference of machine learning models while keeping both model parameters and input data secret. One prominent approach to enable PPML is Secure Multiparty Computation (MPC). Although training models on large datasets like ImageNet using state-of-the-art (SOTA) MPC-based PPML frameworks would take multiple years of runtime, private inference is becoming increasingly practical: Piranha [54] and CryptGPU [51] reduced the MPC-based inference throughput of large convolutional neural networks from a few inferences per hour to a few inferences per minute by utilizing the GPU for all local operations required by an MPC protocol and leaving only network communication to the CPU. Despite these advancements, we demonstrate that a GPU-only approach constrains the performance and scalability of private inference by not considering several unique challenges in PPML.

1.1 Experimental Setup

While this work introduces implementation-related techniques applicable to any MPC protocol based on additive or Boolean secret-sharing, experiments conducted focus on honest-majority computation. All experiments presented in this work are based on semi-honest 3-PC and malicious 4-PC honest-majority protocols with the assumption that the model owner and data owner share their

inputs with a non-colluding set of servers conducting the MPC computation. Note that this outsourced computation model [11] is orders of magnitude faster than performing a native 2PC between model owner and data owner due to the high preprocessing costs of 2PC.

All evaluations are based on three to four nodes, each equipped with a 32-core AMD EPYC 7543 CPU and one 24GB NVIDIA L4 GPU in a LAN setting with 25 Gbit/s of network bandwidth, and 0.3ms round trip latency. As a baseline for our experiments, we choose the Piranha PPML framework [54]. Piranha was recently discovered as the PPML framework achieving the highest throughput out of over 50 recently proposed PPML frameworks [38]. Unless stated otherwise, all reported timings include the complete end-to-end forward pass, including preprocessing and the online phase. In the context of this work, we consider an implementation that achieves 25 Gbit/s of network throughput capable of fully saturating the network of any realistic MPC setting. Network saturation in the context of this work is defined as the total communication size in Gbit for the whole PPML inference divided by the available network bandwidth between parties in Gbit/s. Note that a PPML implementation is capped at 100% network saturation for a given MPC protocol and network setting. We refer the reader to §B for more details on network saturation and its relevance in evaluating the performance of MPC deployments.

1.2 Unique Challenges in PPML

Our findings that GPUs limit PPML performance may come as a surprise. Plaintext ML generally benefits from the high parallel processing power and memory bandwidth of GPUs. In plaintext ML, layers that require a large number of dot products, such as convolutional layers, are the primary contributors to inference and training runtime. Non-linear layers consume an insignificant amount of GPU resources in plaintext. Thus accelerating all layers in plaintext inference using GPUs is a logical choice. However, PPML presents distinct challenges, which we introduce in the following paragraphs.

Challenges in Non-linear Layers: GPU Memory 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 [35]. 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 [54], the SOTA PPML framework, was to evaluate the Boolean circuit in place and reduce GPU memory compared to CryptGPU [51].

Nevertheless, depending on the MPC setting and ring size, Piranha still requires 2.4-7GB of peak GPU memory for a single-image inference of VGG-16 on ImageNet which exceeds plaintext PyTorch memory requirements by more than one order of magnitude. Consequently, even on a 24 GB Nvidia L4 GPU per node, we are limited to batch sizes between 2 (64-bit ring size) and 8 (32-bit ring size) before Piranha crashes when evaluating VGG-16 on ImageNet. The results along with comparison to PIGEON and PyTorch [1] are shown in Table 1. Note that large batch sizes are paramount in PPML to amortize constant network latency between distributed parties over multiple inputs. Yet, due to the memory overhead of SOTA PPML frameworks, larger batch sizes need to be evaluated sequentially, thus suffering multiple times from network delay.

Table 1: Highest Supported[a] Inference Batch Size (B) and Resulting Peak GPU memory of PIGEON and Piranha on 3-4 Nodes with 24GB NVIDIA L4 GPU (32 Bit, VGG16, ImageNet)

Framework	Max Batchsize$_{Piranha}$		Max Batchsize$_{PIGEON}$	
	3PC: $B = 8$[a]	4PC: $B = 4$[a]	3PC: $B = 192$	4PC: $B = 192$
Piranha	11.44 GB	9.36 GB	-	-
PIGEON GPU	1.05 GB	0.83 GB	20.46 GB	22.48 GB
PyTorch[p]	0.57 GB	0.72 GB	9.98 GB	

[a] Next power of two throws runtime error
[p] Plaintext 32-bit floating-point computation on a single node

Challenges in Non-linear Layers: Network Saturation

While keeping all computations on the GPU minimizes CPU/GPU data movement in plaintext ML, the opposite is true in PPML. Non-linear layers such as ReLU and MaxPooling require multiple rounds of communication in MPC. In each communication round, GPU-accelerated frameworks need to move this data to the CPU to send it to 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 [46], these solutions require servers to be co-located in the same data center which is not a realistic deployment scenario for MPC. In addition to the communication overhead, the large Boolean circuits that need to be evaluated during non-linear layers consist of operations on individual bits which does not leverage the GPU optimally.

Given a fixed MPC protocol and circuit, 100% network saturation sets a strict lower bound on the runtime achievable by an optimal implementation. We use the metric as an indicator for runtime bottlenecks and identify that Piranha's 4PC ReLU implementation saturates less than 2% of a 25 Gbit/s network. An ideal implementation should aim to saturate nearly 100% of available network bandwidth given the low-latency test setup, thus leaving large room for implementation-related improvements. Note that frameworks such as Delphi [34] or FALCON [52] use the CPU instead to evaluate non-linear layers but as shown by the large-scale evaluation of [38] their performance significantly falls behind the GPU-based alternatives such as CryptGPU [51] or Piranha [54] due to their lack of acceleration techniques to achieve high throughput on the CPU.

Challenges in Linear Layers Convolutions and Fully Connected layers are computationally demanding both in plaintext ML and PPML. Evaluating a dot product, for example, requires only communicating a single message between parties in many MPC protocols [35]. Thus, when scaling to large dot products and batch sizes, these layers are constrained by computation rather than communication [17]. Hence, linear layers are suited for GPU acceleration even in PPML, provided that a framework can achieve a sufficiently large batch size where the performance gap between CPU and GPU computation is significant.

1.3 Our Contribution

To address these challenges, an efficient implementation must: 1) minimize CPU/GPU data movement when evaluating multi-round circuits, 2) support high batch sizes to amortize network latency, and 3) fully utilize available network bandwidth in distributed settings. Note that PPML frameworks can only benefit from the reduced communication overhead achieved by recent MPC protocols [2, 17, 43] in terms of real-world performance if they adapt implementation techniques that fully utilize the available network bandwidth.

While SOTA PPML frameworks [51, 54] accelerate all layers on the GPU we find that the unique challenges in PPML require a more targeted approach. Our approach accelerates non-linear layers on the CPU while using *Arithmetic Vectorization* and *Bitslicing* to overcome the throughput limitations of CPU-based PPML frameworks [34, 52]. We then provide novel PPML-specific enhancements for linear layer implementations to maximize GPU resources. In particular, we propose the *ABG* programming model that utilizes Arithmetic Vectorization, Bitslicing, and GPU acceleration depending on the PPML layer type. To switch between these techniques efficiently, we implement CUDA transformations and Bitslicing conversions. We demonstrate that by using the ABG programming model we can get the best of both worlds: High throughput and large batch sizes. Our PPML framework, PIGEON, implements the ABG programming model and provides the following contributions.

Contributions to Achieve High Network Saturation

(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 [54].
(3) PIGEON proposes several MPC-specific optimizations to reduce redundancy and interleave communication and computation when evaluating matrix multiplications. This enables PIGEON 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 computation domains (c.f §4.5). Along with MPC-specific pipelining techniques (c.f. §5), PIGEON saturates 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).

Contributions to Minimize Peak GPU Memory

(1) By offloading only convolutions to the GPU, PIGEON reduces peak GPU memory usage by an order of magnitude and supports ImageNet batch sizes 24–96 times larger than Piranha on the same hardware (c.f §7).
(2) These improvements enable us to support large ImageNet batch sizes (e.g. 192) for the first time in MPC-based PPML and consistently improve Piranha's throughput for ImageNet and CIFAR-10 inferences by one to two orders of magnitude.

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 proposed by [17] out of the box (c.f. §6) that currently provide the best-known communication complexity in their respective setting. Hence, achieving 100% of network saturation using these protocols would imply reaching the practical limit of PPML deployments until further theoretical contributions are achieved.

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, thereby omitting the need for expensive GPUs. When utilizing multiple cores, the CPU-only version even improves over Piranha's 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. In addition, we fully integrate PIGEON into the general-purpose HPMPC [17] framework, which benefits from our efficient conversions, GPU support, and PyTorch interface.

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 [12] and are typically deployed in the 2PC, 3PC, or 4PC settings. While 2PC protocols introduce high communication complexity in the preprocessing phase, the semi-honest 3PC and malicious 4PC settings tolerating up to one corruption are characterized by an inexpensive input-independent preprocessing phase followed by an input-dependent online phase. By leveraging the outsourced computation model [11], any number of input parties can secretly share their inputs with a set of non-colluding computation nodes, which perform the 3PC or 4PC computation and distribute the output shares to the intended recipients. This model is well-suited for private inference-as-a-service solutions, which, like their plaintext counterparts, handle independent client inference requests in parallel.

Recently, frameworks based on Function Secret Sharing (FSS) [5] have also shown impressive results, surpassing the performance of frameworks based on additive secret sharing [16, 20]. 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 inference requests, Garimella et al. [14] 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. In line with related work [51, 54], we assume that the model owner and dataset holder honestly submit their inputs to a setup of 2-4 non-colluding servers

that carry out the MPC computation and return the results. The model owner may also be replaced by a distributed set of parties who obtained a secretly shared model using private training. PIGEON implements both semi-honest and maliciously secure MPC protocols proposed by [17]. We provide an overview of models and datasets commonly used in PPML in §A.

While existing frameworks like Delphi [34] and HPMPC [17] have employed GPU offloading and Bitslicing, respectively, PIGEON's holistic ABG programming model allows for the first time to achieve high throughput for all PPML layers by combining acceleration strategies with efficient conversions. We implement these techniques on top of HPMPC which makes them also accessible for other applications apart from PPML.

MPC-based PPML Frameworks Early work on PPML goes back as far as 2006 [3] but the first training of a deep learning model on MNIST was only achieved in 2017 by SecureML [36]. SecureML set a standard for PPML frameworks by utilizing additive secret sharing for linear layers and Yao's garbled circuit protocol [55] 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 [35] 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 [6, 8–10, 23, 24, 44].

CryptGPU [51] first broke the trend of optimizing PPML mainly from a protocol perspective but instead proposed system 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 ML framework PyTorch [1] that allowed for easy integration of existing models and datasets but introduced some trade-offs by using floating point CUDA [39] kernels for fixed point computation. Piranha [54] improved on CryptGPU's performance by utilizing NVIDIA's CUTLASS [40] 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 [49]. MPC-Pipe [53] is the first work to introduce communication and computation overlapping for PPML. We adapt both their inner-layer and inter-batch pipelines for our ABG programming model.

In 2023 an SOK on PPML [38] was published that studied 53 PPML frameworks and identified Piranha as the fastest PPML framework to date. Interestingly, despite its relatively simple protocol design, Piranha outperforms PPML frameworks that employ more efficient underlying MPC protocols [8, 9, 24], primarily due to its emphasis on software and hardware optimizations. Also, according to the SOK, Piranha's 3PC implementation of Falcon [52] 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 [51, 54], 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) [7, 37] and mixed circuits [9, 35, 43] 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 [11], P_I and P_O are not required to participate in the computation. We denote a linear secret share (LSS) of a value x by $[\![x]\!]$ where x_i is the share held by $P_i \in P$. An LSS 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 these 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 [35] allow evaluating a matrix multiplication with $O(mnk)$ local operations but exchange of 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 $b \in \mathbb{Z}_2$ 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 $O(\ell \cdot log_2 \ell)$ bits in $O(log_2 \ell)$ communication rounds. Note that in terms of message size exchanging $O(\ell \cdot log_2 \ell)$ bits is equivalent to exchanging $O(log_2 \ell)$ ring elements in $\mathbb{Z}_{2^\ell}$.

Bit to Arithmetic Conversion (Π_{Bit2A}) Given the Boolean sharing of bit $[\![\mathrm{b}]\!] \in \mathbb{Z}_2$, 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 (FPA) require truncation to prevent overflows during computation and maintain precision [7]. 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.

ML Notations. We denote by $X^{\{i\}}$, $Y^{\{i\}}$, and $W^{\{i\}}$ the input, output, and weight matrices of the i-th Layer respectively. We distinguish between multiplication-based layers, comparison-based layers, and matrix-multiplication-based layers. The layer type indicates which operation is predominantly used to evaluate a layer and will prove helpful for accelerating its evaluation.

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 $\mathcal{P}$.

Matrix-Multiplication-based Layers

Layers such as Convolutional and Fully Connected layers require evaluating 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. Evaluating Convolutional Layers first requires an im2col conversion. The im2col (image-to-column) algorithm allows expressing a convolution as a matrix multiplication via input transformation [56]. Parties locally perform the im2col transformation on their shares to obtain $[\![\hat{X}^{\{i\}}]\!]$ and $[\![\hat{W}^{\{i\}}]\!]$, followed by Π_{MatMul} to obtain output matrix $Y^{\{i\}}$.

Comparison-based Layers

Comparison-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 $[\![b^{\{i,j\}}]\!] \in \mathbb{Z}_2$ of all individual shares $[\![x^{\{i,j\}}]\!] \in [\![X^{\{i\}}]\!]$. They, then use Π_{Bit2A} to obtain $[\![b^{\{i,j\}}]\!] \in \mathbb{Z}_{2^\ell}$. Finally, for each $y^{\{i,j\}} \in Y^{\{i\}}$, they set $[\![y^{\{i,j\}}]\!] = [\![(1 - b^{\{i,j\}})]\!]$. ReLU outputs $\max(0, x^{\{i,j\}})$ for each $x^{\{i,j\}} \in X^{\{i\}}$. 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 $[\![x^{\{i,a\}}]\!], [\![x^{\{i,b\}}]\!]$ can be computed as $DReLU([\![x^{\{i,a\}}]\!] - [\![x^{\{i,b\}}]\!]) \cdot ([\![x^{\{i,a\}}]\!] - [\![x^{\{i,b\}}]\!]) + [\![x^{\{i,b\}}]\!]$. 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.

Multiplication-based Layers

Multiplication-based layers require parties to evaluate element-wise multiplications and additions. These layers are characterized by moderate computational and 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 $Y^{\{i\}} = \frac{X^{\{i\}} - \mu}{\sqrt{\sigma^2 + \epsilon}} \cdot \gamma + \beta$. 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 $\hat{\sigma} = \gamma \cdot \frac{1}{\sqrt{\sigma^2 + \epsilon}}$ 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 introduces a novel ABG programming model to address system challenges in PPML. The ABG programming model utilizes Arithmetic Vectorization to accelerate multiplication-based layers, Bitslicing to accelerate comparison-based layers, and GPU offloading to accelerate matrix-multiplication-based layers. PIGEON additionally provides novel MPC-friendly implementations of matrix multiplications that reduce redundant computations and interleave computation with communications to further improve network utilization. Finally, PIGEON provides an automated approach to switch between the various approaches with low overhead.

4.1 Accelerating Multiplication-based Layers using Arithmetic Vectorization

Multiplication-based layers, such as Average Pooling and Batch Normalization, consist of large numbers of element-wise arithmetic operations on secret shares. While these layers benefit from GPU acceleration in plaintext inference, we observe that effectively utilizing the CPU in PPML allows servers to fully saturate any realistic network bandwidth for these layers. As a result, evaluating these layers on the GPU provides no additional benefit.

Modern CPUs support SIMD (Single Instruction, Multiple Data) instructions, enabling them to execute identical arithmetic operations on multiple elements simultaneously. For example, x86's AVX-512 instructions can operate on 16 32-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 addition, since element-wise operations can be evaluated in parallel they can be further accelerated on multiple cores. Figure 1 illustrates this vectorization process.

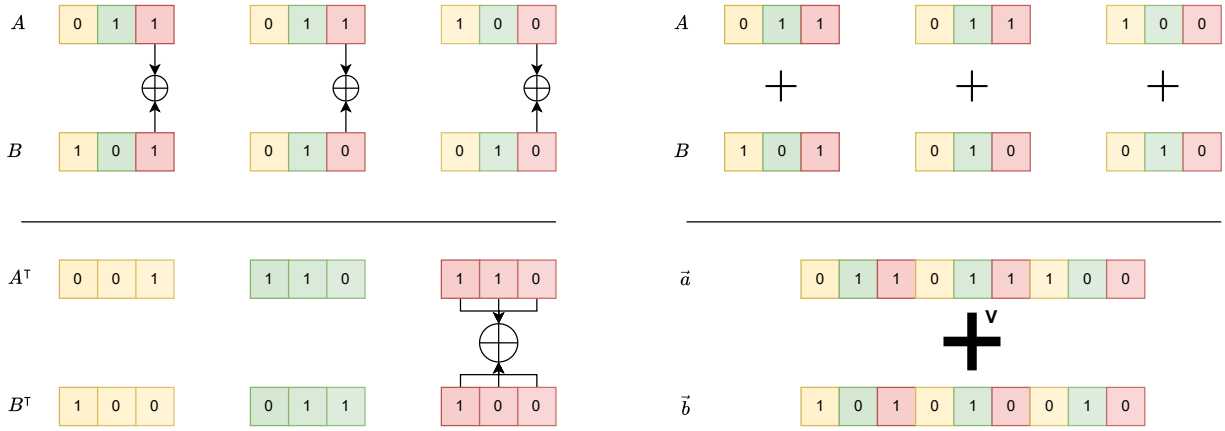


Figure 1: Bitslicing (left) and Vectorization (right) of Independent Integers Replace Multiple Independent CPU Instructions with a Single One

Since multiplication-based layers involve thousands to millions of independent element-wise operations, these layers are ideal candidates for Arithmetic Vectorization. Figure 2a shows how vectorization improves the throughput of Batch Normalization by up to five times on a single core, and Figure 2b 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 using just CPU-based parallelism.

4.2 Accelerating Comparison-based Layers using Bitslicing

Evaluating non-linear layers like ReLU requires share conversion. Comparison-based layers are constrained by the addition circuit evaluated in the Boolean domain during Π_{BitExt}, which involves hundreds of Boolean gates per input. These Boolean gates operate on 1-bit variables. Thus evaluating Boolean gates bit by bit is inefficient even with the smallest registers available on a CPU or GPU. To overcome this limitation we propose *Bitslicing*-based share conversion primitives.

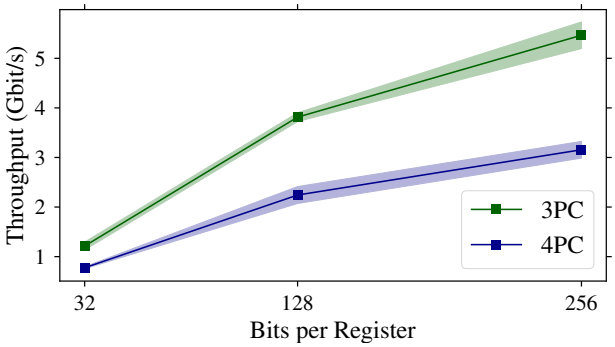
Bitslicing enables processing a batch of multiple Boolean inputs packed in a large register using only a single bit-wise instruction. The core concept of Bitslicing is that performing a bitwise logical operation on an m-bit register is equivalent to executing m parallel Boolean conjunctions, each processing a single bit. 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. Figure 1 illustrates how Bitslicing can replace multiple independent CPU instructions with a single one.

PIGEON supports Bitslicing using a wide range of register sizes from 64 bits to much wider register sizes provided by the SSE, AVX-2, and AVX-512 instruction sets. Figure 2c 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 2d shows how this throughput scales with the number of cores. Once again, by leveraging Bitslicing and parallelizing across multiple cores, PIGEON achieves nearly 100% network bandwidth utilization for comparison-based layers. Recall that once the network bandwidth is saturated no further implementation-related throughput improvements are theoretically feasible.

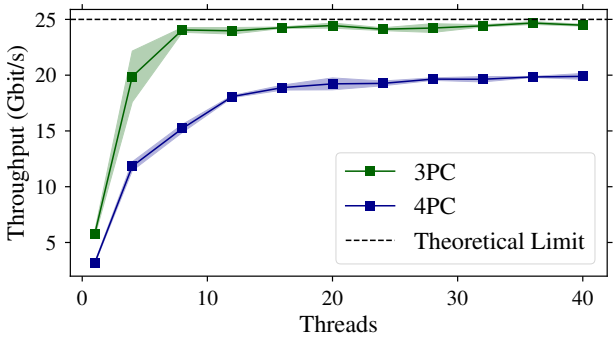
4.3 Accelerating Matrix-Multiplication based Layers using the CPU

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. To efficiently implement matrix multiplication on the CPU we combine our vectorization efforts with cache tiling approaches to avoid inefficient memory access patterns [42]. Nevertheless, it is not feasible in PPML to saturate high network bandwidths for matrix-multiplication-based layers using only known concepts from high-performance computing. For this reason, we propose MPC-friendly matrix multiplication approaches that optimize both the MPC-induced computation and communication overhead. We first outline PIGEON's optimizations for the CPU-only implementation before introducing GPU-accelerated approaches.

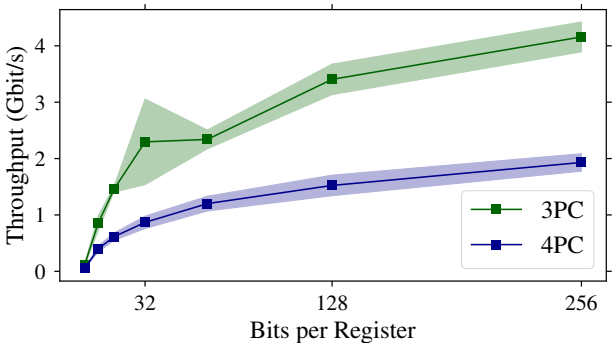
Reducing Redundancy Existing frameworks typically perform matrix multiplication on secret shares by substituting the local multiplication and addition operators in a secure multiplication protocol Π_{Mult} with their matrix-based counterparts. Consequently, a Π_{Mult} protocol that requires each party to perform three local multiplications and two local additions for each gate before communicating with other parties, would require each party to perform three local matrix multiplications and two local matrix additions for each secure matrix multiplication. Subsequently, the parties mask and exchange elements to compute the final result. Following this approach, each party needs to iterate over the same memory locations multiple times which introduces redundancy. As a result,



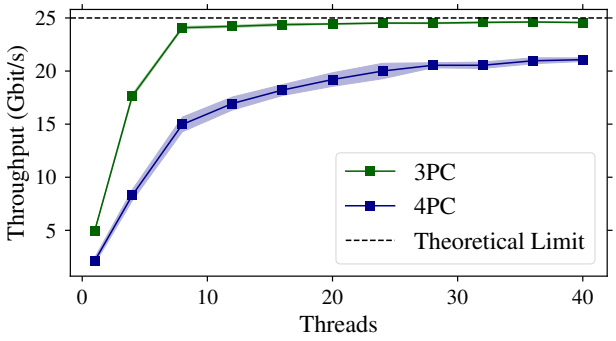
(a) Batchnorm2D Throughput using A: Single Core



(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 2: Accelerating PPML Layers using Arithmetic Vectorization and Bitslicing
Figures (b) and (d) show multi-core performance with 256 bits per register

state-of-the-art frameworks such as CryptGPU and Piranha require between 3 and 7 local matrix multiplications per party, depending on the underlying MPC protocol. To reduce this redundancy, PIGEON instead evaluates a secure matrix multiplication using only a single local matrix multiplication by replacing the local multiplication operator for each dot product with a fused operator that computes all individual local operations required by Π_{Mult} in a single pass before initiating communication.

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 $T = A_1B_2 + A_2B_1$ and $S = A_1B_1$, where T and S are temporary matrices required to initiate communication with other parties and compute a share of output matrix C. Existing approaches would require this party to compute three separate matrix multiplications resulting in $2mnk$ memory accesses for input matrices A_1, B_1 to compute the three products A_1B_2, A_2B_1, A_1B_1. Using PIGEON, the party instead computes $T[i][j]$ += $A_1[i][k] \cdot B_2[k][j] + A_2[i][k] \cdot B_1[k][j]$ and $S[i][j]$ += $A_1[i][k] \cdot B_1[k][j]$ whenever the plaintext matrix multiplication would normally calculate $C[i][j]$ += $A[i][k] \cdot B[k][j]$ as part of local dot product computation. This way, each value of a matrix share is immediately utilized for all required local computations, thus reducing the number of memory accesses for each input matrix to mnk irrespective of the MPC protocol's computational overhead.

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. Line 13 of our MPC-friendly matrix multiplication algorithm (c.f. algorithm 1) shows 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 3 visually shows that our optimization (Figure 3b) is different from the traditional approach (Figure 3a).

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. 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. Additionally, when evaluating multiple images in parallel, PIGEON utilizes independent CPU processes which allow a set of processes to evaluate communication-intensive layers

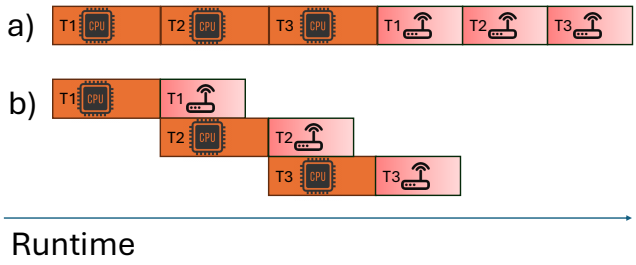


Figure 3: Sequential Evaluation (a) vs Interleaved Communication and Computation (b) in Tiled Matrix Multiplication

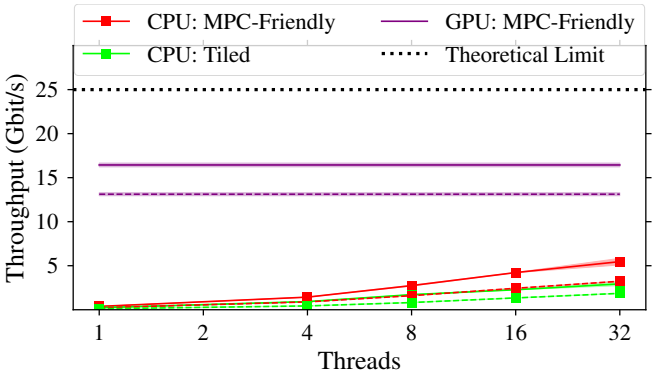


Figure 4: Conv2D Throughput of MPC-friendly Matrix Multiplication. Dotted lines indicate 4PC, solid lines 3PC

Algorithm 1 MPC-friendly MatMul

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

$B^T \leftarrow \text{transpose}(B)$
for $i \leftarrow 0; i < M; i$ += TILE_SIZE **do**
  for $j \leftarrow 0; j < N; j$ += TILE_SIZE **do**
    for $k \leftarrow 0; k < K; k$ += TILE_SIZE **do**
      for $ii \leftarrow i; ii < \min(i + \text{TILE\_SIZE}, M); ii$ += 1 **do**
        for $jj \leftarrow j; jj < \min(j + \text{TILE\_SIZE}, N); jj$ += 1 **do**
          $temp \leftarrow 0$
          for $kk \leftarrow k; kk < \min(k + \text{TILE\_SIZE}, K); kk$ += 1 **do**
            $temp$ += $A[ii][kk] \times B^T[jj][kk]$ ▷ Fused multiplication to eliminate redundancy
          $C[ii][jj] \leftarrow C[ii][jj] + temp$
      for $ii \leftarrow i; ii < \min(i + \text{TILE\_SIZE}, M); ii$ += 1 **do**
        for $jj \leftarrow j; jj < \min(j + \text{TILE\_SIZE}, N); jj$ += 1 **do**
          Communicate $C[ii][jj]$ ▷ Exchange individual dot products as soon as possible for interleaving

Note: Operator $\hat{\times}$ denotes fused multiplication on the entire secret share as described in the text. By default, we set TILE_SIZE = 64 in our implementation.

while another set of processes evaluates computation-intensive layers. This ensures that both communication channels and hardware resources are fully utilized throughout the inference.

Figure 4 compares the throughput of our MPC-friendly matrix multiplication when evaluating a 2D convolution with a highly optimized tiled matrix multiplication algorithm that does not utilize any MPC-specific tweaks. The results demonstrate that reducing redundancy and interleaving communication with computation lead to approx. 85% higher throughput than the baseline and enable even our CPU-based implementation to achieve a throughput of more than 5 Gbit/s with PIGEON. For the parameters of the convolutional layer, we use the largest Convolutional layer of VGG-16 on ImageNet with an input size of $224 \times 224 \times 64$.

4.4 Accelerating Matrix-Multiplication based Layers using the GPU

Even with tiled CPU implementations and our various MPC-specific tweaks, there is still a gap in the achieved throughput and the available network bandwidth for linear layers. Thus, linear layers in high-bandwidth settings are best accelerated by using GPUs that are optimized for matrix operations. Similar to Piranha [54], PIGEON uses NVIDIA's CUTLASS library [40] which provides highly optimized templated CUDA kernels for matrix multiplications and convolutions. PIGEON also supports multi-GPU setups where individual GPUs are assigned to mini-batches 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. To do so, PIGEON splits up a batch of k matrices into n mini-batches of size k/n and interleaves the computation of a mini-batch with the communication of the previous mini-batches. This overlap is similar to the one shown in Figure 3 except that we now interleave over mini-batches and not over tiles. Figure 4 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 a 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 CPU register of bit width $r = k \cdot \ell$ from a vectorized to a bitsliced 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 $k \cdot \ell$ integers. Naively, this transposition would be performed bit by bit in four cycles per bit [32]. However, the authors of Usuba [32] observed that this transposition can be optimized using a recursive matrix transposition approach proposed by Knuth [22]. Using this insight, Usuba reduced the number of cycles to transpose 512 AVX-512 variables to 0.09 cycles per bit [33]. The authors provide an open-source version of these transpositions for various hardware architectures and block sizes. In the appendix, §C describes the transition procedure in detail.

Figure 5 shows a 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 [35] 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 $[\![a]\!]^B$.

We show 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} (c.f. §3) follows the same procedure as Π_{A2B} but uses a carry adder in step 3 to only compute the sign bit of the addition. Share conversion from the Boolean to the arithmetic domain can also be implemented with local transpositions.

Protocol $\Pi_{\text{A2B}}([\![\vec{a}]\!]^A) \rightarrow [\![\vec{a}]\!]^B$

(1) Bitslicing Transformation:
   $\mathcal{P}_\phi$: Transpose contiguous blocks of ℓ values in $\vec{a}_1$ into bitsliced representation.
   $\overline{\mathcal{P}_\phi}$: Transpose contiguous blocks of ℓ values in $\vec{a}_2$ into bitsliced representation.
(2) Secret Sharing:
   $\mathcal{P}_\phi$: Boolean Secret sharing of bitsliced representation of $\vec{a}_1$.
   $\overline{\mathcal{P}_\phi}$: Boolean Secret sharing of bitsliced representation of $\vec{a}_2$.
(3) Jointly compute using Boolean Adder:

$$\mathcal{P} : [\![\vec{a}]\!]^B = [\![\vec{a}_1]\!]^B + [\![\vec{a}_2]\!]^B$$

Figure 5: Vectorized Arithmetic to Boolean 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 CUDA kernel already takes care of the im2col subroutine. Therefore, only the raw input, kernel, and output matrices need to be transferred between the CPU and the GPU.

While GPUs are optimized for large batch sizes we find that splitting up a large batch size into smaller batches for GPU offloading is preferable in MPC-based PPML as it allows interleaving communication and computation. These mini-batches also require a matrix layout change from PIGEON's $CHWN$ vectorized layout to an $NHWC$ supported by CUTLASS's convolution kernel. A $CHWN$ layout interprets a contiguous block of data as a 4D matrix with the dimensions of channel size (C), image height (H), image width (W), and batch size (N), while an $NHWC$ layout assumes a different order. To convert between these layouts, we provide optimized CUDA kernels that closely follow NVIDIA's reference implementations for matrix transpositions. In the appendix, §D describes the transition procedure in detail. Table 2 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 2: Throughput of ABG Transformations in Gbit/s

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

[a] Layout change from $CHWN$ to $NHWC$
[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 is 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 Arithmetic Vectorization during share conversion to the arithmetic domain. Similarly, to perform a convolution or matrix multiplication, weights and inputs for a convolution are moved to the GPU in mini-batches which allows interleaving communication and computation. The results are then transferred back to the CPU and loaded into vectorized variables. Table 3 shows the utilized accelerations and transformation techniques for each common neural network layer.

Table 3: Utilizing the ABG Programming Model in PIGEON

Layers	Acceleration	Transformation
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 Pipelining

In the previous section, we showed how PIGEON utilizes the ABG programming model to accelerate each individual layer. In this section, we describe how PIGEON optimizes the evaluation across layers by utilizing MPC-specific pipelining techniques.

Load Balancing 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. For large batch sizes, 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-, MPC-, and NN-components. In this section, we describe the functionalities of each component. Figure 6 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

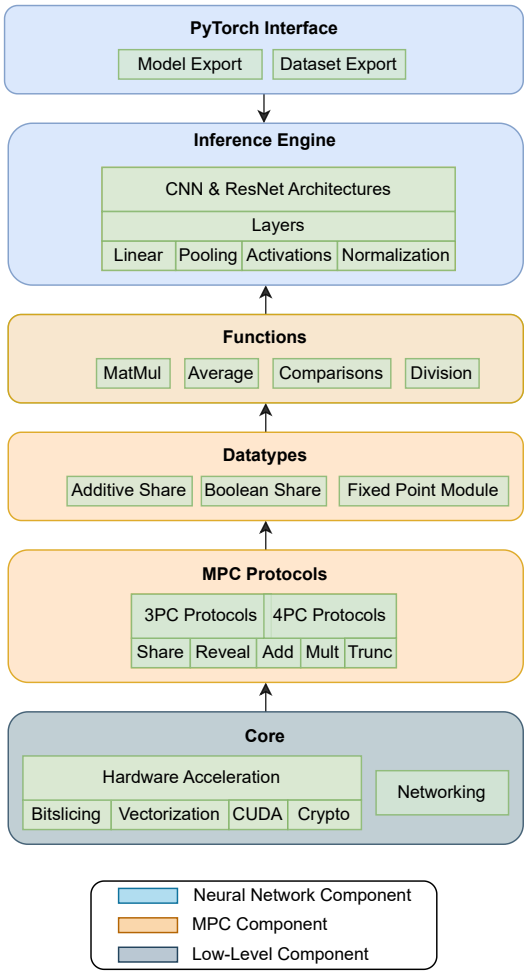


Figure 6: PIGEON's Software Architecture.

for different architectures. As a result, each function invoked by the higher-level components automatically utilizes the introduced acceleration techniques. PIGEON also accelerates cryptographic instructions such as AES and SHA on various platforms.

MPC Software Components PIGEON'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 [17]. Protocols in PIGEON require implementing the basic primitives introduced in §3. PIGEON provides an abstraction layer of its core component to inherently accelerate novel protocols and primitives.

Table 4: Throughput and Runtime for End-To-End Inference of 192 Images, 32 Bit

Setting	Framework	CIFAR-10			ImageNet	
		AlexNet	ResNet50	VGG16	ResNet50	VGG16
Throughput (Images per second)						
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
TTP[b]	PyTorch	8183.47 ± 174.09	7352.94 ± 173.18	7585.57 ± 310.47	499.20 ± 1.50	392.17 ± 3.20
Runtime (Seconds for 192 images)						
3PC	Piranha	7.82 ± 0.02	61.94 ± 0.20	17.68 ± 0.05	186.41 ± 0.50	914.29 ± 2.00
	PIGEON CPU	**0.14** ± 0.01	**0.78** ± 0.03	**0.92** ± 0.02	**22.94** ± 0.10	40.34 ± 0.12
	PIGEON GPU	3.93 ± 0.15	5.30 ± 0.10	4.53 ± 0.09	24.37 ± 0.18	**25.10** ± 0.20
4PC	Piranha	20.89 ± 0.05	188.24 ± 0.30	45.82 ± 0.10	-	2400.00 ± 5.00
	PIGEON CPU	**0.19** ± 0.02	**1.12** ± 0.04	**1.56** ± 0.03	33.68 ± 0.15	59.26 ± 0.18
	PIGEON GPU	4.18 ± 0.17	6.06 ± 0.12	5.14 ± 0.11	**31.74** ± 0.20	**31.37** ± 0.22
TTP[b]	PyTorch	0.024 ± 0.001	0.026 ± 0.001	0.025 ± 0.001	0.385 ± 0.001	0.490 ± 0.0040

[a] Runtime error
[b] Trusted Third Party (TTP): Plaintext 32-bit floating-point computation on a single node

Neural Network Software Components CryptGPU [51] 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 performing matrix multiplications on fixed point numbers by using intermediate floating point representations. Piranha [54] 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 black-box 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 [49] and ResNet50 [18] and common linear layers, pooling layers, activation functions, and batch normalization.

7 Evaluation

Given that we identified Piranha [54] as the state-of-the-art PPML framework for end-to-end private inference we compare our framework mainly to Piranha. In §F, we also provide benchmarks with CryptGPU [51].

7.1 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 [30]. Throughput measurements report Piranha's best-performing batch size for each model which almost exclusively is a batch size of one due to Piranha's scaling limitations. We limit PIGEON's batch size to 192. While PIGEON supports larger 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.

Table 4 shows the throughput the two frameworks achieve when evaluating different models and datasets using a ring size of 32 bits. In §F, 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 shows similar performance improvements for large batch sizes in §F, Figure 7. When comparing the throughput to plaintext PyTorch inference on ImageNet, PIGEON reduces the throughput overhead from more than three orders of magnitude achieved by state-of-the-art PPML frameworks to two.

7.2 Overcoming PPML Limitations

With its ABG programming model, PIGEON overcomes the current limitations of state-of-the-art PPML frameworks.

Network Saturation Given the large performance improvement of PIGEON over Piranha, we investigate whether PIGEON's end-to-end throughput reaches the network cap of 25 Gbit/s. Table 5 contains the layer-wise and total runtimes with resulting network

Table 5: Layer-wise Benchmark: ImageNet, Batchsize 192, 3PC, 32 Bit

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

saturation that PIGEON achieves. The total runtime measures the entire forward pass of the CNN.

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 except for 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 than PIGEON GPU. However, this gap is partly 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 a system's perspective.

GPU Memory Requirements 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 a peak GPU memory utilization of only 205 MB for a single inference of VGG-16 on ImageNet compared to 2.4-7GB required by Piranha. By minimizing the GPU memory footprint, PIGEON supports large batch sizes such as 192 for VGG-16 on ImageNet while using less than 24GB of peak GPU memory. This is more than one order of magnitude improvement compared to Piranha. Noticeably, PIGEON only requires twice the GPU memory for a given batch size than plaintext PyTorch. This narrow gap to plaintext computation might be explained our the mini-batch optimization to interleave communication and computation (c.f. §4.4). On CIFAR-10, PIGEON is able to perform inferences on all 10,000 images in a single batch without running out of memory.

7.3 Further Throughput Improvements

We can improve throughput further by evaluating large ReLU layers with Ripple Carry Adders, using key-bit optimizations [31], and scaling to large batch sizes.

Utilizing Different Adders In PPML, the most widely used adder circuit to evaluate share conversion required by non-linear layers is the Parallel Prefix Adder (PPA) [35] which introduces a communication complexity of $O(\ell\ log(\ell)$ per input and a round complexity of $log(\ell) + 1$. When scaling PPML to large models, datasets, and batch sizes the higher round complexity of the Ripple Carry (RCA) seizes to become a bottleneck. An RCA can be evaluated with a communication complexity of $O(\ell - 1)$ per input and a round complexity of $\ell - 1$. Table 6 shows that using an RCA over a PPA pays off when evaluating ImageNet models or scaling to large batch sizes for CIFAR-10. This is expected as the number of operations increases significantly when increasing batch sizes or using datasets with larger dimensions while the number of communication rounds stays constant.

Scaling to Large Batch Sizes While we set the maximum batch size in our benchmark to 192, some settings may benefit from even larger batch sizes. Table 6 shows that setting the batch size to 12,288 for VGG-16 on CIFAR-10 still provides higher throughput than a batch size of 192. One can also observe that as the batch size

Table 6: Throughput (Images per Second) with Different Tweaks, 3PC, 32 Bit

Framework	CIFAR-10		ImageNet	
	VGG-16	VGG-16, $B = 12,288$[a]	ResNet50	VGG16
PIGEON CPU (PPA)[b]	**208.70** ± 3.51	277.90 ± 3.08	8.37 ± 0.15	4.76 ± 0.11
PIGEON CPU (RCA)[c]	57.40 ± 0.93	308.99 ± 5.02	12.70 ± 0.33	5.08 ± 0.08
PIGEON CPU (RCA_8)[d]	134.37 ± 3.68	341.05 ± 1.84	11.30 ± 0.18	5.48 ± 0.07
PIGEON CPU (ON,RCA_8)[e]	144.92 ± 2.22	412.90 ± 6.72	13.78 ± 0.53	6.44 ± 0.08
PIGEON GPU (PPA)	42.37 ± 1.79	294.94 ± 0.48	7.88 ± 0.12	7.65 ± 0.12
PIGEON GPU (RCA)	31.40 ± 0.61	324.62 ± 2.35	9.54 ± 0.05	9.99 ± 0.29
PIGEON GPU (RCA_8)	45.72 ± 0.08	354.24 ± 0.98	10.69 ± 0.24	12.47 ± 0.29
PIGEON GPU (ON,RCA_8)	60.57 ± 0.24	**573.51** ± 15.10	**16.96** ± 0.75	**19.42** ± 1.31

[a] All model's batch sizes are set to 192, except this model where the batch size is set to 12,288
[b] Parallel Prefix adder is utilized to evaluate ReLU layers
[c] Ripple carry adder is utilized to evaluate ReLU layers
[d] Only 8 key bits are considered when evaluating ReLU layers
[e] Only the online phase is measured. All other measurements include end-to-end inference throughput

increases the RCA starts outperforming the PPA even for the small images in the CIFAR-10 dataset.

Key Bit Optimization The authors of [31] propose to evaluate ReLU heuristically by only considering certain most- and least-significant bits per ReLU layer, the so-called key bits. While their approach has an error probability that increases when considering fewer key bits, it reduces the round and communication complexity of ReLU layers significantly. Table 6 shows that considering 8 instead of 32 key-bits using an RCA improves end-to-end inference throughput by 10% for larger, bandwidth-constrained networks, and by more than 2 times for the smaller latency-constrained networks. Note that according to [31] even 6 key bits are sufficient to achieve nearly plaintext PyTorch accuracy for various models.

Online-only Computation While PIGEON efficiently interleaves the preprocessing and online phases, preprocessing costs may be less relevant for some applications. In this case, PIGEON can also be instructed to separate both phases. Table 6 shows the resulting throughput when only measuring PIGEON's online phase. The end-to-end throughput of separating both phases is significantly worse compared to PIGEON's interleaved end-to-end time throughput, however, when considering only the online phase one can achieve up to 58% additional improvement in throughput.

8 Conclusion

In this work, we propose PIGEON, a framework for private inference of neural networks. PIGEON provides protocol-agnostic implementation techniques that improve the hardware utilization of MPC servers by introducing MPC-friendly matrix multiplications, pipelining techniques, and the ABG programming model. The ABG programming model utilizes a targeted acceleration strategy for each PPML layer based on its underlying MPC primitives while ensuring high network saturation and low GPU memory requirements. This enables PIGEON to scale to large batch sizes and achieve new records in PPML inference throughput. PIGEON shows that system improvements to PPML framework design and addressing the unique challenges that separate PPML from plaintext ML can enable performance improvements of up to two orders of magnitude compared to state-of-the-art frameworks.

PIGEON can import model weights and datasets from PyTorch to easily evaluate existing models. Its software architecture enables developers to add new MPC protocols or neural network layers in a high-level fashion while inherently benefiting from the acceleration strategies introduced in this work. While PIGEON is a private inference framework, all acceleration techniques presented in this work are applicable to private training as well. Looking forward, an interesting direction for future research could be to investigate which additional challenges private training introduces that require innovations from a framework perspective.

Acknowledgements

We sincerely thank all the reviewers for their time and constructive comments. This material is based upon work supported by Defense Advanced Research Projects Agency (DARPA) under Contract Nos. HR001120C0088, NSF award number 2224319, REAL@USC-Meta center, and VMware gift. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. We used ChatGPT4 to revise the text to correct typos and grammatical errors.

References

[1] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, CK Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Michael Suo, Phil Tillet, Eikan Wang, Xiaodong Wang, William Wen, Shunting Zhang, Xu Zhao, Keren Zhou, Richard Zou, Ajit Mathews, Gregory Chanan, Peng Wu, and Soumith Chintala. 2024. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. ACM. https://doi.org/10.1145/3620665.3640366

[2] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. 2016. High-throughput semi-honest secure three-party computation with an honest majority. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 805–817.

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

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

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

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

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

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

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

[10] Anders PK Dalskov, Daniel Escudero, and Marcel Keller. 2021. Fantastic Four: Honest-Majority Four-Party Secure Computation With Malicious Security.. In *USENIX Security Symposium*. 2183–2200.

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

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

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

[14] Karthik Garimella, Zahra Ghodsi, Nandan Kumar Jha, Siddharth Garg, and Brandon Reagen. 2023. 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*. 89–104.

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

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

[17] Christopher Harth-Kitzerow, Ajith Suresh, Yongqin Wang, Hossein Yalame, Georg Carle, and Murali Annavaram. 2025. High-Throughput Secure Multiparty Computation with an Honest Majority in Various Network Settings. *Proceedings on Privacy Enhancing Technologies* (2025). https://eprint.iacr.org/2024/386

[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *CVPR*. 770–778.

[19] 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. 2012. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal processing magazine* 29, 6 (2012), 82–97.

[20] Neha Jawalkar, Kanav Gupta, Arkaprava Basu, Nishanth Chandran, Divya Gupta, and Rahul Sharma. 2023. Orca: FSS-based Secure Training and Inference with GPUs. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 63–63.

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

[22] Donald Ervin Knuth. 1997. *The art of computer programming*. Vol. 3. Pearson Education.

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

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

[25] Alex Krizhevsky. 2009. *Learning Multiple Layers of Features from Tiny Images*. Technical Report. University of Toronto, Toronto, Ontario. https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf

[26] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *NeurIPS*. 1106–1114.

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

[28] Yann LeCun, Bernhard E. Boser, John S. Denker, Donnie Henderson, Richard E. Howard, Wayne E. Hubbard, and Lawrence D. Jackel. 1989. Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Comput.* 1, 4 (1989), 541–551.

[29] Yann LeCun, Corinna Cortes, and Christopher J. Burges. 2010. The MNIST Database of Handwritten Digits. http://yann.lecun.com/exdb/mnist. Accessed: July 19, 2024.

[30] Yehuda Lindell, Benny Pinkas, Nigel P. Smart, and Avishay Yanai. 2015. Efficient Constant Round Multi-party Computation Combining BMR and SPDZ. In *CRYPTO*.

[31] Kiwan Maeng and G Edward Suh. 2024. Accelerating ReLU for MPC-Based Private Inference with a Communication-Efficient Sign Estimation. *Proceedings of Machine Learning and Systems* 6 (2024), 128–147.

[32] Darius Mercadier. 2020. *Usuba, Optimizing Bitslicing Compiler*. Ph. D. Dissertation. Sorbonne université.

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

[34] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. 2020. Delphi: A cryptographic inference system for neural networks. In *Proceedings of the 2020 Workshop on Privacy-Preserving Machine Learning in Practice*. 27–30.

[35] Payman Mohassel and Peter Rindal. 2018. ABY3: A mixed protocol framework for machine learning. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 35–52.

[36] Payman Mohassel and Yupeng Zhang. 2017. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *IEEE Symposium on Security and Privacy*. 19–38.

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

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

[39] NVIDIA Corporation. n.d.. CUDA Libraries Documentation. https://docs.nvidia.com/cuda-libraries/index.html.

[40] NVIDIA Corporation. n.d.. CUTLASS: CUDA Templates for Linear Algebra Subroutines. https://github.com/NVIDIA/cutlass. Accessed: July 19, 2024.

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

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

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

[44] Arpita Patra and Ajith Suresh. 2020. BLAZE: blazing fast privacy-preserving machine learning. *arXiv preprint arXiv:2005.09042* (2020).

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

[46] Gilad Shainer, Ali Ayoub, Pak Lui, Tong Liu, Michael Kagan, Christian R Trott, Greg Scantlen, and Paul S Crozier. 2011. The development of Mellanox/NVIDIA GPUDirect over InfiniBand. A new model for GPU to GPU communications. *Computer Science-Research and Development* 26 (2011), 267–273.

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

[48] 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. 2016. Mastering the game of Go with deep neural networks and tree search. *nature* 529, 7587 (2016), 484–489.

[49] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *ICLR*.

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

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

[52] Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. 2021. FALCON: Honest-Majority Maliciously Secure Framework for Private Deep Learning. *Proc. Priv. Enhancing Technol.* 2021 (2021).

[53] Yongqin Wang, Rachit Rajat, and Murali Annavaram. 2024. MPC-Pipe: An Efficient Pipeline Scheme for Semi-honest MPC Machine Learning. In *2024 ACM 29th International Conference on Architectural Support for Programming Languages and Operating Systems*.
[54] Jean-Luc Watson, Sameer Wagh, and Raluca Ada Popa. 2022. Piranha: A {GPU} platform for secure computation. In *31st USENIX Security Symposium (USENIX Security 22)*. 827–844.
[55] Andrew Chi-Chih Yao. 1986. How to generate and exchange secrets. In *27th annual symposium on foundations of computer science (Sfcs 1986)*. IEEE, 162–167.
[56] Yangjie Zhou, Mengtian Yang, Cong Guo, Jingwen Leng, Yun Liang, Quan Chen, Minyi Guo, and Yuhao Zhu. 2021. Characterizing and demystifying the implicit convolution algorithm on commercial matrix-multiplication accelerators. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 214–225.

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 [29]: A small-scale dataset consisting of 60,000 grayscale images and $28 \times 28 \times 1$ pixels per image.
- CIFAR-10 [25]: A dataset consisting of 60,000 color images and $32 \times 32 \times 3$ pixels per image.
- ImageNet [45]: A large-scale dataset with over one 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 [28] and AlexNet [26]: Historically important, small CNNs with two and five convolutional layers respectively.
- VGG-16 [49]: A CNN with 16 convolutional layers and a large number of operations per layer.
- ResNet-18/50/101/152 [18]: Deep CNNs with 18 to 152 convolutional 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 [51] was the first MPC-based PPML framework to achieve private inference of VGG-16 and ResNet architectures on ImageNet-sized images.

B Network Saturation and Batch Sizes as a Key Performance Metric of MPC Frameworks

While MPC protocols are typically evaluated by their communication and round complexity per gate, there are no standardized metrics to evaluate MPC deployments. As parties in MPC are distributed and need to communicate over the network to evaluate a circuit the maximum achievable throughput of an MPC implementation is capped by the total size in communication divided by the network bandwidth available on the network added to the number of interactions multiplied by the network latency.

For example, if each party can receive at most 8Gbit/s a second, evaluating a circuit with a communication size of 1GB per party and 1000 communication rounds with 1ms of latency between parties takes at least two seconds. A framework that achieves a runtime of 2 seconds in this setting can be considered ideal and achieves a throughput of 0.5 circuits per second.

Supporting large batch sizes enables the framework to amortize the network latency over multiple circuits. When evaluating 10 circuits an ideal framework can achieve a runtime of 11 seconds in this setting or a throughput of 0.91 circuits per second. To reduce PPML overhead, a framework should thus attempt to support large batch sizes to reduce the negative impact of network latency on inference throughput and be able to achieve 100% network saturation.

An additional consideration is the overhead of local computation. Recent work [17] discovered that several open-source MPC frameworks are often unable to achieve high network saturation due to their lack of effectively accelerating local computations. We therefore consider network saturation a key metric to compare MPC and PPML deployments assuming they use a state-of-the-art MPC protocol with low communication complexity.

C Arithemtic Vectorization to Bitslicing Conversion

PIGEON utilizes efficient conversions from Arithmetic Vectorization to Bitslicing. These conversions are based on USUBA [32] and shown in Algorithms 2 and 3. The conversion operates on blocks of r ℓ-bit inputs where ℓ represents the utilized bitlength and r is the register width supported on a target architecture. Note that the $A \leftrightarrow B$ conversion can also be implemented with hardware-specific instructions to further improve conversion throughput.

Algorithm 2 $A \leftrightarrow B$ Conversion

Require: Contiguous block of r ℓ-bit integers in arithmetic resp. bitsliced representation
Ensure: Contiguous block of r ℓ-bit integers in bitsliced resp. arithmetic representation
1: **for** $i \leftarrow 0$ to r **by** ℓ **do**
2: `Orthogonalize`$(data[i])$ ▷ Algorithm 3

Algorithm 3 Orthogonalize

Require: ℓ-bit integer in arithmetic resp. bitsliced representation
Ensure: ℓ-bit integer in bitsliced resp. arithmetic representation
1: **for** $i \leftarrow 0$ to $\log_2(\ell) - 1$ **do**
2: $v \leftarrow 2^i$
3: **for** $j \leftarrow 0$ to ℓ **by** $2v$ **do**
4: **for** $k \leftarrow 0$ to $v - 1$ **do**
5: $u \leftarrow data[j + k] \wedge mask_l[i]$
6: $v \leftarrow data[j + k] \wedge mask_r[i]$
7: $x \leftarrow data[j + v + k] \wedge mask_l[i]$
8: $y \leftarrow data[j + v + k] \wedge mask_r[i]$
9: $data[j + k] \leftarrow u \vee (x \gg v)$
10: $data[j + v + k] \leftarrow (v \ll v) \vee y$

Note: $mask_l, mask_r$ are arrays containing fixed values. The general procedure presented here is further optimized with hardware-specific instructions. For more details, we refer to our source code.

D Arithmetic Vectorization to GPU Conversion

To ensure compatibility between the CHWN data layout resulting from the use of Arithmetic Vectorization and the NHWC layout

expected by Cutlass's convolution implementation, we need to convert the input and output matrices between CHWN and NHWC when evaluating convolutional layers during the A ↔ G conversion. PIGOEN proposes a two-step conversion mechanism. The first step is to use a custom implementation that transposes between CHWN and NCHW layouts. Then we convert the NCHW layout to an NHWC layout, which is an existing function provided by CUTLASS [40].

The pseudo-code for the customized transpose function is provided in Algorithm 4. To perform the CHWN to NCHW conversion, the transpose function is invoked with $r = C \times H \times W$ and $c = N$, while the inputs and outputs are contiguous blocks of memory ordered with respect to the respective data layout.

Algorithm 4 NCHW ↔ CHWN Transpose

Require: Matrix I in CHWN resp. NCHW layout.
Ensure: Transposed matrix O in $NCHW$ resp. $CHWN$ layout.

Set c and r to the number of columns and rows of I.
if c mod TILE_DIM $\neq 0$ **then**
    $c \leftarrow c + (\texttt{TILE\_DIM} - (c \bmod \texttt{TILE\_DIM}))$
if r mod TILE_DIM $\neq 0$ **then**
    $r \leftarrow r + (\texttt{TILE\_DIM} - (r \bmod \texttt{TILE\_DIM}))$
grid_dim $\leftarrow (c/\texttt{TILE\_DIM}, r/\texttt{TILE\_DIM}, 1)$
block_dim $\leftarrow$ (TILE_DIM, BLOCK_ROWS, 1)
Launch GPU `Transpose Kernel` with: ▷ Algorithm 5
        grid dimensions = grid_dim
        block dimensions = block_dim
        arguments = (O, I, c, r)

Note: By default, we set TILE_DIM = 64, and BLOCK_ROWS = 16 in our implementation.

Algorithm 5 Transpose Kernel

Require: O, I, c, r.

Declare shared memory tile[TILE_DIM][TILE_DIM]
$x \leftarrow$ BId.x × TILE_DIM + TId.x
$y \leftarrow$ BId.y × TILE_DIM + TId.y
for $i = 0$ to TILE_DIM step BLOCK_ROWS **do**
    if $x < c$ and $(y + i) < r$ **then**
        tile[TId.y + i][TId.x] $\leftarrow$ input$[(y + i) \times c + x]$
Synchronize threads
$x \leftarrow$ BId.y × TILE_DIM + BId.x
$y \leftarrow$ BId.x × TILE_DIM + BId.y
for $i = 0$ to TILE_DIM step BLOCK_ROWS **do**
    if $x < r$ and $(y + i) < c$ **then**
        output$[(y + i) \times r + x]$ $\leftarrow$ tile[BId.x][TId.y + i]

Note: *BId* and *TId* refer to CUDA's builtin *blockIdx* and *threadIdx* variables

E Making PPML More Accessible

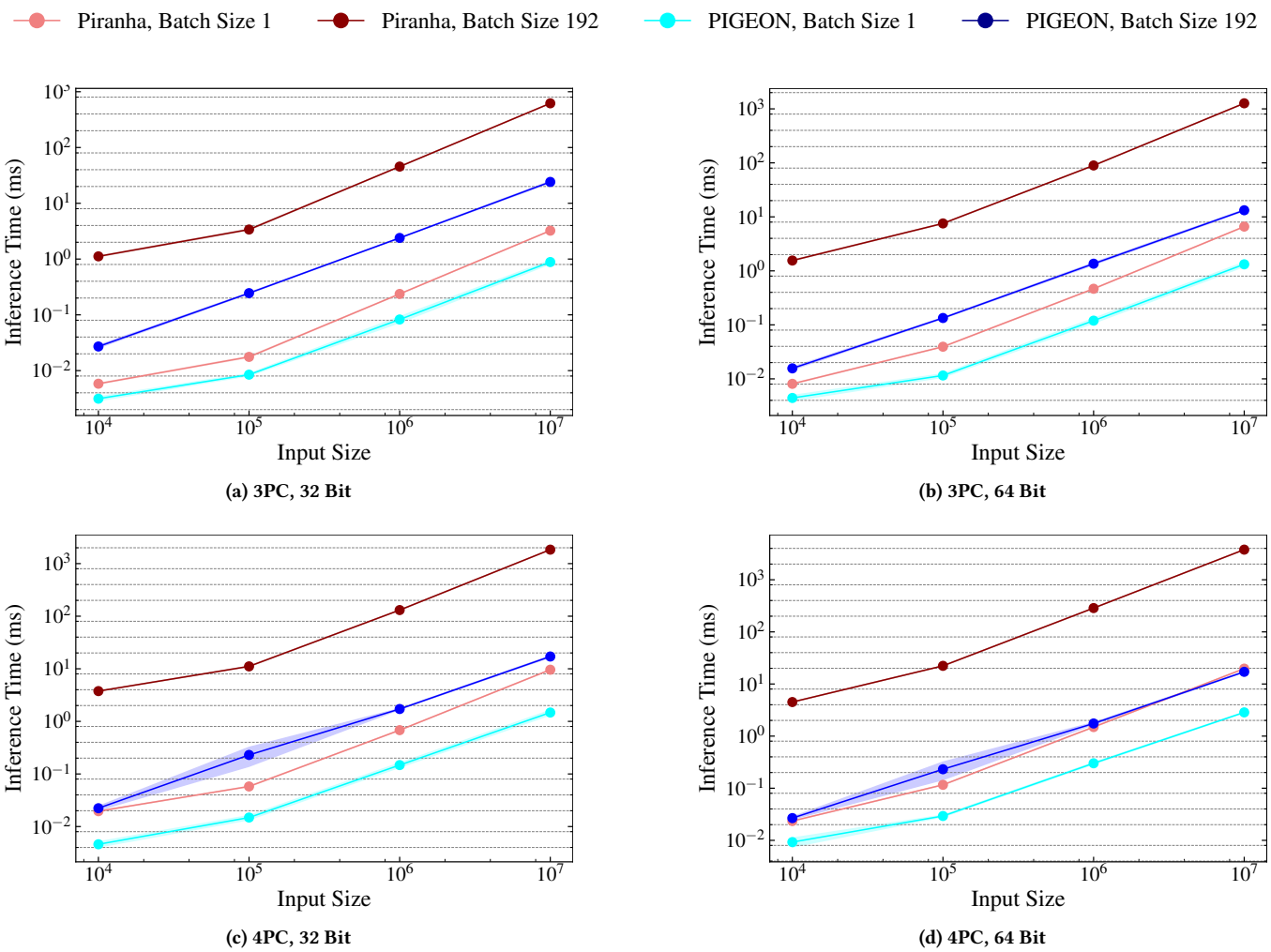
In order to support ImageNet inference with large batch sizes, existing GPU frameworks require High-End GPUs. While CPU-only frameworks such as FALCON [52] exist, they achieve less than one order of magnitude of throughput for batched inference of large neural networks than their GPU-only alternatives [54]. 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 a respectable convolution throughput of over 5Gbit/s by utilizing the techniques described in §4.3. Table 4 shows that even PIGEON CPU achieves one to two orders of magnitude higher throughput than Piranha. For models evaluated on small datasets such as 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 few compute resources. In §F, 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 tables show 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's modular design allows developers of different domains to extend either the neural network, MPC, or core software components of PIGEON while being oblivious to our presented optimizations.

F Additional Benchmark Results



(a) 3PC, 32 Bit

(b) 3PC, 64 Bit

(c) 4PC, 32 Bit

(d) 4PC, 64 Bit

Figure 7: ReLU Inference Time with Different Bitlengths

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

Table 7: Throughput and Runtime for End-To-End Inference of 192 Images, 64 Bit

Setting	Framework	CIFAR-10			ImageNet	
		AlexNet	ResNet50	VGG16	ResNet18	VGG16
		Throughput (Images per second)				
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
		Runtime (Seconds for 192 images)				
3PC	Piranha	10.67 ± 0.04	128.00 ± 0.00	37.80 ± 0.07	243.04 ± 3.08	1745.45 ± 0.00
	PIGEON CPU	**0.20** ± 0.00	**2.34** ± 0.05	**2.50** ± 0.04	**54.70** ± 0.93	**108.47** ± 1.23
4PC	Piranha	45.82 ± 0.00	369.23 ± 0.00	103.23 ± 0.00	-[a]	4800.00 ± 0.00
	PIGEON CPU	**0.28** ± 0.01	**2.38** ± 0.03	**2.53** ± 0.05	**69.82** ± 0.76	**137.14** ± 3.92

[a] Runtime error

Table 8: Single-Core Runtime (Seconds), 32 Bit

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] Runtime 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), 64 Bit

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

Table 10: CryptGPU Benchmark: Throughput for ImageNet Inference, 3PC, 64 Bit

Framework	Throughput (Images per second)
PIGEON CPU	1.77 ± 0.02
CryptGPU[a]	0.022 ± 0.00012

[a] Highest supported batch size: 3