Systems and Algorithms for Efficient, Secure and Private Machine Learning Inference

HARVARD UNIVERSITY

Graduate School of Arts and Sciences

DISSERTATION ACCEPTANCE CERTIFICATE

The undersigned, appointed by the

Harvard John A. Paulson School of Engineering and Applied Sciences
have examined a dissertation entitled:

"Systems and Algorithms for Efficient, Secure and Private Machine Learning Inference"

presented by: Maximilian Lam

Signature Michael Mitzenmacher
Typed name: Professor Michael Mitzenmacher

Signature Edward E Suh
Typed name: Professor Edward Suh

Signature
Typed name: Professor Vijay Janapa Reddi

Signature
Typed name: Professor Gu-Yeon Wei

Signature
Typed name: Professor David Brooks

March 25, 2024

Systems and Algorithms for Efficient, Secure and Private Machine Learning Inference

A DISSERTATION PRESENTED
BY
MAXIMILIAN LAM
TO
THE SCHOOL OF ENGINEERING AND APPLIED SCIENCES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
IN THE SUBJECT OF
COMPUTER SCIENCE

HARVARD UNIVERSITY
CAMBRIDGE, MASSACHUSETTS
MARCH 2024

Dissertation advisor: Vijay Janapa Reddi, Gu-Yeon Wei, David Brooks Maximilian Lam

Systems and Algorithms for Efficient, Secure and Private Machine Learning Inference

Abstract

As artificial intelligence and machine learning become ubiquitous, data privacy emerges as a critical concern. The use of sensitive data in machine learning applications exposes vulnerabilities that could jeopardize user privacy, posing ethical and legal risks. Current machine learning systems require significant modifications to protect privacy, such as on-device computation or encryption, which increases computational costs and may reduce accuracy, posing an issue for deployment. These computational challenges are the key barrier to adoption and addressing the challenges at this intersection of machine learning, data privacy, and computational efficiency is essential for the future deployment of privacy-enhanced machine learning systems.

My PhD focuses on this unique intersection of machine learning, data privacy and systems, with the high level aim of making privacy-enhanced machine learning techniques efficient enough to be deployed. Over the course of my PhD I have developed systems and algorithms that accelerate by up to an order of magnitude techniques for privacy-preserving machine learning inference, such as on-device machine learning inference and secure neural network inference, by leveraging unique aspects of neural networks like quantization, harnessing systems and hardware acceleration techniques like GPU acceleration, and co-designing these hardware-software optimizations with the specific privacy preserving machine learning algorithm to obtain maximal efficiency at inference time, all while remaining cognizant of and defending against potential attack vectors (i.e: data privacy leaks, such as by gradients in federated learning) that may compromise the security and privacy of the machine

learning system. My PhD pushes the boundary of solutions towards machine learning systems that are simultaneously efficient, private and secure.

Towards efficiency, we develop PrecisionBatching, a general neural network acceleration technique which utilizes quantization to accelerate neural network inference by up to 2× through maximizing GPU utilization on inference over small batchsizes by turning a memory-bound operation into a compute-bound operation. Although PrecisionBatching makes non-private neural network inference more efficient, it is a crucial step towards realizing the unique application of quantization towards privacy-preserving machine learning systems, as well the criticality of leveraging hardware acceleration, specifically GPU acceleration, for obtaining maximal system performance.

Towards privacy, we develop Tabula, an approach which utilizes quantization to enable the use of secure lookup tables to speed up the private computation of activation functions for neural networks by over 100×. Tabula enables private neural network inference that is over an order of magnitude more efficient in terms of runtime and communication than prior works, enabling the real-world deployment of secure neural network inference applications. We furthermore develop GPU-DPF, a GPU algorithm that accelerates distributed point functions (DPF) for private information retrieval by over 30× over a CPU by harnessing massive parallelization towards computing expensive cryptographic primitives, for the purpose of enabling private on-device machine learning inference with embedding tables too large to store on-device.

Finally, towards security, we develop Gradient Disaggregation, an attack that the disaggregates sums of gradients of up to thousands of users that are observed during federated learning for the purpose of undermining the privacy safeguards of federated learning systems, and furthermore propose possible defenses against our attack, with the high level goal of developing machine learning systems that are more secure.

Contents

Previous work

Portions of this dissertation appear in the following works:

Maximilian Lam, Zachary Yedidia, Colby Banbury, Vijay Janapa Reddi. "Quantized neural network inference with precision batching". 2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT 2021).

Maximilian Lam, Michael Mitzenmacher, Vijay Janapa Reddi, Gu-Yeon Wei, David Brooks. "Tabula: Efficiently computing nonlinear activation functions for secure neural network inference".

Maximilian Lam, Jeff Johnson, Wenjie Xiong, Kiwan Maeng, Udit Gupta, Minsoo Rhu, Hsien-Hsin S Lee, Vijay Janapa Reddi, Gu-Yeon Wei, David Brooks, Edward Suh. "GPU-based Private Information Retrieval for On-Device Machine Learning Inference". 29'th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2024).

Maximilian Lam, Gu-Yeon Wei, David Brooks, Vijay Janapa Reddi, Michael Mitzenmacher. "Gradient disaggregation: Breaking privacy in federated learning by reconstructing the user participant matrix". 38th International Conference on Machine Learning (ICML 2021).

1

Introduction

1.1 The Growing Importance of Data Privacy in the Future of the Information Age

Machine learning systems power increasingly many aspects of the applications and devices that we interact with in our daily lives. For example, search [82], social networks [219], ad/product recommendation [165], and text-completion on mobile devices are all embedded with machine-learning technolo-

gies that are critical to their function.

A natural consequence of this is that users' behavioural data is being collected at an increasingly finer granularity than ever before driven by demand for these applications – and this trend will likely continue as machine learning systems are increasingly deployed across services and applications. Simultaneously, machine learning systems are becoming increasingly powerful as their information processing abilities scale with the advances in computational hardware and the amount of hardware applied towards learning, allowing the learning of signals from massive amounts of datasets. Combined, this phenomenon poses a serious risk to consumers and individuals in the future of the information age: not only is more of our data being collected, but the algorithms are getting better at learning from them. Hence, in the limit, machine-learning systems of the future will infer increasingly more about us, and such fine granularity knowledge may pose a serious risk to the general populace should these systems fail or be compromised – mitigating these risks in case of a catastrophic failure or a security compromise is a fundamental challenge facing the machine-learning systems of the future.

The central aspect of this challenge lies in the key ingredient that powers these learning systems: data. Naturally, data privacy, which is fundamentally about control over data, will thus be a key issue as it is the data which are the unique and critical ingredient that determines whether a learning system is effective or not. Intuitively, the importance of data privacy will scale with these systems' learning ability, which is the product of the processing power of these systems and the amount of data that is collected, both of which will only increase in the future.

Consequently, data privacy laws that have been introduced across the world in this decade[62,209,221,231] will be used to regulate machine learning systems, for the purposes of ensuring security in the case of failures, for the purpose of enforcing pre-existing laws like HIPAA, GDPR and PIPL, and for competitive purposes like obtaining a regulatory moat. To a more specific degree, this implies that in the future there will be a greater demand for techniques that ensure data privacy in machine learn-

ing systems to safeguard users' information and limit the growing power of this technology. A dual effect is that data privacy will be essential not just for the purpose of limiting the growing power of this technology but also for adapting these systems to meet the regulatory challenges that might occur in an age where data privacy regulations may have considerable impact on the deployment and efficacy of these learning systems.

1.2 A New Frontier: Systems for Efficient, Secure and Private Machine Learning

Unfortunately, machine learning systems today are ill-equipped to begin to support privacy-enhancing features – for example, machine learning systems today, when performing inference, require that users' data be sent in the raw to application servers to perform inference; this poses a considerable risk to users in situations where the data may be privacy sensitive (for example, user preferences in ad recommendation, or health information for a medical application) particularly in cases where these machine learning systems might fail or be compromised, leaking private information to malicious actors. The key challenge is computational: there exists algorithms for enhancing machine learning systems with privacy safeguards (i.e: allow the inference over user data without revealing user data to the application servers), but they are too slow to be practically used. Consequently, the next frontier for privacy in machine learning is the development of algorithms and systems that are efficient enough to be practically deployed, but are also private and secure to being compromised.

1.3 Thesis Direction

My PhD focuses on pushing the boundary of solutions at this frontier. Specifically, my PhD consists of works that develop approaches for making private machine learning inference, such as on-device machine learning inference and secure neural network inference, efficient enough to be prac-

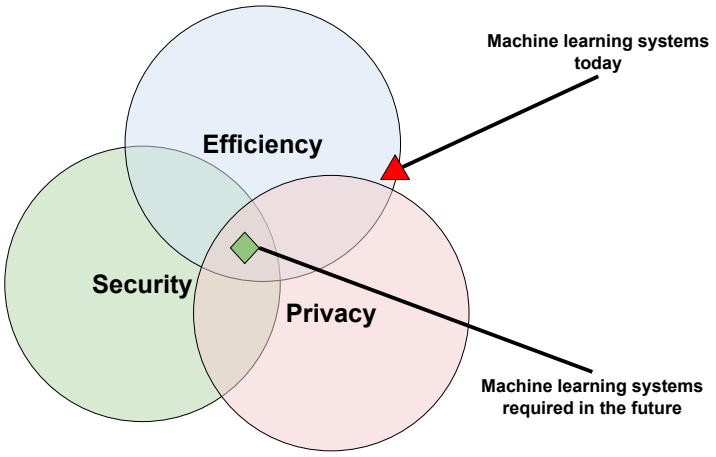


Figure 1.1: My PhD thesis pushes the boundary of solutions at the intersection of efficiency, privacy, and security. Over the course of my PhD I developed approaches for making private machine learning inference, such as on-device machine learning inference and secure neural network inference, efficient enough to be practically deployed, by leveraging unique aspects of neural networks like quantization, harnessing the power of hardware acceleration like GPU parallelism, and jointly applying these techniques towards the specific privacy preserving algorithm to ensure maximal efficiency, all while being aware of security risks (such as information leakage from gradients) that may compromise the safeguards of data privacy of the system.

tically deployed, by leveraging unique aspects of neural networks like quantization, harnessing the power of hardware acceleration like GPU parallelism, and jointly applying these techniques towards the specific privacy preserving algorithm to ensure maximal efficiency, all while being aware of security risks (such as information leakage from gradients) that may compromise the safeguards to data privacy of the system. Broadly, my PhD focuses on three key requirements for privacy-enhanced machine learning systems of the future:

- **Privacy:** Machine learning inference systems, particularly those that employ neural networks, should mitigate the amount of raw data (i.e: raw data that are inputs to the neural network) that is collected by the central servers hosting the system, and preferably collect no data for the purpose of performing inference. Note that in context of this thesis we do not focus on information leakage that occurs from the predictions of the model (though this is a

considerable and central challenge in the broader goal of securing machine learning systems).

- **Efficiency:** Machine learning systems, particularly those that employ neural networks, enhanced with privacy must be efficient enough at inference time to be practically deployed to the real world. Concretely, these models must be efficient in terms of both speed, memory consumption, and storage usage.

- **Security:** Machine learning systems should be cognizant of and secure to attacks that might undermine data privacy. Specifically, in the context of training (i.e: federated learning), systems should be aware of attacks that might undermine individual's data privacy and take appropriate measures to mitigate these security risks that threaten privacy.

My PhD thesis contributes novel solutions that push the boundary of this frontier (Figure 1.1). Today, machine learning systems can be categorized as being only efficient (though scale and efficiency may still be improved). The target to reach is systems that are simultaneously efficient, private, and secure. The works in this thesis aim to present solutions straddling both algorithms and hardware, bridging both machine learning and cryptography to get closer to this target.

1.4 Thesis Contributions

We present the following works that push the frontier of efficiency, privacy, and security for neural network based machine learning systems (Figure 1.2).

- **Quantized Neural Network Inference with Precision Batching**

  A general method for accelerating neural network inference on small batch sizes, utilizing GPU acceleration and the unique property that neural networks may be heavily quantized without accuracy loss.

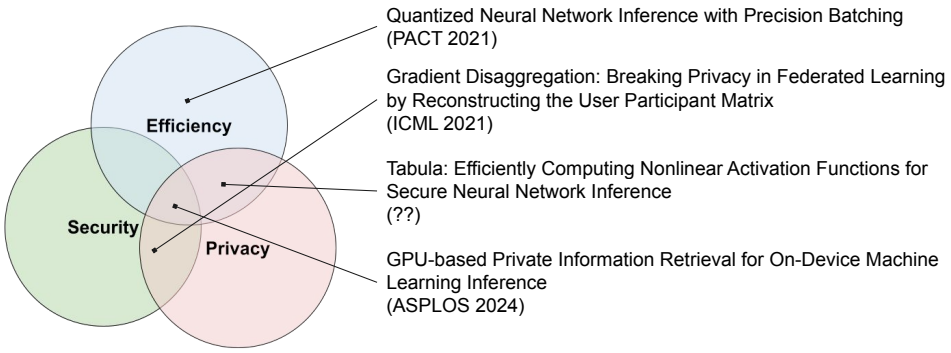


Figure 1.2: Thesis works contributing towards the frontier of efficient, private and secure machine learning systems.

- **Tabula: Efficiently Computing Nonlinear Activation Functions for Secure Neural Network Inference**

  An algorithm for accelerating the private computation of nonlinear activation functions, a main system bottleneck for private neural network inference.

- **GPU-based Private Information Retrieval for On-Device Machine Learning Inference**

  A GPU algorithm for accelerating private information retrieval, specifically distributed point functions, in context of privacy enhanced machine learning systems that require private accesses to large embedding tables.

- **Gradient Disaggregation: Breaking Privacy in Federated Learning by Reconstructing the User Participant Matrix**

  An attack on federated learning that uncovers individual gradients from sums of gradients, breaking individuals' data privacy during federated learning training.

My thesis starts with **PrecisionBatching** to accelerate non-private neural network inference by leveraging two key aspects of neural network based machine learning systems: 1) that neural networks may be heavily quantized with minimal accuracy loss and 2) utilizing dedicated hardware, specifically the GPU, to accelerate computation. We next use these key ideas and apply them in two separate ways towards privacy-enhanced neural network inference: first by leveraging quantization to accelerate a major bottleneck in private neural network inference (**Tabula**), and by leveraging GPU acceleration to accelerate private information retrieval in machine learning systems requiring private accesses to large tables (**GPU DPF**). Finally, we undermine data privacy security in federated learning with **Gradient Disaggregation**, demonstrating the need for new algorithms to secure these machine learning systems.

1.5 Thesis Roadmap

(Chapter 2) Background and Related Work introduces the relevant literature on existing methods for privacy-preserving machine learning as well as machine-learning / neural network based performance optimizations that will be relevant in the thesis. We specifically introduce methods spanning across multi-party computation, homomorphic encryption, quantization and federated learning.
(Chapter 3) Quantized Neural Network Inference with Precision Batching develops a method for accelerating neural network inference on small batch sizes by 1) leveraging the unique property that neural networks may be quantized and 2) utilizing GPUs to accelerate computation. The fundamental insight is that inference with small batch sizes is bottlenecked by memory operations rather than arithmetic operations; by reframing the computation of a neural network in a bitserial manner, recasting it as a sequence of 1-bit matrix-matrix operations (arithmetic heavy), and quantizing the resulting terms, we can gain an overall speedup by fully leveraging the arithmetic capabilities of the hardware, while sacrificing a bit of accuracy. While this work does not touch upon privacy,

it introduces two key and unique properties of modern neural network based machine learning systems that are key to subsequent works that accelerate privacy based machine learning systems: 1) neural networks may be quantized while maintaining reasonable accuracy and 2) computation may be greatly accelerated by using specialized hardware, specifically GPUs.

(Chapter 4) Tabula: Efficiently Computing Nonlinear Activation Functions for Secure Neural Network Inference accelerates the private computation of nonlinear activation functions, the main performance bottleneck of private neural network inference, the goal of which is to perform inference on a client's data without leaking the model data to the client, nor the client's data to the server hosting the model. We leverage a key and unique property of neural networks introduced in a previous work: quantization – by heavily quantizing neural network activations, we can fit all possible function calls of the nonlinear function in a table, enabling the use of efficient secure lookup tables without running into issues imposed by exponential memory requirements. This work greatly accelerates the private neural network inference by eliminating a major system performance bottleneck.

(Chapter 5) GPU-based Private Information Retrieval for On-Device Machine Learning Inference accelerates two-server based methods for private information retrieval (PIR) in context of machine learning recommendation systems that require PIR for privately accessing large embedding tables. We develop novel GPU algorithms for accelerating the distributed point function (DPF) for PIR on a GPU, speeding up the overall operation by over two orders of magnitude. This work accelerates private neural network inference for applications that require private accesses to large embedding tables.

(Chapter 6) Gradient Disaggregation: Breaking Privacy in Federated Learning by Reconstructing the User Participant Matrix develops an attack to disaggregate sums of gradients observed by a central server during federated learning, which allows the server to recover individual users' input data, breaking privacy. Our method leverages client participation statistics (i.e: how

many rounds of federated learning a client participated in) to factor the participant matrix. Our attack demonstrates the importance of security in machine learning methods, and motivates new algorithms for distributed learning with privacy.

2

Background and Related Work

Privacy-enhanced machine learning systems require applying techniques from various disparate fields including machine learning, cryptography and multiparty computation. Here we provide a background and an overview of the topics that are relevant to understand the main body of the thesis. We describe background and related work on topics such as neural network quantization, private neural network inference with multiparty computation, private information retrieval, and federated learning.

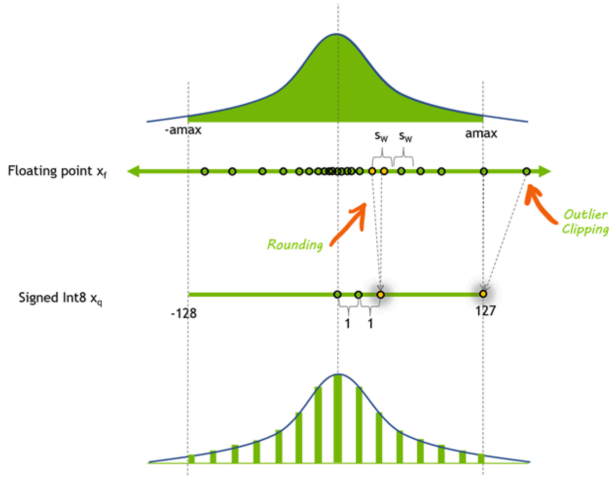


Figure 2.1: Example of linear quantization applied to the distribution of weights of a trained neural network. The distribution of weights are mapped linearly into a specified range so that the weights can be represented using lower-precision fixed point numbers, facilitating faster computational inference. Source: NVIDIA.

2.0.1 Neural Network Quantization

Neural network quantization is an optimization that reduces both the memory and compute requirement of neural networks by using lower precision floating point representation for the weights and activations of the network. Concretely, neural networks consist of a sequence of layers of the form $a_i = F_{nonlinear}(W_i a_{i-1})$ and both the weights W_i and activations a_i can be quantized to lower precision (this is termed weight quantization and activation quantization respectively). Generally, a trained neural network's weights are taken and quantized according to some quantization function, for example mapping the full precision values of the weights linearly and evenly across a limited set of points between a minimum and maximum value (which may be determined by analyzing the distribution of the pretrained weights). By reducing the precision of the weights and activations, neural network inference can be sped up by using fast, high-throughput lower precision operations (i.e:

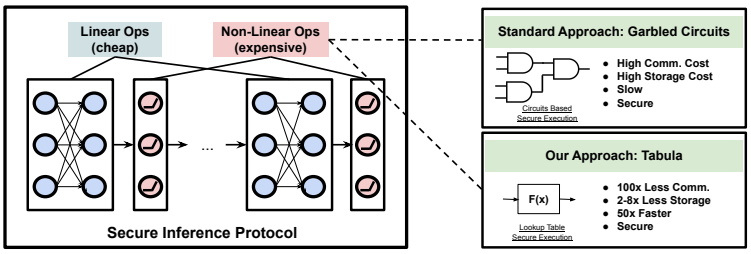


Figure 2.2: Private neural network inference is composed of linear and nonlinear operations. The main performance bottleneck is in computing the nonlinear operations; this thesis will present a method based on lookup tables to considerably reduce the costs of nonlinear operations.

8-bit floating point operations), and additionally the network's memory requirements are reduced as the weights require fewer bits of storage. However, this optimization comes at a cost: quantization introduces error into the weights and activations, which may negatively affect model accuracy. Surprisingly, neural networks have demonstrated resilience to being heavily quantized, with many works [119,39,45,57,146] showing that neural networks can be quantized down to 8-bit or even 4-bit and 1-bit values with little accuracy loss. Quantization is a unique and effective optimization for neural networks and is widely used in practice to speed up inference and reduce memory requirements [119,45,57,185]. Some notable works on quantization include [137,233,46,119,39,45,57]. We depict an example of quantizing the distribution of weights of a neural network in Figure 2.1. In this thesis we lever'age quantization as a unique feature of neural networks to both develop faster, more efficient machine learning inference, and also to develop more efficient privacy-enhanced machine learning systems.

2.0.2 Private Neural Network Inference with Multiparty Computation

Private neural network inference is the task of performing inference with a neural network on a client's data while ensuring that the client's data is not leaked to the application servers, and also ensuring that the neural network model is not leaked to the client. This capability is useful when op-

erating over privacy sensitive data such as health information or user preferences, and for situations where the the model might be proprietary and should not be leaked. Current methods for private neural network inference involve secure multiparty computation: both client and server interact and jointly compute the prediction over secretly shared inputs such that neither party can recover the full intermediate result (which could leak information about either the client's input or the application server's model). Broadly, secure multiparty computation operates over secret shares which guarantee that no intermediate information is leaked during the computation [203]. For example, a secret integer value x with values less than k may be encoded in a field of order k, $\mathbb{F}_k$, and arithmetically shared between two parties such that one party (the secret holder) holds $x - n \bmod k$ and the other party holds $-n \bmod k$; together the parties hold shares that sum to the secret value, but the second party learns no information about the value of x. The parties can subsequently perform simple primitive operations such as adding and multiplying secret shares [203,21], and this forms the basis of secure multiparty computation frameworks for neural network inference. Unfortunately, the main bottleneck in private neural network inference is performing nonlinear operations, and current techniques heavily utilize expensive cryptographic primitives like garbled circuits [170,120,132] for their computation. As such, these private inference systems cost considerable amounts of memory, compute and storage; for example individual ReLUs implemented as garbled circuits require over 2 KB of communication per scalar element during inference and impose over 17 KB of preprocessing storage per scalar element for each individual inference [170]. We depict a diagram showing the key operations in private inference (linear and non-linear operations) in Figure 2.2. In this thesis we make private neural network inference systems more efficient by optimizing nonlinear function calls utilizing unique aspects of neural networks, namely that they can be heavily quantized without accuracy loss.

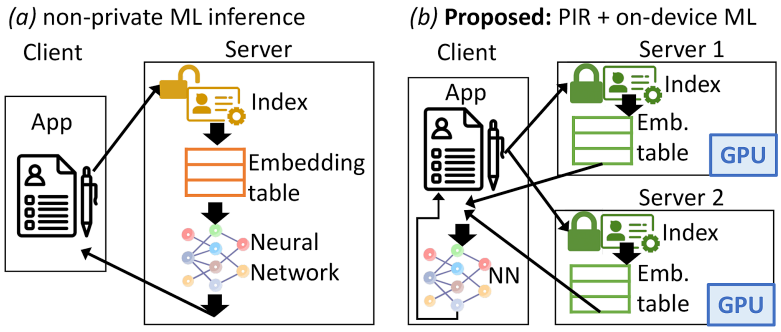


Figure 2.3: Private information retrieval for machine learning systems under a two-server PIR setup. In this setup, we duplicate large embedding tables too large to store on device across two non-colluding servers; a private distributed-point-function (DPF) query is issued to the two servers to privately retrieve an entry in the table, which are then inputted to the on-device network.

2.0.3 Private Information Retrieval in Machine Learning Systems

Private information retrieval (PIR) is the task of retrieving a particular entry from a table while ensuring that the table index is not leaked to the table holder; PIR is useful in privacy-enhanced machine learning systems that require accesses to large embedding tables. For example, recommendation systems often require accesses to large embedding tables with millions of entries[78,90], and making these systems private requires ensuring that the client's accesses into the embedding tables are kept secret from the application server; PIR is useful in this case to perform these embedding lookups the result of which are then used for private inference or on-device inference. PIR can be single-server or multi-server: single-server approaches involve just a single server hosting the table, but require expensive homomorphic-encryption techniques to ensure privacy. Multiserver approaches, on the other hand, duplicate the table across multiple non-colluding servers, and does not need homomorphic encryption computation but requires the extra trust assumption that the servers are non-colluding. In this thesis we focus on PIR in a two-server context using a distributed point function (DPF) (Figure 2.3): in addition to the client, there are two non-colluding servers where the embedding table is duplicated across[33], and to make a query the client issues two DPF

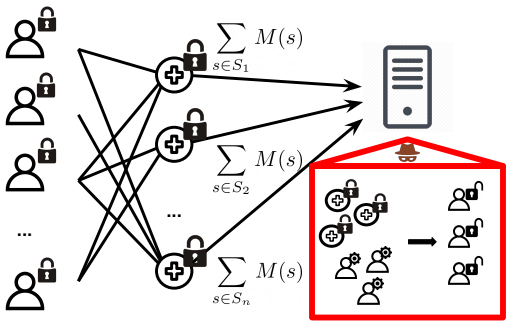


Figure 2.4: Federated learning involves multiple clients that submit gradient updates across numerous rounds (S), which are securely aggregated by the central server to update their model; privacy issues arise if the central server (red-hat) can see the updates in the clear.

keys (which together represent the index they wish to access, but individually is cryptographically private) to the two servers who perform computation to obtain secret shares of the entry the client requested. PIR with DPFs is a viable solution for privacy in machine learning systems that require lookups to large embedding tables (i.e: recommendation systems), however, the main bottleneck with DPFs is computational cost as they rely heavily on expensive cryptographic primitives like PRFs. Despite this, DPF based PIR techniques are considerably less expensive than homomorphic encryption based PIR approaches [160,33]. In this thesis we accelerate DPF based PIR approaches using GPUs for the purpose of enabling privacy-enhanced recommendation systems.

2.0.4 Federated Learning

Federated learning is a framework for collaboratively learning a single neural network model from multiple clients and works by repeatedly aggregating gradient updates across client devices (i.e: mobile phones) [81]. Privacy is enhanced as the user's raw inputs are not sent directly to the application servers hosting the neural network. Concretely, federated learning consists of a sequence of steps where each client device begins by locally computing a gradient update based on their local input and neural network, then sending these updates to the central server where these gradient updates

are aggregated or summed together and applied to the model, which is finally sent back to the devices. Federated learning is used in applications involving mobile devices and may be employed not only to collaboratively learn a shared model, but also to enhance privacy [166]. Unfortunately, recent lines of research show that gradient updates may leak considerable amounts of information about their inputs [244]; this finding has motivated the development of "secure aggregation" techniques that ensure that the server sees only the final sum of updates, rather than an individual update in the clear, which reduces information leakage [26]; this technique is critical in ensuring the privacy of the data of the users. We depict the overall federated learning with secure aggregation set up in Figure 2.4. In this thesis we focus on the security of federated learning frameworks from an attacker who has access to the central server, particularly by developing an attack that undermines the secure aggregation technique, violating user data privacy.

3

Quantized Neural Network Inference with Precision Batching

This chapter introduces **PrecisionBatching**, a technique for accelerating non-private neural network inference. The techniques introduced in this chapter are applied towards a non-private setting, however, these techniques (specifically, the use of neural network quantization, and hardware acceleration using GPUs) will be key ideas for the development of more efficient systems for private

machine learning.

3.1 Introduction

Recent advances in deep learning have demonstrated the wide range of the applications of neural networks [139,105,212,87,23,188,228,54], however, neural network execution remains computationally expensive. In the context of inference, where a trained neural network is executed to make predictions, these computational costs are even more significant as the quality of user facing products is often highly sensitive to the application's responsiveness. In these use cases, slow inference times not only degrade the usability of these applications (e.g: a robot arm must quickly be able to identify and manipulate an object; a text recognition system must react fast enough to ensure quality user experience; a voice recognition system must recognize speech quickly enough to enable real time interaction), but in some extreme cases may even restrict deployment (e.g: a drone must execute a neural network policy quickly to adapt to a changing environment, or risk crashing).

Research in quantization aims to reduce the computational costs of neural network inference by reducing the precision of neural network weights and activations [113,114,236,213,148,236,136], however, this technique incurs an increasingly larger accuracy penalty when quantizing to lower bitwidths due to quantization error [114,40]. Traditionally, without retraining, neural networks suffer significant accuracy degradation beyond 8 bit quantization [232,239], limiting speedups to ~4× the speed of the original network. With retraining, research has shown that networks may be quantized beyond 8 bits [113,114,48], however, retraining for quantization is computationally expensive, requires architectural changes to the network and converges slower [113]. Thus, in the context of quantization without retraining, it remains challenging to enable < 8 bit quantization without significantly degrading model quality, and, in the context of quantization with retraining, convergence time continues to be a major issue.

In this paper, we develop *PrecisionBatching*, a quantized inference algorithm for traditional hardware platforms to speed up low batch neural network inference. *PrecisionBatching* decomposes network weights and activations into 1 bit tensors, batches the 1 bit activations together, and performs quantized inference with low precision weights and high precision activations (see Figure 3.1). This attains speedups by reducing the precision of weight layers, maintains accuracy by keeping activations at higher precision, and utilizes the compute platform's higher arithmetic intensity to absorb the extra computation. Besides speedup, *PrecisionBatching* enables finer granularity control over the weight and activation precision of quantized inference, which may yield further speedups at a given accuracy. *PrecisionBatching* is a quantized inference method, a kernel, which specifies how a traditional compute platform (like a GPU) may efficiently perform quantized inference. This is unlike quantization algorithms such as[40,239,243] which specify how to quantize the network, but not how to execute over the lower precision values; thus *PrecisionBatching* may be used in conjunction with these quantization techniques.

We developed *PrecisionBatching* on three key observations:

- **Insight 1: Small Batch Neural Network Inference is Memory Bound**

  User facing products perform network inference with a small batch size to reduce response time / latency (it is not uncommon to see a batch size of 1)[95,18,88]. However, on traditional hardware platforms like GPUs, memory transfer speeds are much slower than arithmetic compute capabilities (FLOPs), so performing neural network inference with low batch size is *memory bound*, meaning most of the time executing the network is spent on fetching data, rather than on arithmetic computations[91,95]. Specifically, in a regime with batch size 1, the data bottleneck is transferring the weight layers of the neural network, which incurs communication cost on the order of $O(mn)$ where m and n are sizes of the network's hidden layers; conversely, the memory cost of transferring activations is significantly less at $O(m + n)$.

Observing that small batch neural network inference is memory bound is significant for two reasons: 1) it indicates that during neural network inference, the compute cores of the hardware platform are idle, suggesting that one may attain free compute cycles during this duration and 2) significant speedups may be attained by reducing the time spent on transferring weights.

- **Insight 2: More Bits for Activation Precision Improves Model Quality**

  Quantization literature has shown that using more precision for activations improves model quality[40,239]. Intuitively this makes sense, for example, between a network with 4 bit weights and 4 bit activations and a network with 4 bit weights and 8 bit activations one would expect the one with higher precision activations to attain higher accuracy. Additionally, from insight 1, a single bit of precision for weights does not hold the same value as a single bit of precision for activations. Specifically, as inference is weight memory bound, reducing the precision of weights by a single bit is much more valuable than reducing the precision of activations by a single bit. Unfortunately, on traditional hardware platforms like GPUs, kernels fail to capitalize on this insight by requiring both operands of a computation (weights+activations) be the same precision.

- **Insight 3: Matrix Multiplication may be Decomposed Bitserially**

  Full precision matrix vector multiplication may be decomposed into a sum of 1 bit matrix-matrix multiplications; this logical decomposition is known as bitserial computation in the hardware architecture space[7,205]. In the context of traditional hardware platforms like GPUs, implementing such a routine incurs significantly more arithmetic as the terms of the sum are separated out, which may be a reason why, to the best of our knowledge, such a kernel is not used widely. However, leveraging insights 1 (inference is memory bound) and 2 (activations improve model quality) we can see that such a bitserial kernel may yield sig-

nificant gains: the extra arithmetic incurred by the routine is absorbed for free by the idle compute units and now one may perform inference with higher activation precision and lower weight precision, reducing the weight memory-boundedness of network inference and achieving a speedup.

To demonstrate the value of *PrecisionBatching*, we develop optimized computational kernels to perform our algorithm on the GPU and evaluate our method against standard quantized inference implementations (NVIDIA's Cutlass linear algebra library [183]) on various applications including fully connected networks for MNIST and reinforcement learning, and LSTMs/RNNs for language modeling and natural language inference. Across this range of applications and models we demonstrate significant end-to-end speedups over using standard quantized inference methods. We also extensively developed a CPU implementation, however we found that the lack of vectorized 1-bit operations (specifically, popcount), limited the memory boundedness of the operation, and yielded little speedup. We believe that future CPU hardware capabilities (and especially hardware accelerators) would enable these gains on the CPU, which we leave for future research.
In summary, our contributions are as follows:

- We develop *PrecisionBatching* an algorithm for quantized neural network inference targeted to traditional hardware platforms. *PrecisionBatching* enables quantized inference at lower bitwidths and achieves better speedup per accuracy over standard quantized inference without retraining.

- We evaluate *PrecisionBatching* over a variety of applications (MNIST, language modeling, natural language inference, reinforcement learning) and neural network architectures (fully connected, LSTM, RNN) and show net speedups of $> 10\times$ over the full precision baseline ($> 1.5\times$-$2\times$ over standard 8-bit quantized inference) within the same error tolerance. Furthermore, we leverage the finer granularity of precisions supported by *PrecisionBatching* to

boost speed vs model quality.

- We show how using higher precision activations for quantized models as enabled by *PrecisionBatching* allows faster retraining times and achieves higher quality.
- We release optimized GPU kernels for our algorithm (and corresponding baselines) in the form of PyTorch modules.

3.2 Related Work

3.2.1 Post Training Quantization

Post training quantization is the standard method for quantizing neural networks without retraining and involves clipping the values of a pre-trained model based on statistics[239]. Various methods for post training quantization have been researched. Naively, post training quantization involves casting weight and activation values to the nearest n-bit representation. More sophisticated techniques involve clipping the weights and activations so as to minimize some form of error between the quantized and real values[232,239]. Even more advanced techniques change the underlying floating point format to enhance speed/accuracy[214,141,122].

Pre-existing research in post training quantization methods often omit details as to how the resulting quantized weights/activations may be leveraged on existing CPU and GPU platforms to speed up inference. More unusual bitwidths (e.g: 2/3/4/5) lack a corresponding data type on traditional hardware platforms and hence it is unclear how these levels of quantization improve inference. The implied benefit of post training quantization methods on these bitwidths is either space/memory savings or deployment to specially developed hardware accelerators for which fixed point operations for various bitwidths may be developed. By framing n-bit fixed point inference operations as a sum of binary operations, *PrecisionBatching* is an effective solution to realize these quantization gains on

traditional hardware platforms. Hence, *PrecisionBatching* extends the memory-savings benefits of various post training quantization methods to speed gains on traditional hardware architectures.

3.2.2 PACT

The importance of activations in quantization quality has been noted in research. Specifically, PACT (Parameterized Clipping Activation for Quantized Neural Networks)[40] demonstrated that neural network weights and activations may be quantized to very low bitwidths (< 4) if an activation scale is optimized during training. Although PACT requires changes to the training process (and hence does not work out of the box), their research demonstrates the importance and difficulty of quantizing activations in maintaining quantization quality. Motivated by their findings, *PrecisionBatching* opts to keep activations in higher precision (8,16,32 bit) to maintain accuracy at very low quantization level. This comes at minimal cost during inference as compute is dominated by memory access times. Thus, *PrecisionBatching* circumvents the need to maintain a quantization scale at training time by giving more bits of precision to activations at inference time.

3.2.3 Outlier Channel Splitting

Recently, research into quantization without retraining has emerged as a topic of interest. One notable work is Outlier Channel Splitting[239], which eliminates large magnitude weights/activations (which increase quantization error) by splitting them into separate channels, then applying standard post training quantization on the splitted weights, improving quantization performance. Outlier Channel Splitting demonstrates better performance-per-bit by using their technique in conjunction with standard post training quantization methods. Importantly, the authors note that outlier channel splitting may also be done to activations at runtime, though this is computationally difficult as it requires repeatedly finding the maximum of a matrix and adding rows to it. *PrecisionBatching* elimi-

nates this need by using more bits to represent activations, improving accuracy. Like many standard post-training quantization methods, Outlier Channel Splitting may be applied along with *PrecisionBatching* to improve quantization quality and to extend their memory-saving gains to speed gains on traditional hardware platforms.

3.2.4 Bitserial Computation

Bitserial computation is a technique leveraged by *PrecisionBatching* for quantized inference and operates by decomposing fixed point operations into bitwise operations [124,7,205]. Bitserial computation frames n-bit fixed point operations as a sum of bitwise operations and accumulates the result layer by layer. This formulation is most popular in the hardware architecture space to develop specialized accelerators for machine learning and realizing the technique in this context requires dedicated hardware constructs. Various hardware accelerators that leverage the bitserial formulation to reduce energy costs include [124,7,205]. The bitserial formulation is less used in context of traditional hardware architectures (e.g: CPUs and GPUs) as it is less clear how the technique would perform as it greatly increases the total amount of arithmetic per operation, though early works such as [220,48] explore CPU implementations. The key insight behind *PrecisionBatching* is that on traditional architectures, particularly the GPU, low batched inference is heavily memory bound and by batching the decomposed 1-bit vectors the extra overhead in compute is negated by the reduction in memory accesses, effectively turning a memory bound problem into a compute bound problem, and yielding a net speedup.

3.2.5 Streamlined Deployment for Quantized Neural Networks

Another related work to *PrecisionBatching* is Streamlined Deployment for Quantized Neural Networks [220], which leverages a bitserial formulation to speed up deployment on the CPU. Similar to

PrecisionBatching, Streamlined Deployment for Quantized Neural Networks frames quantized operations in terms of 1-bit operations. However, the key difference is that Streamlined Deployment separates the the bitlayers of the activations into different product terms, rather than batching them into one large matrix multiplication. As shown in their paper, the impact is that both weights and activations must be kept in very low precision (e.g: 2-bit activations) due to the computational overhead of performing multiple matrix products, which naturally leads to significant degredation in accuracy. The key observation of *PrecisionBatching* is that activation bitlayers may be batched together into one single matrix and a single large matrix product may be performed over this batch at high efficiency. This allows quantized inference with activations at or near full precision with minimal computational overhead, enhancing quantization performance.

3.2.6 Automatic Generation of Quantized Machine Learning Kernels

Automatic Generation of High-Performance Quantized Machine Learning Kernels[47] leverages a similar bitserial decomposition of kernels as PrecisionBatching to automatically generate quantized kernels for machine learning applications. In their work,[47] build a compiler to generate quantized kernels and demonstrate a speedup on CPU hardware platforms. Our work on *PrecisionBatching* differentiates in several key respects. Firstly, we show that our quantized inference kernel can perform inference with higher precision activations and attain significant speedup-vs-accuracy benefits; the work in[47] do not explore the impacts of higher precision activation on model accuracy. Secondly, our method utilizes the GPU, while the work in[47] demonstrates their kernel on the CPU, so we attain better speedups as the GPU is more compute bound. Finally, our work leverages the observation that inference is memory bound to develop a more effective quantized inference kernel, while the work in[47] is primarily focused on building a compiler for kernel generation.

3.3 Precision Batching

We describe the mechanics behind *PrecisionBatching* including how to decompose matrix-multiplies of various weight/activation bitwidths to be amenable for computation on traditional hardware platforms. Additionally, we describe how to efficiently implement our algorithm on standard hardware platforms and furthermore describe how we implemented our baseline standard quantized inference kernels.

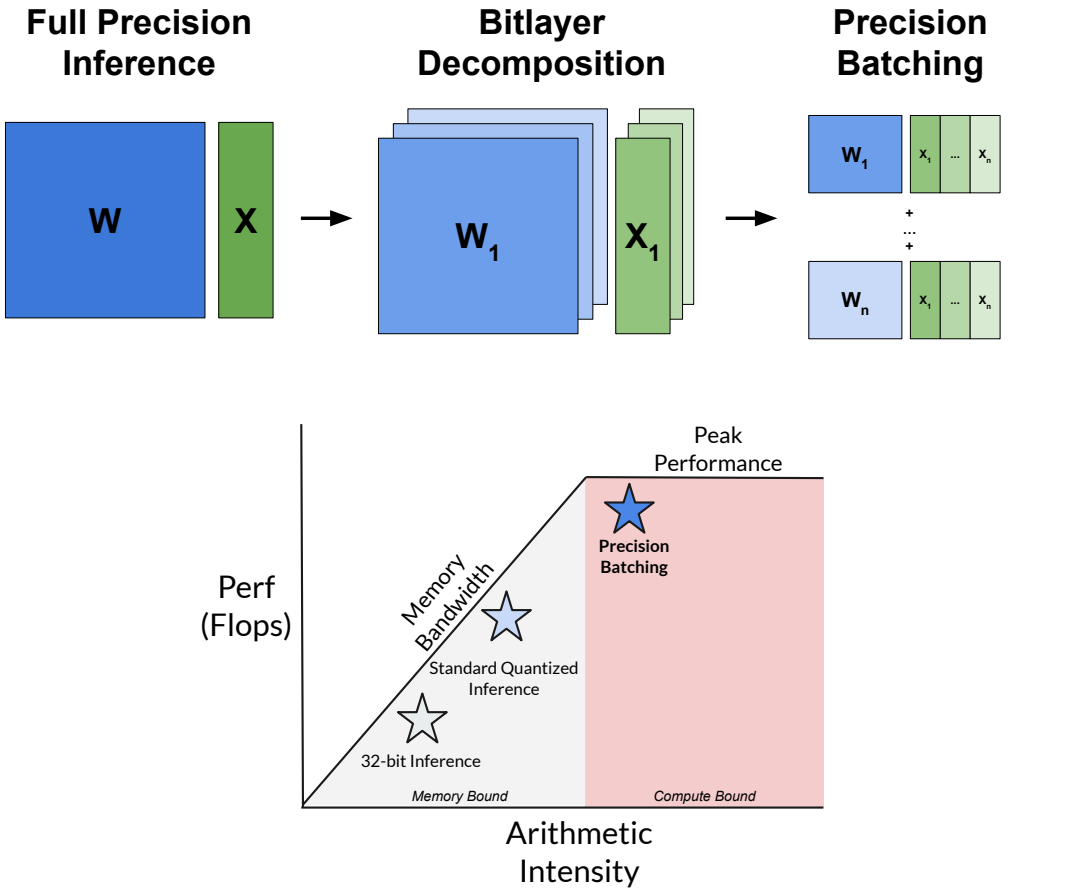


Figure 3.1: *PrecisionBatching* quantized inference decomposes weights and activations into 1-bit tensors and re-frames full precision matrix-vector multiplication as a sum of binary matrix-matrix opeartions, increasing the arithmetic intensity of the operation, improving computational efficiency.

3.3.1 Precision Batching Quantized Inference

PrecisionBatching decomposes weights and activations into 1-bit tensors and replaces the main matrix-vector multiplication operation with a sum of 1-bit matrix-matrix operations. Figure 3.1 presents a diagram showing the core mechanism behind *PrecisionBatching*. The core operation of neural network inference with a batch size of 1 is matrix-vector multiplication.

$$L_i(x) = Wx$$

L_i represents the function that transforms activation input x at the specific layer of the neural network and W is the trained weights of the neural network at layer i. Assuming that $W > 0$ and $x > 0$, we can decompose W and x into a sum of bitlayers (binary tensors) as in fixed point format

$$W = \frac{1}{2^{16}}(2^{n-1}W_0^{(b)} + ... + 2^0 W_{n-1}^{(b)}) \text{ where } W_i^{(b)} \in [0,1]$$

$$x = \frac{1}{2^{16}}(2^{k-1}x_1^{(b)} + ... + 2^0 x_k^{(b)}) \text{ where } x_i^{(b)} \in [0,1]$$

In the decomposition above, n and k represent the precision at which weights and activations are quantized to, respectively. Making n and k larger provides more accurate approximations of W and x. n describes the precision at which W is estimated and represents the number of bitlayers to accumulate. The fraction $\frac{1}{2^{16}}$ represents the location of the fixed point and enables representation of values 16 binary digits < 1. The fixed point may be changed depending on the scale of values of the weights and activations. Substituting back into the first equation and rearranging we get

$$L_i(x) = Wx$$

$$= \frac{1}{2^{32}}(2^{n-1}W_0^{(b)} + ... + 2^0 W_{n-1}^{(b)})(2^k x_1^{(b)} + ... + 2^0 x_k^{(b)})$$

$$= \frac{1}{2^{32}} \sum_{i=0}^{n} 2^{n-i-1} W_i^{(b)} (2^k x_1^{(b)} + ... + 2^0 x_k^{(b)})$$

One key observation is that the terms of the sum above can be rewritten as a single matrix multiplication. The idea is to batch together the bitlayer decomposition of x into a single matrix and to frame the equation as a sum of matrix-matrix products.

$$\frac{1}{2^{32}} \sum_{i=0}^{n} 2^{n-i-1} (W_i^{(b)} [x_1^{(b)} ... x_k^{(b)}]) [2^{k-1} ... 2^0]$$

The main workload $W_i^{(b)} [x_1^{(b)} ... x_k^{(b)}]$ exclusively consists of terms that are binary and facilitates efficient computation using 1-bit operations on CPU and GPU. Memory is reduced by a factor of approximately $\frac{32}{n}$, given that the matrix W dominates the majority of memory accesses. Note that the number of arithmetic ops is increased by a factor of k as separating out the sum induces more work. However, as the reformulation leverages batching, the cost of the extra compute is negated by the higher computational efficiency of the matrix-matrix multiplication, and the reduction in memory accesses yields a net speedup.

As indicated, by choosing n and k, any precision of weights and activations can be attained. In this paper k (activation precision) is set to either 8, 16 or 32. Note that higher activation precision does not linearly impact performance due to the increase in computational efficiency. However, for CPUs that are less efficient (more compute bound), setting k to be lower may significantly improve overall speed versus accuracy; hence k and n are parameters that determine the precision and speedup for quantized execution and may be tuned to the platform and requirement at hand. We analyze the impact of varying n and k on both speed and accuracy in the results.

Note that both the inputs and outputs of the *PrecisionBatching* algorithm (as well as intermediate values such as partial sum accumulators) are full precision. The overhead of maintaining inputs and outputs as full precision is minimal as much of the computational and memory costs are at-

tributed to large matrix multiply routines which are quantized (much of the memory costs are from loading the weights, rather than loading activations/inputs). Thus, keeping the intermediate inputs/activations in full precision is still aligned with the high level goal of speeding up inference.

3.3.2 Extending to Negative Values

We extend the formulation to any real valued W and x matrix (allowing negative values). Allowing any real valued input and matrix is important as it enables *PrecisionBatching* to handle weights with negative values and cases where the input is not passed through a positive activation function (e.g: the first layer of the neural network whose inputs are real and may potentially contain negative values). The simple but effective idea is to leverage two's complement by adding an extra bitlayer with a negative scale to handle negative values.

$$W = \frac{1}{2^{16}}(-2^{n-1}W_0^{(b)} + ... + 2^0 W_n^{(b)}),\ W_i^{(b)} \in [0,1]$$

$$x = \frac{1}{2^{16}}(-2^{k-1}x_0^{(b)} + ... + 2^0 x_k^{(b)}),\ x_i^{(b)} \in [0,1]$$

Here, the first bitlayer for both x and W are negated, allowing for a complete representation of values between $[-2^n, 2^n - 1]$. This formulation is logically equivalent to two's complement format. Note that this technique incurs an extra bitlayer of computational overhead (for weights) and thus increases the computational and memory costs; we found in practice that the extra bitlayer of computational overhead for activations is minimal.

3.3.3 Weight/Activation Quantization

In the *PrecisionBatching* formulation, W and x are converted into fixed point format and quantized to reduce computation and memory accesses. However, any standard post training quantization

Algorithm 1: PrecisionBatching Preprocessing

Input

W	Full precision weight matrix
n	Number of bits to quantize

Output

$W^{(b)}$	Bitlayers corresponding to quantized W
S	Scales corresponding to quantized bitlayers

$W_q \longleftarrow Int(QuantizeRound(W, n) \times 2^{16})$
$max\_bit \longleftarrow max(log_2(|W_q|))$
$W^{(b)} \longleftarrow [W_q \wedge (1 \ll i)$ for i in $max\_bit - n \,..\, max\_bit + 1]$
$S \longleftarrow [1 \ll i$ for i in $max\_bit - n \,..\, max\_bit + 1]$
$S[0] \longleftarrow S[0] \times -1$
return $W^{(b)}, S$

Algorithm 2: PrecisionBatching Inference

Input

$W^{(b)}$	Weight bitlayers
S	Weight bitlayer scales
x	Full precision input

Output

z	Full precision prediction

$z \longleftarrow 0$
$x_q \longleftarrow Int(x \times 2^{16})$
for $W_b, scale$ in $W^{(b)}, S$ **do**
    $z \longleftarrow z + \frac{scale}{2^{32}} \times (W_b x_q)[-2^{31}\ 2^{30} \,..\, 2^0]$
end for
return z

technique (e.g: KL divergence, MSE, etc) can be applied to W and x to improve accuracy, as long as the resulting set of quantization values are linearly spaced.

For applications, we use standard post training quantization before quantized execution.

$$Q(W) = d \times round\left(\frac{W}{d}\right), d = \frac{max(W) - min(W)}{2^n}$$

This rounds W to the corresponding closest n-bit representable fixed point values. We found that in practice, rounding produces significantly better results than truncation at very lower bitwidths (< 4 bits). Additionally, for quantizing to 1 bit, we found it extremely beneficial to exclude representing 0 and instead opt to represent a positive and negative value. After the n-bit rounding, $Q(W)$ is applied in the *PrecisionBatching* algorithm where the corresponding bitlayers and scales are deduced. Additionally, we also optimize over a clipping threshold to find a quantized matrix with the smallest mean error versus the full precision weight matrix. Note that quantizing W is a preprocessing step that is done offline and hence does not affect inference performance measurements.
The full *PrecisionBatching* algorithm is broken into two stages: a preprocessing step which converts full precision weights to bitlayers, listed in algorithm 1, and the inference stage which makes predictions given a full precision input, listed in algorithm 2.

3.3.4 Efficient Implementation

As indicated above, the core computation is an accumulation of products of binary tensors.

$$W_i^{(b)}[x_1^{(b)}...x_k^{(b)}]$$

As all values are 0 or 1, memory is reduced by packing the 0s and 1s into the bits of an integer array, yielding $32\times$ reduction in memory for each product of bitlayers. Operating over these packed formats is inspired by standard binary quantized neural networks which uses logical operations and popcounts for implementing multiply accumulate. An important difference is that typical binary quantized neural network weights contain values that are -1 or 1 rather than 0 or 1. Hence, instead of the *xnor* operation we use the *and* operation to simulate 1-bit multiplication. This is an important distinction for current and future hardware accelerators; current hardware accelerators (e.g: T4 binary MMA) perform the xnor operation rather than the and/or operation. Hence, in this work

we are only capable of leveraging basic GPU ands/popcounts rather than the accelerator, though using an accelerator would yield much better performance improvements due to heightened compute speed vs memory speeds.

To leverage binary operations to compute over full precision values, the floating point input vector must be converted to fixed point and then packed in such a way to layout the bits to be conducive to the *and/popcount* instruction. Conversion to fixed point is a simple multiply and cast. Rearranging the bits is done with a bitwise matrix transpose, for which there are efficient implementations on both CPUs and GPUs that leverage parallelism / SIMD. In practice, we found the bitwise matrix transpose to have negligible overhead. We furthermore note that multiple bitlayers may be stacked together so that the entire product across bitlayers can be performed with a single operation. However, in practice we found that there is negligible performance difference in accumulating multiple bitlayers separately, though a more optimized implementation may be the subject of analysis for future work.

3.3.5 Integer Quantized Inference

Standard quantized inference methods quantize both weight and activation to the same precision before execution (so that both operands are the same datatype); for example, 8-bit quantized execution quantizes both weights and activations to 8-bit ints before operation. Weights and activations are scaled down before quantization (so that the maximum value is representable in the quantized range), then dequantized after the operation. Like in *PrecisionBatching* we apply the same quantization preprocessing techniques (rounding, optimizing a clipping threshold) to weights before evaluation. In our experiments, we leverage NVIDIA's T4 tensorcore capability (via NVIDIA's Cutlass linear algebra library) in the implementation of the standard quantized inference baselines (1, 4, 8, 16, 32-bit inference methods).

3.4 Experimental Setup

This section describes the hardware we use to demonstrate *PrecisionBatching* in practice. We also describe the different neural network benchmark applications we use for evaluation.

3.4.1 Hardware Testbed

We perform all performance benchmarks and tests on NVIDIA's Tesla T4 GPU (as no previous GPU version supports 1/4/8 bit inference). For benchmarking kernels, we measure the wall-clock time of performing at least 1000 iterations of the target algorithm.
Note that the choice of using GPUs for *PrecisionBatching* is key: GPUs exhibit much higher compute vs memory capabilities than CPUs, which allows us to fully leverage *PrecisionBatching*'s higher operational intensity. Current CPUs exhibit a much lower compute vs memory ratio without vectorized popcount instructions and so on current generations of CPUs *PrecisionBatching* attains lower speedups, though with more advanced CPU architectures supporting vectorized popcounts we expect to see the same improvements.

3.4.2 Software Implementation

Baseline 4, 8 and 16 bit standard quantized inference utilizes the NVIDIA Cutlass library which performs low-precision matrix multiply using WMMA (warp matrix multiply accumulate) hardware operations that leverage Tensorcores for compute. We implemented *PrecisionBatching* using standard CUDA (no tensorcore acceleration). In all experiments the batch dimension is one, as we are targetting application scenarios for inference, where examples are processed one at a time where latency is important (rather than throughput).

3.4.3 Neural Network Benchmarks

We evaluate our method on the following applications: MNIST, language modeling, natural language inference and reinforcement learning.

- **MNIST** we train a 3-layer fully connected neural network with a hidden size of 4096 for 20 epochs, reaching a baseline accuracy of 98%. We uniformly quantize the weights and activations of each layer to the target precisions.

- **Language Modeling** We train a model with a 1-layer 2048 unit LSTM [105] as the encoder, and a 1-layer 2048 unit fully connected as the decoder (a common architecture used in language modeling [157]). We apply dropout with a factor of .5 to the inputs of the encoder LSTM's recurrence, and to the encoder LSTM's output. We train the model on the Wikitext-2 dataset [162] for 40 epochs, reaching a baseline perplexity of 93. During evaluation of quantization on model accuracy, we quantize the LSTM's input and hidden layers to the same weight and activation levels; however, we keep the final fully connected decoder in full precision (as it is not the main runtime bottleneck).

- **Natural Language Inference** We train a model with a 1-layer 3072 unit LSTM encoder and a 3-layer 3072 unit fully connected decoder (a larger version of that seen in [28]). We train on the SNLI dataset [28] for 10 epochs and reach a baseline accuracy of 78%. During evaluation of quantization on model accuracy, we uniformly quantize both the weights and activations of the LSTM encoder and the fully connected decoder to the target precisions.

- **Reinforcement Learning** We train models on reacher hard (easy), cheetah run (medium) and humanoid stand (hard) [218] using D4PG [106] with a 3-layer 4096 unit neural network (same, but larger, architecture in [218]) until convergence (task difficulties from [106]). We train on features rather than pixels. During evaluation we quantize all layers of the policy. An

episode is 1000 steps, (so maximum evaluation score is 1000, but this is not always attainable).

3.5 Results

Our results section is organized as follows. Firstly, we evaluate the performance of the *PrecisionBatching* kernel in isolation to verify that our quantized inference method attains similar or better speedup than standard quantized inference even with higher activation precision. Secondly, we verify (across various tasks) that higher activation precision yields better model quality than when activation precision is the same as weight precision (the case when using standard quantized inference). Then, we evaluate the end-to-end speedup vs accuracy benefits of *PrecisionBatching* over standard quantized inference. Finally, we evaluate the benefits of higher precision activations motivating *PrecisionBatching* for quantization with retraining.

3.5.1 Precision Batching Kernel Performance

We implement optimized GPU kernels for the *PrecisionBatching* algorithm and measure the speedup of the kernel over the full precision (32-bit) operation (provided by NVIDIA's Cutlass linear algebra library) across multiple precisions and matrix sizes. Inference times include all activation processing steps necessary for the algorithm, for example, transposing the activation bitmatrix before 1-bit execution.

Table 3.1 shows the performance of the *PrecisionBatching* kernel with weight bits $\in (1, 2, 4, 8)$ and activation bits $\in (8, 16, 32)$, along with baseline quantizated inference kernels (Int1, Int4, Int8, Float16, Float32). We see that at fewer bits, the *PrecisionBatching* kernel achieves significant speedups over full precision inference: 10-14x speedup for 1-bit, 5-7x for 4-bit (note that the optimal speedup for PBatch-n is $\frac{32}{n+1}$ with the sign layer taken into account). Using fewer activation bits

Table 3.1: Quantized inference speedups over 32-bit inference across different methods, matrix sizes and activation quantization levels on the NVIDIA T4 GPU. PBatch-n (a=k) means n+1 bitlayers are accumulated with k-bit activations. (n-bit weights, k-bit activations). We see that using more activations yields only minor slowdowns and demonstrates the memory-boundedness of low batched inference.

Method	512x512	1024x1024	2048x2048	4096x4096
PBatch-1 (a=8)	10.8	13.8	12.0	13.6
PBatch-1 (a=16)	9.5	12.1	10.3	13.2
PBatch-1 (a=32)	8.0	10.7	8.0	10.7
PBatch-2 (a=8)	6.6	9.9	8.3	11.8
PBatch-2 (a=16)	6.8	8.8	7.1	10.9
PBatch-2 (a=32)	5.7	7.5	5.4	8.3
PBatch-4 (a=8)	4.9	6.5	5.1	7.3
PBatch-4 (a=16)	4.2	5.5	4.3	6.8
PBatch-4 (a=32)	3.6	4.8	3.4	5.3
PBatch-8 (a=8)	2.9	3.6	3.2	4.7
PBatch-8 (a=16)	2.5	3.2	2.5	4.0
PBatch-8 (a=32)	2.0	2.7	2.1	3.1
Int1	3.6	5.0	8.5	34.3
Int4	3.6	4.7	5.8	11.0
Int8	3.3	4.0	4.2	8.0
Float16	2.3	1.8	2.0	2.8
Float32	1	1	1	1

increases performance only slightly as compute is not the main bottleneck in these operations. Generally, higher performance is seen at larger matrix sizes as the effect of the reduction in memory on performance is more pronounced. Baseline kernels (Int1, Int4, Int8 especially) perform much better at larger matrix sizes; we believe this is the case as their kernels are more optimized than ours and leverage Tensorcore capability for more efficient compute. This table is useful for roughly estimating the amount of speedup that may be attained on various applications. For example, if we believe for our application that inference with 4-bit weights, 16/32 bit activations would achieve the same accuracy as 8-bit weights and activations, then using *PrecisionBatching* on various matrix dimensions would yield a $1.5\times - 2\times$ speedup.

We additionally plot the operational intensity of *PrecisionBatching* versus standard inference in Figure 3.2. For each method, we compute its operational intensity: the number of operations that the

method performs, divided by the number of bytes of memory required for the method. Note that for standard inference, an operation is dependent on its bitwidth, for example, for 32-bit inference, a single operation is a 32-bit multiply or add, while for 8-bit inference, a single operation is an 8-bit multiply or add.

For *PrecisionBatching*, each 1-bit operation counts towards the computational ops. As shown in Figure 3.2, *PrecisionBatching* achieves much higher operational intensity as the number of compute operations is increased by a factor of the specified activation precision. Standard inference on the other hand achieves low operational intensity. Thus, *PrecisionBatching* achieves much greater efficiency than standard inference, and combined with the better model accuracy obtained by operating over higher activations, achieves better performance per accuracy threshold.

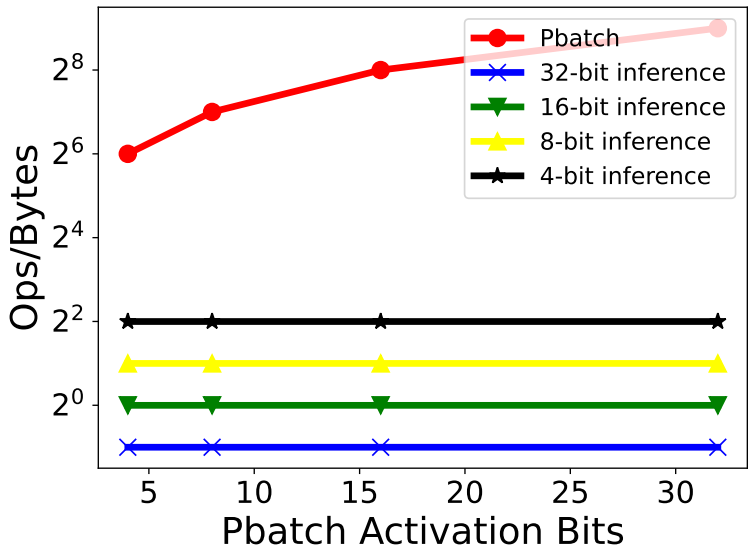


Figure 3.2: Operational intensity of *PrecisionBatching* versus standard inference. *PrecisionBatching* achieves higher operational intensity with more activation bits, enabling it to operate more efficiently on GPUs. Standard inference is primarily memory bound and achieves low operational intensity, resulting in lower compute efficiency.

Table 3.2: Benefits of using more precision for activations on model quality, evaluated on MNIST, language modeling (Wikitext-2), natural language inference (SNLI) and reinforcement learning (reacher:hard, cheetah:run, humanoid:stand). Generally, using higher precision activations allows quantizing twice as many bits with little degradation of model accuracy. Best scores per weight precision is bolded.

Task		$Q_{activ.} = 32$	$Q_{activ.} = 16$	$Q_{activ.} = 8$	$Q_{activ.} = Q_{weight}$
MNIST (acc.)	$Q_{weight} = 1$	85.8	86.7	**87**	10.1
	$Q_{weight} = 4$	97.1	**97.3**	**97.3**	94.3
	$Q_{weight} = 8$	**98.0**	97.8	97.8	**98.0**
	$Q_{weight} = 16$	**98.0**	97.9	97.9	**98.0**
	$Q_{weight} = 32$	-	-	-	98.0
Language Modeling (ppl.)	$Q_{weight} = 1$	**188.0**	188.0	188.0	828.1
	$Q_{weight} = 4$	**94.3**	**94.3**	**94.3**	148.9
	$Q_{weight} = 8$	**94.0**	**94.0**	**94.0**	**94.0**
	$Q_{weight} = 16$	**91.7**	**91.7**	**91.7**	92.8
	$Q_{weight} = 32$	-	-	-	92.8
Natural Language Inference (acc.)	$Q_{weight} = 1$	**76.1**	**76.1**	74.0	32.8
	$Q_{weight} = 4$	**78.7**	**78.7**	76.8	77.4
	$Q_{weight} = 8$	78.9	78.9	76.9	**79.1**
	$Q_{weight} = 16$	**78.9**	**78.9**	76.9	78.8
	$Q_{weight} = 32$	-	-	-	78.8
Reacher Hard (rew.)	$Q_{weight} = 1$	9.8	6.6	9.5	**24.9**
	$Q_{weight} = 4$	676.4	765.1	**960.1**	826.2
	$Q_{weight} = 8$	969.1	973.8	962.5	**976.4**
	$Q_{weight} = 16$	960.0	**974.6**	966.5	957.3
	$Q_{weight} = 32$	-	-	-	968.0
Cheetah Run (rew.)	$Q_{weight} = 1$	.8	**0.9**	0.8	0.0
	$Q_{weight} = 4$	616.3	**685.6**	651.6	480.3
	$Q_{weight} = 8$	700.3	**701.0**	681.6	700.7
	$Q_{weight} = 16$	706.1	**707.9**	682.4	705.4
	$Q_{weight} = 32$	-	-	-	702.9
Humanoid Stand (rew.)	$Q_{weight} = 1$	5.0	4.9	**7.4**	4.7
	$Q_{weight} = 4$	443.0	410.6	**466.9**	25.7
	$Q_{weight} = 8$	753.3	692.7	**816.0**	789.4
	$Q_{weight} = 16$	824.1	781.2	**864.2**	798.9
	$Q_{weight} = 32$	-	-	-	808.1

3.5.2 Accuracy Benefits of Higher Precision Activations

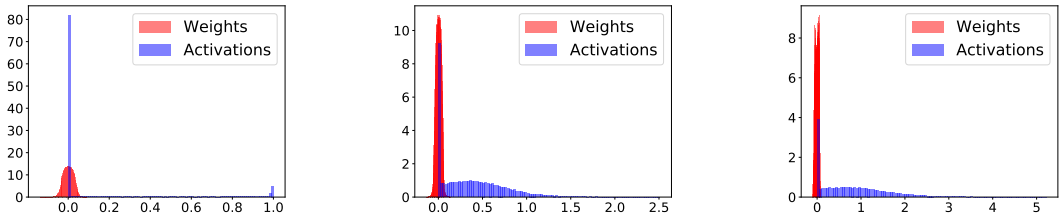
Next we show that using higher precision for activations leads to significantly better model accuracy at low bitwidths. We benchmark model accuracy across: MNIST, language modeling, natural language inference and reinforcement learning. For each we train one baseline full precision model and evaluate the effects of various levels of weight and activation quantization on the model's end performance. For each model/application we quantize weights and activations to 1, 4, 8, 16 and 32

bits. Note the $Q_{activ.} = Q_{weight}$ column uses standard quantized inference while the other columns use *PrecisionBatching*.

Table 3.2 shows model performance (accuracy for MNIST and natural language inference, perplexity for language modeling, reward for reinforcement learning) for different weight and activation precisions. For weight bitlevels < 8, keeping activations at higher precision (8, 16 or 32 bit) greatly increases model accuracy; generally, keeping activations at a higher precision allows quantizing twice as many bits, from 8-bits to 4-bits, without significant loss in model accuracy.

For MNIST, with 1-bit weights, using higher precision activations is the difference between 85% accuracy and random guessing (10% accuracy); with 4-bit weights, higher precision activations maintains within $< 1\%$ of the full precision model's performance. Similarly, for language modeling, with 1-bit weights, higher precision activations reduces perplexity from 800 to 180; for 4-bit weights, higher precision activations reduce perplexity from 180 to within a few points of the full precision performance. For natural language inference, using full precision activations allows us to quantize down to 1-bit with only a couple percentages of accuracy degredation (78% to 76%), whereas quantizing activations to 1-bit degrades to random guessing (33%). Interestingly, for language inference, the 8-bit quantized model outperformed the full precision result, a known phenomenon seen in quantization literature [136,236]. For reinforcement learning, trends are generally similar: better reward is attained with higher activation precision, though in some interesting cases (e.g: reacher hard), lower activation precision performed better. Harder tasks (e.g: humanoid stand) are generally more difficult to quantize and higher activations typically yield more consistent reward gains on such tasks. Additionally, on some tasks, the score achieved by weights=activations is dissimilar to that reported in the $Q_{activ.} = Q_{weight}$ column; this is due to slight differences in implementation between *PrecisionBatching* and standard quantized inference (e.g: we dynamically scale activations in standard quantized inference, whereas for *PrecisionBatching* a static scale is used).

Additionally, Figure 3.3 shows the histogram of values of both weights and activations on the



(a) Layer 1 **(b)** Layer 2 **(c)** Layer 3

Figure 3.3: Histogram of weights and activations of a 3 layer neural network on the MNIST task. Note that the plotted activations are the inputs to the corresponding matrix multiply operation with the weights. Besides the first layer (image input = activations), activations have a significantly wider distribution of values than weights, thus quantizing activations incurs more error and motivates using higher precision for activations.

MNIST task for each layer of the network. As shown, across all layers of the network, activation values have a much wider spread than the weights, with the exception of the first layer, for which the activations are the mnist input features. This indicates that quantizing activations would yield a significantly higher quantization error than for weights, and motivates keeping activations in higher precision.

3.5.3 End to End Performance Gains

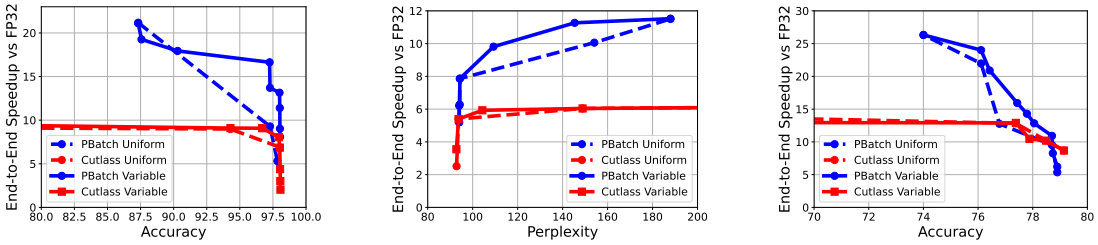
We demonstrate the end to end speedups achieved by *PrecisionBatching*. We combine the observations from our previous results: we leverage the high runtime performance of the *PrecisionBatching* kernel and the better model accuracy of keeping activations in higher precision to attain significant end-to-end speedups over the full precision model while maintaining model quality. We use the same applications (MNIST, language modeling (Wikitext-2), natural language inference (SNLI), reinforcement learning) with the same model architectures and training parameters described previously.

We apply each target quantized inference algorithm as follows. For the MNIST/reinforcement learning model, we replace each linear layer with the corresponding quantized inference algorithm;

for the language modeling and natural language inference Seq2Seq model, we replace each linear layer of the encoder 1-layer LSTM with the target quantized inference algorithm, however we keep the final fully connected decoder in full precision as it is not the main runtime bottleneck. For reinforcement learning, we replace each layer of the policy with the target quantized inference algorithm.

Additionally, for both baseline quantized inference and *PrecisionBatching*, on MNIST, language modeling and natural language inference, we use variable-bit quantization on different layers (e.g: 1-bit quantization on layer 1, 4-bit quantization on layer 2, etc) to further boost performance per accuracy. Accordingly, we perform an exhaustive grid search over weight/activation precision assignments. On the 3-layer fully connected for MNIST, for baseline quantized inference we assign each layer a precision $\in (1, 4, 8, 16, 32)$ (note that for quantized inference activations are the same precision as weights); for *PrecisionBatching*, we assign each layer a precision $\in (1, 2, 3, 4, 8)$ and activations $\in (8, 16, 32)$. On the Seq2Seq LSTM for language modeling and natural language inference, for baseline quantized inference we assign each layer a precision $\in (1, 4, 8, 16)$; for *PrecisionBatching*, we assign each layer a precision $\in (1, 2, 4, 8)$ and activations $\in (8, 16, 32)$. For the reinforcement learning tasks, we opt instead to maintain each layer with the same weights/activation precision; however, we leverage *PrecisionBatching*'s finer precision granularity in the evaluation (weight precision $\in (1, 2, 3, 4, 5, 6, 7, 8)$). In benchmarking the runtime performance of each model/application, we measure the wall clock time of inference with a batch size of 1 for 10 iterations on a given input repeated over 10 runs and take the minimum. We measure speedups by comparing the model with the target quantized inference algorithm against the model with the baseline quantized inference method.

Figure 3.4 and Figure 3.5 shows the Pareto curves of the end-to-end speedups of *PrecisionBatching* over standard quantized inference. On average, *PrecisionBatching* yields speedups of $8\times$ - $10\times$ over full precision inference, and $1.5\times$ - $2\times$ over standard 8-bit quantized inference at the same error

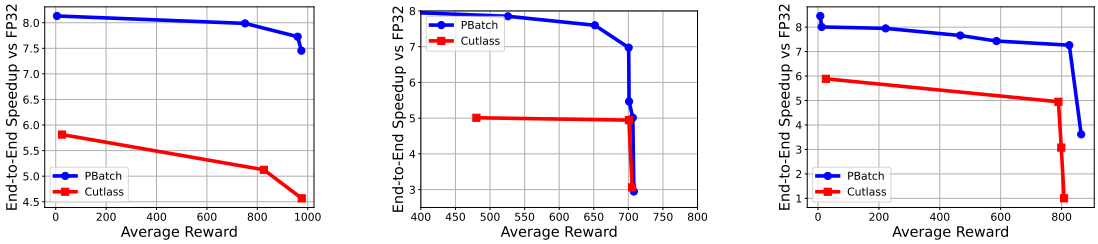


(a) MNIST (FC)

(b) Language modeling (LSTM)

(c) Natural language inference (LSTM)

Figure 3.4: End-to-end speedup over full precision model vs model quality on MNIST, language modeling and natural language inference over a grid search of precisions for weights and activations. Standard quantized inference is limited to weights=activations $\in (1, 4, 8, 16, 32)$. *PrecisionBatching* leverages execution with finer granularity inference with weights $\in (1, 2, 4, 8)$, activations $\in (8, 16, 32)$. Dotted lines show Pareto boundary where layers have the same precision, while solid lines indicate layers may have different precision. *PrecisionBatching* attains significant speedups over standard quantized inference methods.



(a) Reacher Hard (easy)

(b) Cheetah Run (medium)

(c) Humanoid Stand (hard)

Figure 3.5: End-to-end speedup over full precision model vs model quality on reinforcement learning tasks (Deepmind Control Suite; difficulty from [106]). All layers have the same precision/activations. Standard quantized inference is limited to weights=activations $\in (1, 4, 8, 16, 32)$. *PrecisionBatching* leverages execution with finer granularity inference with weights $\in (1, 2, 3, 4, 5, 6, 7, 8)$, activations $\in (8, 16, 32)$. *PrecisionBatching* attains significant speedups over standard quantized inference methods.

tolerance. Additionally, the finer granularity of precision supported by *PrecisionBatching* enables greater speedup per accuracy when using variable quantization across layers. The same data is reflected in Table 3.3, which shows the corresponding best achieved speedup for each method for different error margins.

Table 3.3: Best model quality, speedups and precision assignments per error margin for *PrecisionBatching* and baselines over a grid search of precisions for weights and activations. *PrecisionBatching* precision assignments format: (L_i bits, A_i bits); quantized inference precision assignments format: (L_i=A_i bits). Standard quantized inference is limited to weights=activations $\in (1, 4, 8, 16, 32)$. *PrecisionBatching* leverages execution with finer granularity inference with weights $\in (1, 2, 4, 8)$, activations $\in (8, 16, 32)$. For reinforcement learning, all layers have the same precision (though for *PrecisionBatching* activation precision is not necessarily the same as weight precision). *PrecisionBatching* yields $1.5\times - 2\times$ speedup over standard quantized inference.

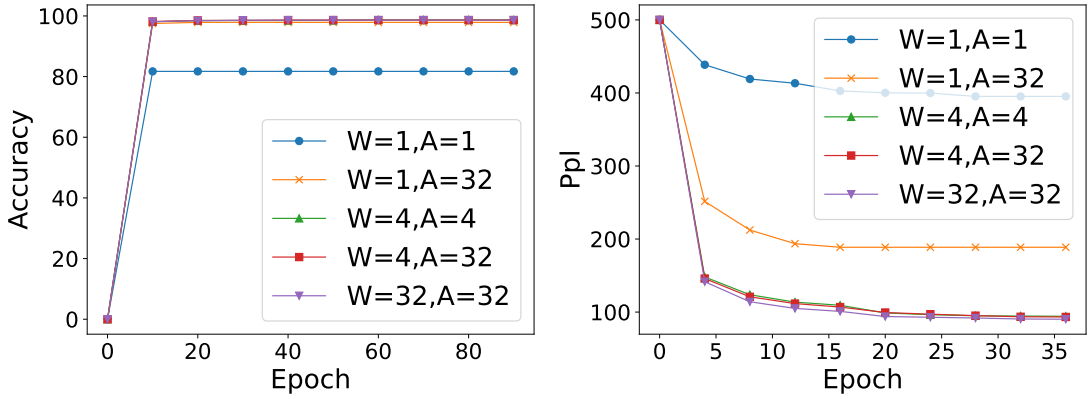
Task	Error	Quality	Speedup vs FP32	Speedup vs Int8	Method	Precision Assign.
MNIST (acc.)	$< 1\%$	97.3%	16.6	2.4	PBatch	(4,8)(1,8)(1,8)
		97.9%	8.0	1.2	Baseline	(8,4,8)
	$< 5\%$	97.3%	16.6	2.4	PBatch	(4,8)(1,8)(1,8)
		94.3%	9.1	1.3	Baseline	(4,4,4)
	$< 15\%$	87.3%	21.0	3.1	PBatch	(1,8)(1,8)(1,8)
		94.3%	9.1	1.3	Baseline	(4,4,4)
Language Modeling (ppl.)	< 5	94.3	7.9	1.5	PBatch	(4,8)(4,8)
		93.7	5.4	1	Baseline	(8,8)
	< 25	109.3	9.8	1.8	PBatch	(1,8)(4,8)
		104.3	5.9	1.1	Baseline	(4)(8)
	< 50	145.3	11.3	2.1	PBatch	(1,8)(2,8)
		148.9	6.0	1.2	Baseline	(4,4)
Natural Language Inference (acc.)	$< 1\%$	77.8	14.3	1.6	PBatch	(4,16)(1,8)
		77.9	10.5	1.2	Baseline	(4,8)
	$< 5\%$	74.0	26.3	3.0	PBatch	(1,8)(1,8)
		77.4	12.9	1.5	Baseline	(4,4)
	$< 15\%$	74.0	26.3	3.0	PBatch	(1,8)(1,8)
		77.4	12.9	1.5	Baseline	(4,4)
Reacher Hard (rew.)	< 50	975.0	7.7	1.69	PBatch	(4,8)(4,8)(4,8)
		976.4	4.5	1	Baseline	(8,8,8)
Cheetah Run (rew.)	< 50	651.6	7.6	1.53	PBatch	(4,8)(4,8)(4,8)
		700.7	5.0	1	Baseline	(8,8,8)
Humanoid Stand (rew.)	< 50	825.3	7.2	1.47	PBatch	(6,8)(6,8)(6,8)
		789.5	4.95	1	Baseline	(8,8,8)

3.5.4 Retraining Benefits of Higher Activation Precision

We show that higher activation benefits the retraining process, leading to better convergence and accuracy. Retraining generally improves the quantized model's quality at lower bitwidths and works by training the model to account for quantization error. As standard quantized inference methods require weights and activations be the same precision, the model must be retrained with same precision weights and activations. However, this often makes retraining slow and quality frequently falls short of the corresponding full precision baseline at low bitwidths. We show that higher activation

Table 3.4: Final scores achieved by quantized models with ≤ 4 bit weights, with and without retraining. Using higher precision activations, as *PrecisionBatching* enables, consistently achieve higher quality, even with retraining.

Task	Base Score	Weight Bits	No Retrain		Retrain (scratch)		Retrain (finetune)	
			W=A	W=32	W=A	W=32	W=A	W=32
MNIST (acc.)	98.5	W=1	10.1	85.8	81.7	97.9	80.0	**98.2**
		W=4	94.3	97.1	**98.6**	**98.6**		
Language Modeling (ppl.)	90.1	W=1	828.1	188.0	396.0	190.7	316.5	**142.4**
		W=4	148.9	94.3	94.0	**93.2**		



(a) MNIST (FC)

(b) Language modeling (LSTM)

Figure 3.6: Benefits of higher precision activations for retraining quantized models. Higher activations (A=32) achieves better convergence and quality after retraining. This is especially true for 1 bit weights. W=32,A=32 is the full precision baseline.

precision, as *PrecisionBatching* enables, facilitates retraining, thus leading to better final quality and faster convergence.

We perform retraining from scratch and from pretrained model on both MNIST and language modeling with quantization aware training, the standard method to retraining for a quantized model[80]. Note we do not perform any modifications to training (e.g: architectural changes to the network to assist better performance). We train MNIST for 100 epochs and the language model

for 40 epochs using the same hyperparameters as described in previous sections. During retraining, quantization aware training is performed immediately from the very beginning (no quantization delay).

Figure 3.6 shows the results of retraining on 1 bit and 4 bit precision weights for MNIST and language modeling. Particularly for 1 bit weights, retraining with 32 bit (full precision) activations enables faster and better convergence. For MNIST, 1 bit weights and activations is stuck at 80% accuracy whereas 1 bit weights, 32 bit activations achieves near full precision performance. For language modeling, 1 bit weights and activations converges at a worse quality (300 ppl), while 1 bit weights and 32 bit activations achieves much better quality (190 ppl). Table 3.4, compares scores achieved with no retraining, retraining from scratch and retraining from pretrained model (finetuning) and demonstrates that higher precision activations consistently achieves better scores across the tasks in all regimes.

3.5.5 Larger Batch Sizes

PrecisionBatching favors low batch sizes (batch 1 is best, a matrix-vector multiplication) to leverage the weight boundedness of the problem. With larger batch sizes the technique sees significantly less gains, and may even incur a slowdown, as larger batch sizes are more compute and activation bound. The significance of this limitation means that *PrecisionBatching* is primarily targeted for the linear components of a network (that have low batch dimension), which limits application of the algorithm primarily to fully connected neural networks, RNNs and LSTMs particularly for inference with low batch sizes where latency is important. For this reason, convolutions, which may be framed as a matrix-matrix multiply will see less speedup using *PrecisionBatching*.

However, despite these shortcomings, we argue that speeding up low batched fully connected layers of a network is a significant contribution as many real world applications deploy such networks in practice. For example, OpenAI Five[24] employs a 4096 layer LSTM and inference during game

play processes frame by frame; Google's on-device keyword prediction model[98] similarly employs an LSTM and inference (not training) is performed sentence by sentence to minimize latency; likewise, Waymo's ChauffeurNet model[19] consists of large LSTM and RNN components which perform inference per environment step. We believe *PrecisionBatching* is a step towards fully utilizing the parallel compute capabilities of traditional hardware systems and will be useful in many high performance machine learning use cases.

3.6 Conclusion

We present *PrecisionBatching*, a quantized inference algorithm for speeding up neural network execution on traditional hardware platforms at low weight bitwidths. *PrecisionBatching* leverages the compute efficiency of traditional hardware platforms (e.g: GPUs) to perform inference with higher activation precisions, enabling execution with lower precision weight layers, achieving a net speedup. Across various models (fully connected, LSTMs, RNNs) and applications (MNIST, language modeling, natural language inference, reinforcement learning) we show that *PrecisionBatching* yields end-to-end speedups of over $8\times$ that of full precision inference ($1.5\times - 2\times$ that of standard 8-bit quantized inference) at the same error tolerance.

4

Tabula: Efficiently Computing Nonlinear Activation Functions for Private Inference

In this chapter we build off the critical insights of the prior chapter towards constructing more efficient private machine learning systems. Specifically, we leverage the key observation that neural networks may be heavily quantized with little accuracy loss to accelerate private nonlinear activation functions in private neural network inference by using lookup tables. Nonlinear activation

functions are the bulk of the system costs in private neural network inference and accelerating them considerably speeds up private inference.

4.1 Introduction

Secure neural network inference seeks to allow a server to perform neural network inference on a client's inputs while minimizing the data leakage between the two parties. Concretely, the server holds a neural network model M while the client holds an input x. The objective of a secure inference protocol is for the client to compute $M(x)$ without revealing any additional information about the client's input x to the server, and without revealing any information about the server's model M to the client. A protocol for secure neural network inference brings significant value to both the server and the clients. The clients' sensitive input data is kept secret from the server, shielding the user from malicious data collection. Additionally, the client does not learn anything about the server's model, which prevents the model from being stolen by competitors.

Current state-of-the-art multiparty computation approaches to secure neural network inference require significant communication between client and server, lead to excessive runtime slowdowns, and incur large storage penalties [171,74,121,192,125,38]. The source of these expenses is computing nonlinear activation functions with garbled circuits [238]. Garbled circuits are costly in terms of computation, communication, and storage. Concretely, executing ReLU activation functions using garbled circuits requires over 2 KB of communication per scalar element of the input [171] and imposes over 17 KB of preprocessing storage per scalar element of the input [171,74]. These costs make state-of-the-art neural network models prohibitively expensive to deploy: on ResNet-32, state-of-the-art multiparty computation approaches for a single secure inference require more than 300 MB of data communication [171], take more than 10 seconds for an individual inference [171], and impose over 5 GB of preprocessing storage per inference [74]. These communication, runtime, and storage costs

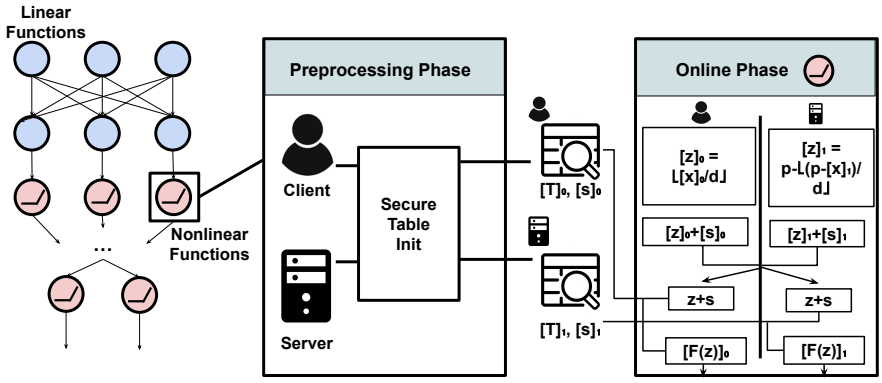


Figure 4.1: The Tabula approach to computing nonlinear functions for secure neural network inference. Tabula precomputes lookup tables $[T]_0$, $[T]_1$ stored on client and server respectively, and also initializes shares of the secret s so that the client holds $[s]_0$ and the server holds $[s]_1$. The lookup tables $[T]_i$ contain the result of all possible nonlinear function calls to an activation function and uses quantization to make storing all possible function calls in the table feasible. These lookup tables map secret shares of the quantized inputs to the nonlinear function to secret shares of the output of the activation function. During the online phase, these lookup tables enable extremely efficient nonlinear activation function execution and proceeds by 1) securely truncating the inputs, 2) reconstructing a blinded index and 3) looking up the blinded index in the lookup tables $[T]_i$. Our code is released at https://github.com/tabulainference/tabula.

pose a significant barrier to deployment, as they degrade user experience, drain clients' batteries, induce high network expenses, and eliminate applications that require sustained real time inference. To replace garbled circuits and other methods [192,112] for privately computing nonlinear functions, we propose Tabula, a two-party secure protocol to efficiently evaluate neural network nonlinear activation functions. During an offline preprocessing phase, Tabula generates tables that contain the encrypted result of evaluating a nonlinear activation function over a range of all possible quantized inputs. New tables are precomputed for each nonlinear function performed during inference, and these tables are split across client and server. Then, at inference time, Tabula performs two steps to evaluate a nonlinear activation function: 1) securely quantize neural network activation inputs down to the precision of the range of the inputs to the precomputed tables and 2) securely lookup the result of the activation function using a two-party secure table lookup procedure [117,128,53]. By heavily quantizing neural network activations and reducing the space of inputs to

the nonlinear activation function, TABULA enables storing all possible results of the activation function in a table without requiring an infeasibly large amount of memory. This allows the application of the subsequent two-party secure table lookup protocol, which is efficient and has low storage, communication, and computation overhead.

TABULA achieves significant improvements over garbled circuits and other [192,112] approaches for securely computing nonlinear functions on important system metrics such as communication and runtime, while maintaining or even improving storage costs.

- **Runtime**

  TABULA offers significant runtime improvements over garbled circuits with quantized inputs due to the simplicity of the online phase of the secure table lookup protocol [128,117,53]. TABULA's runtime for an individual activation function is the cost of transferring a single secretly shared value between parties, and performing a single memory access on the subsequent value. Our results show that when computing individual functions, TABULA is over 100× faster than garbled circuits with quantized inputs. This leads to significant overall runtime improvements when performing secure neural network inference. Our results show that across various standard networks (LeNet, ResNet-32, ResNet-34, VGG) TABULA achieves up to 50× runtime speedup compared to garbled circuits with quantized inputs. Additionally, during the online phase, the TABULA protocol requires just one table lookup; this is significantly less computation compared to schemes that use function secret sharing (FSS), which require computing PRGs like AES-128 [223,89,4,31,29,197].

- **Communication**

  TABULA requires significantly less communication than garbled circuits with quantized inputs and also significantly less communication versus other state-of-the-art approaches like [192]. TABULA's communication cost for a single activation function is the cost of commu-

nicating a single secretly shared element between parties, and is independent of the complexity of the nonlinear function. Our experiments show that, compared to garbled circuits with quantized inputs, communication required for a single nonlinear function call is reduced by a factor of over $280 - 560\times$ leading to an overall communication reduction of up to $9\times$ on various standard neural networks. Additionally, compared to other state-of-the-art protocols for computing nonlinear functions like [192], we show TABULA reduces communication by up to $40\times$ on a per-operation basis, leading to up to $10\times$ reduction in communication when performing end-to-end private inference on various neural networks.

- **Storage and Memory**

  TABULA utilizes comparable storage and memory as garbled circuits with quantized inputs. TABULA's table sizes are dictated by how heavily quantized the activations are and increase exponentially with the precision of the activations. Notably, TABULA's table sizes affect the precision of the activation function and hence affect neural network accuracy. However, as neural network activations may be significantly quantized without significantly affecting neural network quality [180,55,241,154], the sizes of these individual tables may be reduced enough to allow a comparable or smaller amount of storage than garbled circuits with quantized inputs. Generally, across different models, TABULA uses between $.25 - 2\times$ as much memory as garbled circuits while maintaining similar model quality. Like garbled circuits, TABULA requires a new table for each individual nonlinear operation to maintain security.

A comparison of our work against others across some of these axes is shown in Table 4.1.

	Works	Comm. Cost (per-op)	Runtime Cost (per-op)	Preproc. Storage Cost (per-op)	Supported Nonlinear Operations	Mode of Operation	Online Phase Computation
Tabula	Ours	2B (Any function, must fit in lookup table)	.55 us	16KB	Any	Secure Lookup Tables	1 RAM lookup
Garbled Circuits	Delphi[171], Gazelle[125], SecureNN[174], DeepReduce[121], Circa[74]	562B	70 us	17.5KB	Any	GCs [238]	-
Tree-Based Comparator	CryptFlow2[192], Cheetah[112]	96B	-	-	ReLU only	Oblivious Transfer [151]	-
Function Secret Sharing (FSS)	Pika[223], Llama[89], [4] Ariann[197]	2B (8-bit compare, [31])	-	512B [31]	Any	FSS [31] [29]	PRG (i.e: AES-128)

Table 4.1: Comparison of our work against other approaches for securely computing nonlinear activation functions across selected axes. Unless specified costs refer to the cost of the online phase. Compared to garbled circuits, the most widely used state-of-the-art protocol for securely computing nonlinear functions, our approach sees significant improvements in communication and runtime at comparable storage costs. We also compare our approach against less generic protocols for non-linear function computation (tree-based comparator, limited to only ReLU) on the basis of communication where we again see considerable improvements. Finally, compared to function secret sharing (FSS) schemes, our approach is comparable in attaining low communication cost while being computationally more efficient.

4.2 Related Work

4.2.1 Multiparty Computation Approaches to Secure Neural Network Inference

Multiparty computation approaches to secure neural network inference have been limited by the costs of computing both the linear and nonlinear portions of the network [174,196,192,127]. Recent works like Minionn, Gazelle and Delphi [150,125,171,143,192,121,38,74] have optimized the linear operations of secure neural network inference via techniques like preprocessing to the point they are no longer a major system bottleneck [171]. Hence, current state-of-the-art approaches to secure inference

like Minionn, Gazelle, Delphi, and CrypTFlow2 are primarily bottlenecked by nonlinear operations. Specifically, these approaches rely on garbled circuits, or a circuit-based protocol, to compute nonlinear activation functions (e.g: ReLU) [150,125,171,129,51], resulting in notable drawbacks including high communication, runtime and storage costs.

Our approach addresses the problems posed by garbled circuits by eliminating them altogether. Our method is centered around precomputing lookup tables containing the encrypted results of nonlinear activation functions, and using quantization to reduce the size of these tables to make them practical.

4.2.2 Lookup Tables for Secure Computation

Lookup tables have been used to speed up computation for applications in both secure multiparty computation [140,52,128,190,56] and homomorphic encryption [144,49]. These works have demonstrated that lookup tables may be used as an efficient alternative to garbled circuits, provided that the input space is small. Prior works have primarily focused on using lookup tables to speed up traditional applications like computing AES [128,52,140,56] and data aggregation [190]. Notable exceptions include [49] which focuses on linear regression, and [191] which applies variants of a lookup table as part of the protocol to secure machine learning inference. Lookup tables are also widely used as cryptographic primitives in SNARKS [202,15], notably as efficient primitives for non-arithmetic operations inside circuits [15].

To the best of our knowledge, there exists little prior work which applies secure lookup tables to the private execution of large neural networks. Most current state-of-the-art secure inference systems like [171,174,74,121] use garbled circuits. Two exceptions to this include [192,112], which use a tree-based secure comparator. However, the tree-based secure comparator used in [192,112] is significantly limited to only the ReLU activation function, and still requires significant computation and communication overhead. Another work, [191], does indeed use lookup tables as part of their protocol for

evaluating activation functions, but crucially focuses on ensuring numerical precision, leading to lower system performance. We highlight that a key distinction in our use of lookup tables is that we store all possible results of the nonlinear function in these tables, which uses exponential storage. This storage is made manageable by heavily quantizing the neural network. This strategy of securely evaluating a function through lookup tables, although known theoretically [117,53], to the best of our knowledge has not been applied practically until now, due to the exponential storage costs. The secure lookup table approach of "storing everything in a table" is remarkably well suited to securely and efficiently computing neural network nonlinear activation functions for two reasons: 1) neural network activations may be quantized to extremely low precision with little degradation to accuracy and 2) neural network activation functions are single operand. These two factors allow us to limit the size of the lookup table to be sufficiently small to be practical, and consequently we can achieve the significant performance benefits of secure lookup tables at runtime (i.e two orders of magnitude less communication). While work on quantization applied to secure inference exists [51,129], these works do not combine this property with lookup tables for evaluating nonlinear functions. In summary, we emphasize that prior works that use secure lookup tables have either applied them towards non-ML applications (i.e: MPC for AES-128), or have not leveraged exponential-sized secure lookup tables in combination with neural network quantization to make them practical; although the exponential-sized secure lookup table approach for securely computing functions is known, the unique combination of this technique with neural network quantization has not been previously explored. In this work, we demonstrate that this unique combination of techniques can be applied to dramatically reduce the costs of secure neural network inference.

4.2.3 Function Secret Sharing

The secure lookup table approach [117] employed by Tabula is related to function secret sharing (FSS) approaches used in various private neural network inference approaches like [22,3,89,4]. The se-

cure lookup table approach of "storing all function inputs/outputs in a secure lookup table" can be theoretically categorized as a FSS approach. But there are several concrete differences between the secure table lookup approach TABULA uses compared to traditional FSS approaches. These distinctions lead to significant runtime differences. Concretely, our secure lookup table approach incurs exponential storage costs which necessitates aggressive activation quantization to make storage costs practical. However, this approach also enables a highly efficient online phase which requires just one 8-bit memory access (in addition to the 2B communication between parties). FSS approaches, on the other hand, rely on using distributed point functions (DPFs) or distributed comparison functions (DCFs),[31,29] which in turn require evaluating PRGs (i.e: AES-128). Specifically, a table lookup using FSS requires at least $\log(N)$ PRG or AES-128 evaluations, where N is the number of entries in the table, leading to 8-16 AES-128 evaluations per activation function call. This cost increases for more complex nonlinear functions[31,29]. Evaluating PRGs like AES-128 is comparatively more expensive than TABULA which requires just 1 8-bit memory access, as a modern processor even with hardware acceleration computes only around 100M AES-128 operations[1], whereas a modern processor has a memory bandwidth in the 100GB/s range. As such, TABULA is much more computationally efficient compared to FSS schemes, though as a drawback requires aggressive quantization to make practical. Another benefit to TABULA is that its communication cost is always 2B regardless of the nonlinear function being securely computed. This is not the case for function secret sharing where more complicated nonlinear functions may cost more than 2B[31,29]. Furthermore, a third advantage is that TABULA exhibits information theoretic security in the online phase, while function secret sharing schemes are only computationally secure up to a factor of the security parameter[29]; that is, the protocol leaks no information about the underlying data unlike FSS schemes, as FSS schemes rely on pseudorandom functions. A final and notable advantage is that TABULA is much simpler than FSS schemes that rely on computing distributed point functions or distributed comparison functions, which are complex cryptographic primitives. TABULA's main

observation is that neural network activations can be aggressively quantized to 8-bits and below with acceptable accuracy degradation and thus enable the use of exponential-sized lookup tables, thereby avoiding the need for evaluating PRGs like AES-128 during online inference.

4.3 Tabula: Efficient Nonlinear Activation Functions for Secure Neural Network Inference

4.3.1 Background

Secure Inference Objectives, Threat Model

Secure neural network inference seeks to compute a sequence of linear and nonlinear operations parameterized by the server's model over a client's input while revealing as little information to either party beyond the model's final prediction. Formally, given the server's model's weights $W_i \in \mathbb{F}_p^{M_i \times L_i}$ and the client's private input x, the goal of secure neural network inference is to compute $a_i = A(W_i a_{i-1})$ where $a_0 = x$ and A is a nonlinear activation function, typically ReLU. $W_i \in \mathbb{F}_p^{M_i \times L_i}$ are the weights of the neural network represented as a fixed point number in the finite field of modulus p. The dimensions M_i and L_i correspond to the output and input dimensions to the linear layer at i. Convolutions may be cast as a matrix multiply and fit within this framework. State-of-the-art secure neural network inference protocols like Delphi operate under a two-party semi-honest setting [171,143], where only one of the parties is corrupted and the corrupted party follows the protocol. Importantly, these secure inference protocols do not protect the architecture of the neural network being executed, only its weights, and furthermore do not secure any information leaked by the predictions themselves [171]. As we follow Delphi's protocol for the overall secure execution of the neural network these security assumptions are implicitly assumed.

Cryptographic Primitives, Notations, Definitions

Tabula utilizes standard tools in secure multiparty computation. Our protocols operate over ad-

ditively shared secrets in finite fields. We denote $\mathbb{F}_p$ as a finite field over n-bit prime p. We use $[x]$ to denote a two party additive secret sharing of the scalar $x \in \mathbb{F}_p$ such that $x = [x]_0 + [x]_1$, where party i holds additive share $[x]_i$ but knows no information about the other share.

Delphi Secure Inference Protocol

The Delphi framework is a set of protocols consisting of an offline preprocessing phase and an online secure inference phase for securely evaluating neural networks over private client data without revealing to the client the weights of the neural network. The Delphi framework is concerned with the overall evaluation of the neural network (both linear and nonlinear layers). TABULA fits into the Delphi framework by replacing their use of garbled-circuits protocols for secure nonlinear function evaluation, which is the most computationally expensive part of their protocol [171]. To understand how TABULA fits into the Delphi framework [172], we outline how this protocol operates.

- **Per-Input Preprocessing Phase**

  This phase prepares for the secure execution of a single input. The purpose of the preprocessing phase is to initialize the parties with correlated randomness that enables efficient online inference. This phase, as specified in the Delphi paper, requires the use of linearly homomorphic encryption [171] to ensure that the parties do not reveal to each other their secret blinding factors which would compromise the privacy of the entire protocol. For each linear layer $W_i \in \mathbb{F}_p^{M_i \times L_i}$, the client generates a random vector $R_c \in \mathbb{F}_p^{L_i}$ where L_i is the length of the inputs to the current linear layer. The client encrypts R_c with their linearly homomorphic public encryption key k to $Enc_k(R_c)$ and sends this value to the server. The server, upon receiving $Enc_k(R_c)$, generates their own secret vector $R_s \in \mathbb{F}_p^{M_i}$ where M_i is the length of the outputs of the current linear layer. The server then encrypts R_s with the client's public key k to obtain $Enc_k(R_s)$. The server then computes and returns to the client $Enc_k(W_i R_c + R_s)$. The client decrypts this value to obtain $W_i R_c + R_s$ which is then stored in preparation for the online inference phase.

- **Online Inference Phase**

  This phase performs inference on the client's input. For linear layers, the client and server begin with additive secret shares of the linear layer's input x. That is the client and server hold $[x]_0$ and $[x]_1$ respectively, such that $[x]_0 + [x]_1 = x$. As the initial step, the client adds $[x]_0$ with that layer's R_c to obtain $[x]_0 + R_c$. Then the client sends this vector to the server who adds their own share of the layer input $[x]_1$ to obtain $x + R_c$. The server, upon calculating $x + R_c$, then computes $W_i(x + R_c) + R_s = W_i x + W_i R_c + R_s$ (recall that R_s was the secret vector that the server generated for this particular layer). At this point, the client holds $W_i R_c + R_s$ from the preprocessing phase and the server has computed $W_i x + W_i R_c + R_s$. The difference between these two values is $W_i x$. Thus the two parties have obtained a secret sharing of $W_i x$. Then, the two parties must compute a nonlinear activation function over these shares to obtain shares of the activations; in the Delphi framework, garbled circuits are employed to securely perform this operation [171], and it is by replacing this part of the protocol that TABULA obtains considerable computational gains. After calculating shares of the activations, the secure inference phase repeats starting from the linear part of the protocol for the rest of the layers of the network.

As stated, after performing the protocol for the linear phase, the client and server hold secret shares of the input to the nonlinear activation function F. Hence we need to construct a secure protocol for performing nonlinear activation functions. This protocol must operate such that the client and server, each holding a secret share of x, can calculate secret shares of $F(x)$ without leaking any information about x itself. F, the nonlinear function for neural network activations, is typically ReLU, but may also be include other nonlinear functions like sigmoid or tanh. As a note, we emphasize that details on the homomorphic preprocessing phase can be found in the Delphi paper [171].

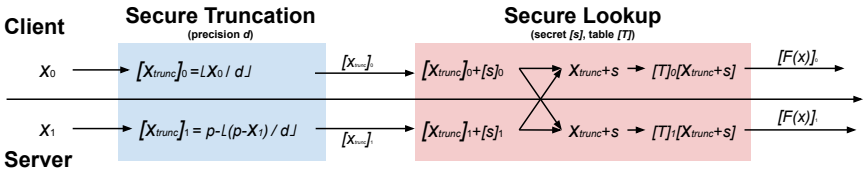


Figure 4.2: Tabula online protocol. Initially, the client and server hold secret shares of the input $[x]$. Both parties begin by executing the secure truncation protocol to obtain shares of $[x_{trunc}]$. Then, the client and server perform the secure table lookup protocol, where they exchange blinded secrets $[x_{trunc}]_i + s_i$ to compute $x_{trunc} + s$. Finally, they use this value as an index into local lookup tables to compute $T_i[x_{trunc} + s]$ which are secret shares of the result of the nonlinear function evaluation.

4.3.2 Tabula for Securely and Efficiently Evaluating Neural Network Nonlinear Activation Functions

Tabula is divided into a preprocessing phase that initializes a lookup table for each individual nonlinear function call used in the neural network, and an online phase which securely quantizes the activation inputs and looks up the result of the function in the previously initialized tables. An overall figure for our protocol is shown in Figure 4.1. We emphasize that our paper primarily focuses on the online phase of execution, which determines the system's real time response speed after knowing a client's input, rather than the preprocessing phase, which may be done offline without knowing the client's input data. We also develop a secure and reasonably efficient algorithm for the preprocessing phase of Tabula and conduct thorough experiments in the results section to demonstrate its viability and effectiveness. We leave further innovations to the preprocessing algorithm to future research.

Below we describe the core building blocks that Tabula utilizes, namely, the secure lookup table procedure [117,128] and secure truncation protocol [174]. We then describe Tabula's online and preprocessing execution phase, and detail its security, communication, and storage properties.

Secure Lookup Table Procedure

We employ the concepts of [117] to enable the computation of a nonlinear function call through a table lookup. By using an exponential amount of preprocessing storage, we obtain a secure protocol under the semi-honest threat model where communication complexity depends only on the size of its input operands, regardless of the complexity of the function being computed. Concretely, we precompute all possible results of a nonlinear function and store them in a table, and utilize these secure tables during the online phase of secure inference. Computing nonlinear functions in the clear is extremely efficient computationally (i.e: cleartext comparison operations), and so the bulk of the pre-computation workload is spent on MPC operations (i.e: Beaver triple multiplication). The lookup table approach is similar to that described in [128,117]. Like in garbled circuits, TABULA requires new circuits per operation to maintain security.

Given a table $T[x] = F(x)$, where $F : \mathbb{F}_p \rightarrow \mathbb{F}_p$ is the target nonlinear function operating over scalars, we initialize a shared table $[T]$ across the parties, so the client holds $[T]_0 \in \mathbb{F}_p^p$ and server holds $[T]_1 \in \mathbb{F}_p^p$. A secret scalar $s \in \mathbb{F}^p$ unknown to both parties is generated and shared between the client and the server, with the client and server holding $[s]_0$ and $[s]_1$ respectively. The shared table T is constructed such that $[T][s + x] = [F(x)]$ for all values of $x \in \mathbb{F}_p$ for some modulus p that determines the precision to compute the nonlinear function. Concretely, this means two tables are generated, $[T]_0$ and $[T]_1$ such that $[T]_0[s+x] + [T]_1[s+x] = F(x)$; both client and server coordinate to initialize their local $[T]_i$ in an offline preprocessing phase. The online phase, given such a shared table, is then straightforward. Initially, the client holds x_0 and the server holds x_1. The client sends to the server $x_0 + s_0$ while the server sends to the client $x_1 + s_1$. This allows both parties to obtain the true value of $x + s$. Both parties then look up this value in their corresponding tables: client looks up $[T]_0[x + s]$, server looks up $[T]_1[x + s]$, the sum of which is $F(x)$. Security is maintained in the online phase as a new table $[T]$ and secret s are used per function call, with s being unknown to either party, perfectly blinding the secret value x.

Securely initializing shared table $[T]$ from T in the offline preprocessing phase is based on the fact

that given an index into a table, a table lookup can be cast as a dot product between the entire table with an indicator vector containing a one in the position of the table index (e.g: the one hot vector encoding of the index). Subsequently, a secure two party demux procedure [128] transforms secret shares of $[s]$ into secret shared vectors $[s']$ which sum to an indicator vector with a one at the s'th position. Finally, a dot product for each entry of the table can be performed to compute $[T]$: $T[x] \times [s'_0] + T[x+1] \times [s'_1] + ... + T[x+n] \times [s'_n] = [T[s+x]]$. We develop an efficient and secure protocol for initializing Tabula tables based on these concepts later in the text.

Secure Truncation

As the size of $[T]$ increases linearly with the size of the field $\mathbb{F}_p$ it becomes necessary to truncate or quantize $[x]$ to prevent $[T]$ from being impracticably large. Linear layers are required to use larger finite fields to ensure that their dot products are computed correctly without overflow. Thus, the input to Tabula is a value secret shared in a larger field, and a secure truncation method is required to switch to a smaller field so that the encoded value may be used to index into a feasibly sized table. We use the secure truncation method in [174] to achieve this. Given a truncation factor d which specifies the precision of the activation inputs, the client and server perform the secure truncation protocol: the client computes $\lfloor [x]_0/d \rfloor$ and the server computes $p - \lfloor (p - [x]_1)/d \rfloor$. After the truncation protocol is performed, the resulting expressions the client and server hold sum to either $[\lfloor [x]/d \rfloor + 1]$ or $[\lfloor [x]/d \rfloor]$ with probability proportional to $1 - \frac{k}{p}$ where k is the maximum value x may take, and p is the maximum value of the finite field of the previous linear layer [174]. These small off by one errors, like quantization error, have little impact on model quality due to neural networks' resilience to noise. However, with probability proportional to $\frac{k}{p}$, a large error occurs that is pessimistically assumed to ruin correctness. To reduce the probability of these catastrophic errors, it is necessary to use a large finite field modulus for linear layers. In practice, we use a 64-bit finite field modulus to reduce the chance that a secure truncation operation catastrophically fails to less than $\frac{1000}{2^{64}}$. Hence, by configuring the modulus appropriately, with high probability, the secure truncation protocol

computes the correctly truncated value with a small off by one error which may be tolerated by neural networks [180,193]. Requiring 64 bits for the field increases the communication cost required by the linear portions of the protocol over other approaches that commonly use 32 bits, however, the reduction in communication cost by using Tabula tables more than makes up for this communication penalty, and this is shown in the results (note without Tabula we use 32-bit precision for the linear layers). In our experiments, using a 64-bit field size was essential to maintaining accuracy; using a 32-bit field size Tabula saw considerably worse accuracy due to catastrophic failures from the secure truncation protocol, as a single catastrophic failure anywhere in the computation propagates throughout the network and ruins the entire inference. We refer to [174] for more details. Developing more effective secure truncation techniques is an important topic for further research.

Tabula Online Phase

Given these fundamental building blocks, we describe the Tabula protocol. In the preprocessing phase, Tabula generates multiple shared tables $[T]$ as described above for each nonlinear function call that is performed when executing the neural network. How much to truncate/quantize the network' activations is chosen offline to maximize network accuracy. In the online execution phase, Tabula quantizes the inputs to the activation function and uses this input to securely lookup up the result of the function. The full protocol is shown in Figure 4.2. The security of Tabula is ensured by the security of the secure truncation protocol [174] and the secure table lookup protocol [128,53,117].

Tabula Online Phase Communication and Storage Cost

Tabula achieves significant communication benefits during the online phase at comparable storage costs. As shown in Figure 4.2, Tabula requires just one round of communication to compute any arbitrary function, unlike garbled circuits, which may require multiple rounds for more complex functions. As an example, ReLU implemented using garbled circuits takes two rounds, whereas Tabula requires just one. Additionally, communication complexity is independent of the com-

plexity of the nonlinear function being computed. Specifically, revealing s_x requires both parties to send their local shares, each nonlinear activation call incurs communication cost corresponding to the number of bits in F_p. Since we use 64-bit F_p, this results in 16 bytes of communication per activation function, the cost of transferring 8-byte field values back and forth. However, we can apply an optimization to reduce this down to twice the cost of transferring the *size of the input to the table*, rather than the field size. If the size of the table is 2^b entries, and if finite field size p is also power of two, then we can have party i first mod their secret shares by 2^b before exchanging them. Hence, the two parties hold $[x_{trunc}]_i \ mod \ 2^b$ before adding their secret shares of the table secret $[s]_i$ to the value and exchanging it; this brings down the total cost of the protocol to $2 \times b$ bits. Modding by 2^b yields the correct answer as $x_{trunc} \ mod \ p \ = \ [x_{trunc}]_0 + [x_{trunc}]_1 + pl$ for some l, and then $(x_{trunc} \ mod \ p) \ mod \ 2^b \ = \ ([x_{trunc}]_0 + [x_{trunc}]_1 + pl) \ mod \ 2^b = ([x_{trunc}]_0 + [x_{trunc}]_1) \ mod \ 2^b = ([x_{trunc}]_0 \ mod \ 2^b) + ([x_{trunc}]_1 \ mod \ 2^b)$. This equivalence shows that the two parties can first perform a modulus of their shares with 2^b, and that their shares would still sum up to the original sum with the correct modulus. With this optimization, communication is now $2 \times b$ bits per nonlinear function call; if we use 8-bit activations, then $b = 8$, and we use 2 bytes of communication total per call during the online phase, an $8\times$ improvement over the 16 bytes as previously stated.

Storage and memory, as mentioned previously, grow exponentially with the precision that is used for activations and linearly with the number of activations in the neural network. Storage costs are thus $n \times 2^k \times N_a$ bits where n is the number of bits to use for the the output of the activation function, k is the number of bits to the input of the activation function, and N_a is the total number of activations that are performed by the neural network. The majority of the storage cost comes from the 2^k factor, the size of the individual tables, which grows exponentially with input space / precision of the activations. However, as neural network activations may be heavily quantized down without significantly affecting model quality [180,55,241], we can reduce this factor enough to be practical; we also highlight more advanced techniques like using a variable number of bits per layer of

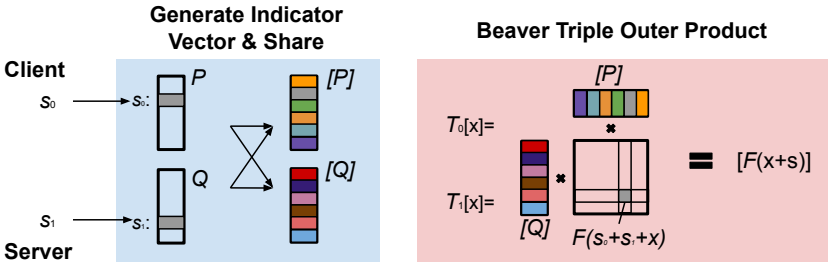


Figure 4.3: Tabula preprocessing protocol. Client and server generate secrets s_0, s_1 and encode them in an indicator vector (i.e: construct a vector of length equal to the field size, then setting a one to the position of the party's secret index). The parties then secret share this indicator vector with the other party. To obtain the entry for $T_i[x]$, the parties compute an outer product between the shared indicator vectors and a 2-dimensional table containing $F(m + n + x)$ (where m, n span the two dimensions of the table), which obtains $[F(x + s)]$ where $s = s_0 + s_1$. This works as the 2-D coordinates formed by where the indicator vectors are set privately select m, n through the dot-product; since this is done via private MPC operations, no information is leaked to either party about their corresponding secrets.

the network can be employed for better performance[60]. We verify that quantization has negligible impact on model quality in our results. We highlight that every bit of precision that is trimmed from the activation yields a factor of two reduction in storage and memory costs, and hence more advanced quantization techniques[180,55,241] to reduce precision yields significant benefits. As storage and memory varies with the precision of activations that is used, there is a natural tradeoff between the accuracy of the model and the achieved memory/storage requirement. We examine these tradeoffs in the results.

4.3.3 Tabula Preprocessing Phase

Similar to garbled cicuits, Tabula tables require a preprocessing phase that initializes the client and server with a single-use table that is used once per activation function call during the inference phase. We develop a secure and efficient protocol for initializing Tabula tables, detailed below.

Preprocessing Phase Problem Statement

Given a nonlinear function $F : \mathbb{F}_p \rightarrow \mathbb{F}_p$, we wish to securely initialize the client and server with

tables that map any possible input in $\mathbb{F}_p$ to secret shares of the result of the nonlinear function F. Specifically, we wish to initialize on the client a table $[T]_0 \in \mathbb{F}_p^p$, and on the server a table $[T]_1 \in \mathbb{F}_p^p$, such that $[T]_0[s+x] + [T]_1[s+x] = F(x)$, where $s \in \mathbb{F}_p$ is a secret unknown to both client and server. Additionally, at the end of the protocol, we want the client to hold $s_1 \in \mathbb{F}_p$ and server to hold $s_2 \in \mathbb{F}_p$ such that $s_1 + s_2 = s$.

Tabula Secure Preprocessing Protocol

To achieve this preprocessing step securely, the client and server first randomly generate s_0 and s_1 respectively, and s is implicitly defined as $s_0 + s_1$ (though, as the parties do not know each others' secrets, they hence do not know what s is). Then, the client generates a random indicator vector $P \in \mathbb{F}_p^p$ such that $P[x] = \begin{cases} 1 & x = s_0 \\ 0 & x \neq s_0 \end{cases}$; the server similarly initializes $Q \in \mathbb{F}_p^p$ with s_1. Client and server exchange shares of P and Q respectively, hence, both parties hold secret shares $[P]$ and $[Q]$ while leaking no information about s_0 or s_1 to either party. Finally, client and server jointly initialize their table $T_i[x] = \sum_{m=0}^{p} \sum_{n=0}^{p} F(m+n+x)([P]_m \times [Q]_n)$, where $i = 0$ for client and $i = 1$ for server, and secret shares $[P]_m, [Q]_n \in \mathbb{F}_p$ are multiplied using Beaver triple multiplication [20]. We depict the full preprocessing phase operation in Figure 4.3 and present the algorithmic details below.

Algorithm 3: Tabula Preprocessing Phase

1: client $\longleftarrow$ random secret $s_0 \in \mathbb{F}_p$

2: server $\longleftarrow$ random secret $s_1 \in \mathbb{F}_p$

3: client, server locally initialize table $F' \in \mathbb{F}_p^{p \times p \times p}$ s.t. $F'[i][j][x] = F(i+j+x)$

4: client computes $P \in \mathbb{F}_p^p$ s.t $P[i] = \begin{cases} 1 & i = s_0 \\ 0 & x \neq s_0 \end{cases}$

5: server computes $Q \in \mathbb{F}_p^p$ s.t $Q[i] = \begin{cases} 1 & i = s_1 \\ 0 & x \neq s_1 \end{cases}$

6: client, server exchange shares of P, Q to obtain secret shares $[P], [Q]$

7: client, server compute $[PQ] \in \mathbb{F}_p^{p \times p}$ where $[PQ][i][j] = [P][i] \times [Q][j]$ via Beaver triple multiplication

8: client, server compute $[T][x] = \sum_{m=0}^{p} \sum_{n=0}^{p} F'[m][n][x] \times [PQ][m][n]$ for all $i \in \mathbb{F}_p$

Preprocessing Phase Correctness

In this protocol the client and server specify the coordinate of s through an outer product of their indicator vectors P, Q, which sets $[T][x] = [F(s+x)]$. This computes the correct answer as

$$[T][x] = \sum_{m=0}^{p} \sum_{n=0}^{p} F(m+n+x)([P]_m \times [Q]_n)$$

$$= \sum_{m=0}^{p} \sum_{n=0}^{p} F(m+n+x)([P_m \times Q_n]) \text{ (Beaver triple multiplication)}$$

$$= \sum_{m=0}^{p} \sum_{n=0}^{p} F(m+n+x) \times \begin{cases} [1] & m = s_0 \text{ and } n = s_1 \\ [0] & \textit{otherwise} \end{cases}$$

$$= [F(s_0 + s_1 + x)]$$

$$= [F(s + x)]$$

Preprocessing Phase Security

Security and privacy is preserved as each step of the protocol consists entirely of secure steps: secret sharing P and Q leaks no information about the vectors (hence leaks no information about either s_0, s_1, and s), and Beaver triple multiplication is likewise secure[20]. Concretely, we see that the only communication between client and server occurs when they exchange blinded secrets (i.e: $[P]$, $[Q]$) and when they perform Beaver triple multiplication. As these steps leak no information to either party about the underlying secrets, the client and server compute $[T]$ without leaking any information about s_0, s_1 and hence leak no information about s.

Preprocessing Phase Communication and Computation Cost

The bulk of the preprocessing phase lies in computing an outer product between P, Q. We perform this outer product just once and reuse it across i, x in $T_i[x]$. Hence, the protocol requires performing just a single outer product between vectors $\in \mathbb{F}_p^p$. This incurs $O(p^2)$ Beaver triple multiplication operations, and assuming that a sufficient number of Beaver triples were generated before the preprocessing phase, communication cost is naively $O(p^2 \log(p))$ bits assuming that the values of the vectors are each $\log(p)$ bits. This naive $O(p^2 \log(p))$ communication cost can be significantly reduced to $O(p^2)$ by having P, Q be secret shared *binary* vectors, rather than be shared in $\mathbb{F}_p$, then doing a conversion back to $\mathbb{F}_p$ after the final inner product. This can be done as the true values of the vectors P, Q are either 0 or 1. Concretely, upon reception of the binary shares of P or Q the current party computes $[T_i(x)] = \sum_{m=0}^{p} \sum_{n=0}^{p} F(m + n + x) * [PQ^T][m, n]$, and observe that $[T_i(x)]_1 - [T_i(x)]_2$ is either $F(s + x)$ or $-F(s + x)$ (in the case that the first party has the 1 and the second party has the 0 in the selected index, and the reverse). We perform an extra Beaver triple multiplication by the correction factor to eliminate this potential negation (by multiplying it by the par-

ity of the sum of $[PQ^T]$), which costs an extra $O(\log(p))$ bits of communication per inner-product. Since there are only p inner products, these correction factors cost a negligible $O(p\log(p))$ communication. With this optimization, preprocessing communication cost is now $O(p^2)$ bits. As p, the quantized field size of the activation domain, is set to be extremely small (i.e.: less than or equal to 256 for 8-bit quantized activations), preprocessing communication costs $2(256)^2 = 2^{16} = 131072$ bits = 16 KB per table (the factor of two at the front is because Beaver triple multiplication requires both parties exchange secret shared values, and we have 256^2 Beaver triple multiplication operations). This is comparable to the 17.5 KB cost that garbled circuits with full precision requires [171].

We emphasize that the prior analysis assumed that Beaver triples were obtained beforehand in a pre-preprocessing phase; we think this is reasonable that in a practical scenario parties would obtain sufficient amounts of Beaver triples for any protocol due to their importance. However, accounting for Beaver triple preprocessing, communication cost is still an asymptotic $O(p^2)$ bits assuming that Beaver triples were generated using oblivious transfer e.g: [181], which requires just 2 OT calls to generate 1-bit Beaver triples. Using the OT procedure proposed in [112] the concrete cost of a single OT is 3 bits for 1-bit values. Hence, pre-preprocessing costs for the Beaver triples would still be $O(p^2)$ bits, with a higher constant factor burden. On a concrete example of 8-bit activations, the cost for preprocessing the Beaver triples would amount to 6×256^2 bits = 48 KB of communication. While this exceeds the 17.5 KB cost of garbled circuits, we believe that the online benefits of Tabula more than make up for this detriment.

In terms of computation, we see that for precomputing a single table, we require summing across p^2 values (each entry of the outer product) for every entry of the table. Since there are p entries in the table, computation costs scale as $O(p^3)$. However, these operations may be efficiently vectorized and parallelized as they are standard matrix operations. For 8-bit tables, this is around 16 million field operations.

4.3.4 Note on Tabula Security

Although the Tabula protocol assumes a semi-honest threat model as inherited from the Delphi [171] framework, more generally Tabula's online phase which consists of utilizing a secure lookup procedure is information-theoretically secure [117]. This is intuitive as all communication between parties are randomly blinded by an additive factor. This is another advantage that Tabula holds over garbled circuits implementations many of which are only computationally secure up to a security parameter in the semi-honest setting [2].

4.4 Results

We present results showing the benefits of Tabula over garbled circuits for secure neural network inference. We evaluate our method on neural networks including a large variant of LeNet for MNIST, ResNet-32 for Cifar10, and ResNet-34 / VGG-16 for Cifar-100, which are relatively large image recognition neural networks that prior secure inference works benchmark [74,171,121,222]. Unless otherwise stated, we compare against an implementation of the Delphi protocol [171] using garbled circuits for nonlinear activation functions, without neural architecture changes, during the online inference phase. As before, we use 64-bit fields for Tabula to reduce the impact of the secure truncation protocol, however, for the baseline implementation that uses garbled circuits we use 32-bit fields to reduce communication cost. Experiments are run on AWS c5.4x large machines (US-West1 (N. California) and US-West2 (Oregon)) which have 8 physical Intel Xeon Platinum @ 3 GHz CPUs and 32 GiB RAM; network bandwidth between these two machines achieves a maximum of 5-10 Gbit/sec, according to AWS. We use the same machine/region specs as detailed in [171], but with 2x more cores/memory (c5.4xlarge vs c5.2xlarge).

To ensure fair comparison, we compare Tabula against garbled circuits with quantized inputs, specifically garbled circuits with 32-bit, 16-bit, and 8-bit inputs, which are commonly used preci-

sions, but we also show more detailed results on a more granular level by fixing accuracy/precision and comparing systems costs between our methods. With the same activation precision, both Tabula and garbled circuits compute the same result. Like other works we benchmark using a batch size of 1 [171,192,112,74]. Tabula is feasible only for precision up to 12-bits due to the exponential storage costs required for each extra bit of precision, hence, we show results up until this limit. Garbled circuits on the other hand can obtain 32/16-bit precision, however this is beyond the range of precision that Tabula may handle to avoid large storage costs. Hence, we may show results for 16/32-bit garbled circuits as a baseline reference, but the main comparison is between 8-bit garbled circuits and 8-bit Tabula, or between the two approaches when obtaining a fixed accuracy.

4.4.1 Communication Reduction

ReLU Communication Reduction

We benchmark the amount of communication required to perform a single ReLU with Tabula vs garbled circuits. Table 4.2 shows the amount of communication required by both protocols during online inference. Tabula achieves significant ($> 280\times$) communication reduction compared to garbled circuits. Note our implementation of garbled circuits on 32-bit inputs achieves the same communication cost as reported by [171] (2KB communication for 32-bit integers).

Garbled Circuits (32-bit)	Garbled Circuits (16-bit)	Garbled Circuits (8-bit)	Tabula	Comm. Reduction (vs 32/16/8 bit GC)		
2.17KB	1.1KB	.562KB	2B	1112×	560×	280×

Table 4.2: Tabula with 8-bit activations vs garbled circuits communication cost for one ReLU.

We additionally compare ReLU communication of our protocol against recent works like CrypTflow2 [192] and Cheetah [112]. CrypTflow2 and Cheetah similarly utilize a tree-based secure comparison

protocol dependent on oblivious transfer [192,112]. However unlike CrypTflow2, Cheetah swaps out the underlying oblivious transfer implementation for a more efficient version [112]. Our following analysis assumes that CrypTFlow2 uses a more efficient OT protocol based on preprocessing which reduces the online communication costs beyond what they present in their paper; broadly, the tree-based comparison method that CrypTflow2 utilizes requires at least 6 calls to 1-out-of-128 oblivious transfer for optimal communication complexity [192], which, with preprocessing, takes at least $6 \times 128 = 768$ bits or 96 bytes, as oblivious transfer with preprocessing requires sending all n bits to the original sender at the end [22,178]. Tabula requires just 16 bits of communication regardless of the nonlinear function being computed, obtaining a 48× improvement in communication over the tree based comparison method of CrypTflow2 / Cheetah assuming the use of this preprocessing-based OT method. Cheetah's approach on the other hand uses the same tree-based comparison approach [112] but swaps out the underlying OT method for a more efficient version; specifically, Cheetah's communication cost is $11 \times L$ where L is the bitlength of the field element, which results in 88 bits of communication for 8-bit values and 352 bits for 32-bit values. Tabula requires just 16 bits of communication, which represents a 5.5× and 22× reduction respectively. SiRNN [191], another paper which utilizes an OT based protocol, uses more communication than that of ReLU of CrypTflow [192], hence our method would see $> 48\times$ communication improvement when compared to their approach. We also compare communication cost against FSS approaches [31,89,4,197]. Generally, for the ReLU op Tabula obtains the same 2B communication cost as FSS, however Tabula obtains several notable qualitative advantages over FSS, and a table comparison is shown in Table 4.3. We summarize these communication cost comparisons in Table 4.4, which compares the online communication cost of a single ReLU operation for Tabula, CryptFlow2 and Cheetah.

Total Online Communication Reduction

We benchmark the total amount of online communication required during the online phase of a single private inference for various network architectures including LeNet, Resnet-32, ResNet-34

	Tabula	FSS
Computational Efficiency	8-bit memory access per op	>= 1 PRG (i.e: AES-128) operation per op
Generality	2B comm. cost for any nonlinear function (must fit in table)	Comm. cost increases for more complex nonlinear ops
Security	Information-Theoretically Secure	Computational Security (up to security parameters λ)
Complexity	Table lookup	DPF / DCF[29]

Table 4.3: Qualitative comparison between Tabula and FSS schemes.

Tabula	CrypTFlow2 (W/ OT)	Cheetah 32-bit	Cheetah 8-bit	FSS (8-bit; ReLU op)	Comm. Reduction
2B	96B	44B	11B	2B	48 × / 22 × / 5.5 × / 1 ×

Table 4.4: Tabula (8-bit activations) vs CrypTFlow2[192] and Cheetah[112] and FSS[31,29] online communication cost for performing a single ReLU operation during the online phase. For CryptFlow2 the communication is based on an OT method that uses preprocessing which achieves better online communication cost than what is described in[192]. Note FSS, Cheetah, CrypTFlow2 costs are specific to the ReLU op, while Tabula communication cost is the same for any function provided they are quantized down to a sufficiently small table size.

and VGG (batch size 1). Table 4.5 shows the number of ReLUs per network, as well as the communication costs of using garbled circuits (for 32/16/8 bit inputs) vs Tabula. Tabula reduces communication significantly ($> 20\times, > 10\times, > 5\times$ vs 32,16,8 bit garbled circuits) across various network architectures.

We additionally compare end-to-end communication costs against[192], the current state-of-the-art for neural network inference, on various networks Minionn and ResNet34[150], shown in Table 4.6. Tabula's compact tables enable much lower communication costs during the online phase of secure neural network inference, leading to an order of magnitude reduction in communication costs. Finally, Figure 4.4 shows the communication reduction Tabula achieves compared to garbled

Network	ReLUs	Garbled Circuits (32-bit)	Garbled Circuits (16-bit)	Garbled Circuits (8-bit)	Tabula	Comm. Reduction (vs 32/16/8 bit GC)		
LeNet	58K	124 MB	62 MB	31 MB	3.5 MB	35.4×	17.7×	8.8×
ResNet-32	303K	311 MB	155 MB	77 MB	14 MB	22.2×	11.1×	5.6×
VGG-16	276K	286 MB	143 MB	72 MB	12.1 MB	23.6×	11.8×	5.6×
ResNet-34	1.47M	1.5 GB	.75 GB	370 MB	59.5 MB	24.7×	12.4×	6.2×

Table 4.5: Tabula vs garbled circuits total online communication cost during secure inference for different network architectures.

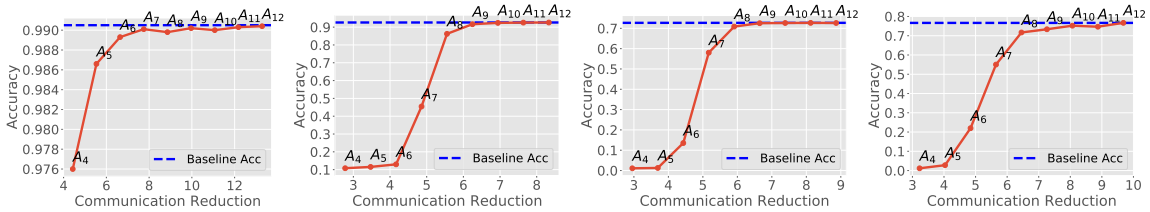
Network	ReLUs	Tabula	CrypTFlow2	Comm. Reduction
MinioNN	176K	25MB	280MB	11.2 ×
ResNet-34	1.47M	59.5 MB	590MB	9.9 ×

Table 4.6: Tabula vs CrypTFlow2 [192] end-to-end communication cost for performing secure neural network inference on selected networks (Minionn [150] CIFAR10 architecture, and ResNet34 CIFAR100).

circuits with A_n-bit quantized inputs at a fixed accuracy threshold, and shows Tabula achieves over $8 - 9\times$ communication reduction across networks to maintain close to full precision accuracy. These values reflect total online communication costs, not just ReLU communication costs, and hence we find we are primarily bottlenecked by the communication for the linear layers rather than nonlinear layers. Also, we do not make any architectural changes to the neural network (e.g: replace any ReLU operations with quadratic operations, retrain, etc).

4.4.2 Storage Costs

We compare the storage savings Tabula achieves against garbled circuits. Recall that Tabula requires storing a single lookup table for each nonlinear activation call. This storage cost grows exponentially with the size of its tables, which dictates the precision of the activations. Using less storage means reducing the precision for the activations of the neural network and introduces some amount



(a) MNIST LeNet **(b)** CIFAR10 ResNet32 **(c)** CIFAR100 VGG16 **(d)** CIFAR100 ResNet34

Figure 4.4: Tabula communication reduction improvement over garbled circuits (x-axis) vs accuracy, when both garbled circuits and Tabula are quantized to n bit precision activations (corresponding to the labels denoted A_n, both of which attain the same accuracy since they compute the same result). Tabula incurs a fixed 16 byte communication cost regardless of the precision of activation inputs; hence when using a higher activation precision, Tabula obtains greater communication reduction over GCs whose communication costs scale with precision; Tabula achieves up to $8 - 9\times$ communication reduction over garbled circuits across tasks when requiring within 1-2% of baseline accuracy. Baseline precision shown as the dashed blue line.

of error into the nonlinear function. This creates a tradeoff between storage and network accuracy. Similar to garbled circuits, Tabula tables must be stored on both client and server, and likewise, storage costs are equivalent for both client and server; hence, in our results we show the storage cost for a single party. Below we show both the storage savings for a single ReLU operation disregarding the accuracy impact from the quantization, and additionally the storage vs accuracy tradeoffs for various networks (LeNet, ResNet32/34, VGG). Storage costs directly translate to memory usage costs at inference time since the lookup tables or garbled circuits must be loaded into memory to be used to evaluate the nonlinear functions.

ReLU Storage Savings vs Precision

We compare the storage use between Tabula and garbled circuits for a single ReLU operation. Tabula's storage use is the the size of its table multiplied by the number of bits of elements in the original field, which we default to 64-bit numbers. Garbled circuits, on the other hand, uses 17KB, 8.5KB, and 4.25KB for each 32-bit, 16-bit, and 8-bit ReLU operation respectively [171].

Figure 4.5 presents the storage usage of both Tabula and garbled circuits for a single ReLU operation, and shows that Tabula achieves comparable storage use to garbled circuits at precisions 8-10,

and lower storage use with precisions below 8. Specifically, with 8 bits of precision for activation Tabula achieves an 8.25×, 4.1× and 2× savings vs 32-bit, 16-bit and 8-bit garbled circuits; with ultra low precision Tabula achieves even more gains (4 bits yields around 136× storage reduction vs 32-bit garbled circuits and 17× reduction vs 8-bit garbled circuits). These results imply that standard techniques to quantize activations down below 8 bits and advanced techniques to quantize below 4 bits [180,55,241] can be applied with Tabula to achieve significant storage savings. Notably, Tabula achieves storage savings at ultra low precision activations as a 1-bit reduction in activation precision yields a 2× storage reduction, unlike for garbled circuits where storage is reduced linearly.

Storage Savings and Accuracy Tradeoff

We present Tabula's total storage usage versus accuracy tradeoff in Figure 4.6. In this experiment, we directly quantize the network's activations during execution time uniformly across layers, recording the achieved accuracy and memory/storage requirements for a single inference. As shown in Figure 4.6, across various tasks and network architectures, activations may be quantized to 9 bits or below while maintaining within 1-3% accuracy. This allows Tabula to achieve comparable or even less storage use than garbled circuits at a fixed accuracy threshold. We emphasize that future work may apply more advanced quantization techniques [180,55] to reduce activation precision below 8-bits and achieve even better storage savings. Our results here show that even with very basic quantization techniques, Tabula achieves comparable storage usage versus garbled circuits, and indicate that Tabula is more storage efficient as fewer bits of precision for the activations are used.

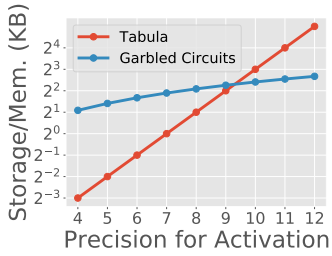
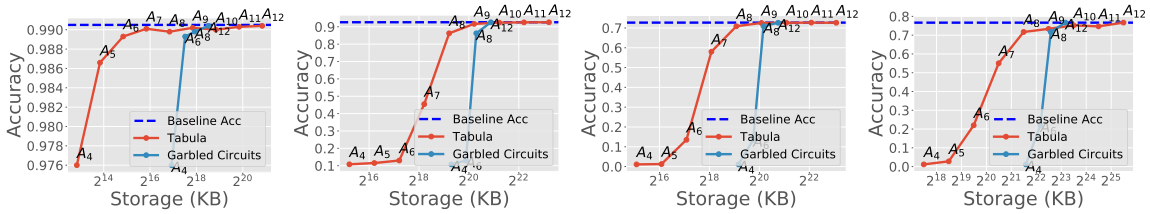


Figure 4.5: Tabula and garbled circuits storage use for a single ReLU operation.



(a) MNIST LeNet **(b)** CIFAR10 ResNet32 **(c)** CIFAR100 VGG16 **(d)** CIFAR100 ResNet34

Figure 4.6: Tabula overall storage usage for a single inference versus accuracy for different tasks and neural networks. Each point is annotated with A_n, specifying the precision of activations for that run. With activation precisions above 10 Tabula uses more storage than garbled circuits due to the exponential increase in the size of its tables; however, below a precision of 8, Tabula achieves notable storage savings ($> \quad 2\times$) over garbled circuits. Baseline precision shown as the dashed blue line.

4.4.3 Runtime Speedup

We compare the runtime speedup Tabula achieves over garbled circuits. As noted in various secure neural network inference works [171,74,38], executing nonlinear activation functions via garbled circuits takes up the majority of secure neural network execution time, hence, replacing garbled circuits with an efficient alternative has a major impact on runtime. Below we present the Tabula's runtime benefits when executing individual ReLU operations and when executing relatively large state-of-the-art neural networks.

ReLU Runtime Speedup

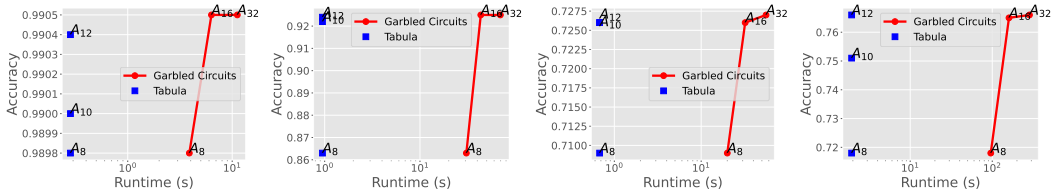
Table 4.7 shows the runtime speedup Tabula achieves over garbled circuits when executing a single ReLU operation. Tabula achieves over $100\times$ runtime speedup due to its simplicity: the cost of transferring 16 bytes of data and a single access to RAM is orders of magnitude faster than garbled circuits. Our implementation of garbled circuits on 32-bit inputs is slower than reported in Delphi [171]. Our implementation of garbled circuits takes around 184 us per ReLU, whereas the reported is 84 us [171]. However, even if the implementation in Delphi achieves an optimal $4\times$ speedup with 8-bit quantization, Tabula is still $38\times$ faster.

Garbled Circuits 32-bit Runtime (us)	Garbled Circuits 16-bit Runtime (us)	Garbled Circuits 8-bit Runtime (us)	Tabula Runtime (us)	Tabula Speedup (vs 32/16/8 bit GC)		
184	111	69	.55	334 ×	202 ×	105 ×

Table 4.7: Tabula runtime speedup vs garbled circuits on a single ReLU operation. Tabula is orders of magnitude faster than garbled circuits.

Neural Network Runtime Speedup

We present Tabula's overall speedup gains over garbled circuits across various neural networks including LeNet, ResNet32/34 and VGG16. Table 4.8 and Figure 4.7 shows that Tabula reduces runtime by up to 50× across different neural networks, bringing execution time below 1 second per inference for the majority of the networks. Bigger networks are increasingly bottlenecked by ReLU operations, and hence Tabula's runtime reduction increases in magnitude with the size of the neural network under consideration. Figure 4.8 shows a breakdown of where execution time is being spent, for both Tabula and garbled circuits. As shown, Tabula reduces the runtime spent on computing activation functions by up to orders of magnitudes. With bigger networks, the impact of executing nonlinear activation functions is larger. Hence, Tabula sees greater runtime improvement on larger networks. Additionally, in the runtime breakdown chart, the linear layers for Tabula were considerably slower than the linear layers when using garbled circuits – we believe that cache effects caused this difference in performance, as Tabula keeps all tables in memory, which may have overflowed to swap memory. Despite the slowdown in the linear layers, this has negligible impact on runtime due to non-linear layers being the dominant cost, and Tabula sees considerably performance gains by being faster on the nonlinear layers.

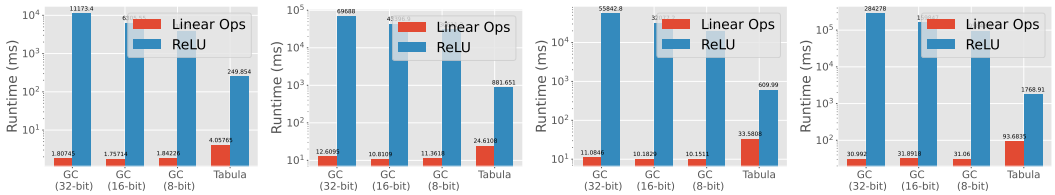


(a) MNIST LeNet **(b)** CIFAR10 ResNet32 **(c)** CIFAR100 VGG16 **(d)** CIFAR100 ResNet34

Figure 4.7: Tabula overall runtime for a single inference versus accuracy for different tasks and neural networks. Each point is annotated with A_n, specifying the precision of activations for that run. At activation precisions 10-12 (achieving within 1-2% of baseline accuracy), Tabula achieves significant runtime speedup ($> 10\times$) over garbled circuits.

Network	ReLUs	32-bit Garbled Circuits Runtime (s)	16-bit Garbled Circuits Runtime (s)	8-bit Garbled Circuits Runtime (s)	TABULA Runtime (s)	Speedup (vs 32/16/8 bit GC)		
LeNet	58K	11.1	6.3	3.9	.29	38.3×	21.7×	13.4×
ResNet-32	303K	69.7	43.4	30.6	.97	71.8×	44.7×	31.5×
VGG-16	284.7K	55.9	32.1	19.9	.67	83.4×	47.9×	29.7×
ResNet-34	1.47M	284.3	159.9	95.9	1.85	153.7×	86.4×	51.8×

Table 4.8: Tabula total online runtime speedup compared with garbled circuits. Compared to garbled circuits, Tabula achieves significant runtime speedup during neural network execution by reducing code complexity, communication costs, and memory/storage overheads.



(a) MNIST LeNet **(b)** CIFAR10 ResNet32 **(c)** CIFAR100 VGG16 **(d)** CIFAR100 ResNet34

Figure 4.8: Runtime breakdown across linear and nonlinear (ReLU) layers comparing Tabula with 8-bit inputs and garbled circuits with 32,16,and 8-bit inputs. Tabula achieves significant performance gains on nonlinear layers, leading to major runtime speedups.

4.4.4 Preprocessing Costs

We benchmark our proposed algorithm for preprocessing Tabula tables against garbled circuits preprocessing times to demonstrate that Tabula preprocessing costs are comparable to garbled circuits.

Preprocessing Runtime & Communication Costs

We compare runtime and communication costs for initializing a single ReLU operation. Table 4.9 shows the cost of preprocessing a single ReLU operation for Tabula with 8-bit inputs, and garbled circuits. In terms of communication costs, Tabula is comparable to GC with 32-bit inputs; however, Tabula requires more communication than GC with 16/8 bit inputs. In terms of runtime, Tabula generally requires significantly more computation than garbled circuits, leading to higher runtime. The majority of Tabula preprocessing runtime is spent towards computing field operations for performing the multiply-add-accumulate operation between the outer product and the nonlinear function (recall that computation costs for an 8-bit input scales as $O(256^3)$). These computation costs can be significantly decreased through further parallelization and vectorization.

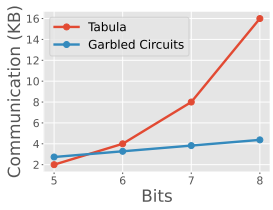


Figure 4.9: Tabula preprocessing communication cost vs Garbled Circuits for different number of bits.

Metric	Tabula Preprocessing (8-bit)	Garbled Circuits Preprocessing (32-bit)	Garbled Circuits Preprocessing (16-bit)	Garbled Circuits Preprocessing (8-bit)
Runtime (ms)	6	.155	.092	.053
Communication (b)	16384	17920	8960	4480

Table 4.9: Tabula vs Garbled Circuits runtime and communication preprocessing costs for a single ReLU operation. Note: Tabula preprocessing costs, like runtime costs, stay constant regardless of activation function, unlike garbled circuits.

We further show the effect of number of bits used for the activation function on preprocessing communication costs. As each bit that is eliminated reduces the size of the table by a factor of 2, saving a single bit exponentially decreases runtime and communication costs. As seen in Figure 4.9, at around 5 bits Tabula preprocessing communication costs become lower than communication cost for garbled circuit at the same bitwidth.

4.4.5 End-to-end Preprocessing Communication Costs

We additionally compare end-to-end preprocessing communication costs across various models (LetNet, ResNet, VGG) between Tabula and Garbled Circuits. Table 4.10 shows a comparison of the communication costs between different models. Again, Tabula requires more communication than GC with 8/16-bit inputs but less than GC with 32-bit inputs, due to the need for computing outer products using Beaver triples that scale with the cardinality of the field. Although this is costly, results show that preprocessing can be feasibly performed at similar cost to GC with 32-bit inputs. Further research and algorithmic developments may drive down the preprocessing cost of

initializing Tabula tables.

Network	ReLUs	Tabula Preprocessing (8-bit)	Garbled Circuits Preprocessing (32-bit)	Garbled Circuits Preprocessing (16-bit)	Garbled Circuits Preprocessing (8-bit)
LeNet	58K	906 MB	991 MB	496 MB	248 MB
ResNet-32	303K	4.6 GB	5.05 GB	2.53 GB	1.27 GB
VGG-16	284.7K	4.3 GB	4.75 GB	2.37 GB	1.19 GB
ResNet-34	1.47M	22.4 GB	24.5 GB	12.25 GB	6.13 GB

Table 4.10: Preprocessing communication cost comparison between Tabula and garbled circuits for various neural network models. Tabula has comparable preprocessing costs compared to garbled circuits.

4.5 Conclusion

Tabula is a secure and efficient protocol for computing nonlinear activation functions in secure neural network inference. Our approach obtains considerable computational benefits over garbled circuits and other approaches to securely computing nonlinear functions. To conclude, we point out the following observation: quantization, as applied to improve standard neural network performance, typically obtains sublinear runtime improvements (as low bitwidth ops typically do not scale linearly in perf. with bits due to hardware inefficiencies), and linear memory improvements. Through our method, quantization as applied to secure neural network inference, obtains superlinear runtime/communication improvements that scale with the complexity of the underlying non-

linear operation, and exponential memory improvements. We believe that, quantization, an already important performance improvement technique for neural networks, will be even more crucial for secure neural network inference, and that our method Tabula is a key approach towards realizing this fact. Additionally, Tabula will see improvement to both the online and offline phases with further advancements to neural network quantization. Tabula is a step towards sustained, low latency, low energy, low bandwidth real time secure inference applications.

5

GPU-based Private Information Retrieval for On-Device Machine Learning Inference

In this chapter we leverage another important system acceleration technique in machine learning, name GPU acceleration, towards making private machine learning systems more efficient. Specifically, we identify that embedding tables in recommendation systems may be too large to store on device and must be stored on the server; accessing them privately requires cryptographic methods to

ensure user data privacy, but unfortunately these methods are computationally expensive. To this end, we develop a novel GPU algorithm for speeding up these private cryptographic accesses and show how this leads to considerably more efficient private machine learning systems.

5.1 Introduction

Privacy is an important consideration for real-world machine learning (ML) applications that use user data. For privacy-sensitive ML applications, users' demand for stronger privacy protection, as well as regulations[70,36] and platform policies[13,84], all increasingly limit the use of private user data. For example, recommendation models, which represent a significant portion of today's ML workloads in practice, inherently rely on individual user data in order to provide personalized recommendations. Ideally, recommendation systems should provide suggestions to users without revealing private user features even to the service provider.

On-device ML inference is a promising solution to provide stronger privacy, as it enables model inference without requiring clients to share private input features with the service provider. Unfortunately, a pure on-device ML inference solution is impractical for many applications such as recommendation, as these applications often require access to an embedding table that is too large to store on device. For example, recommendation models access tables that often take gigabytes or even terabytes of memory[92,179,58,176,242]. These embedding tables are accessed using user features that are important inputs to the recommendation model, and dropping them may negatively impact model quality. Large embedding tables pose a dilemma: storing large embedding tables on device is impractical given device limitations while storing them in the cloud and directly accessing them in the clear could leak private information.

To address this issue, we propose using private information retrieval (PIR) to privately query large embedding tables stored on servers. In this work, we consider distributed point function (DPF)-

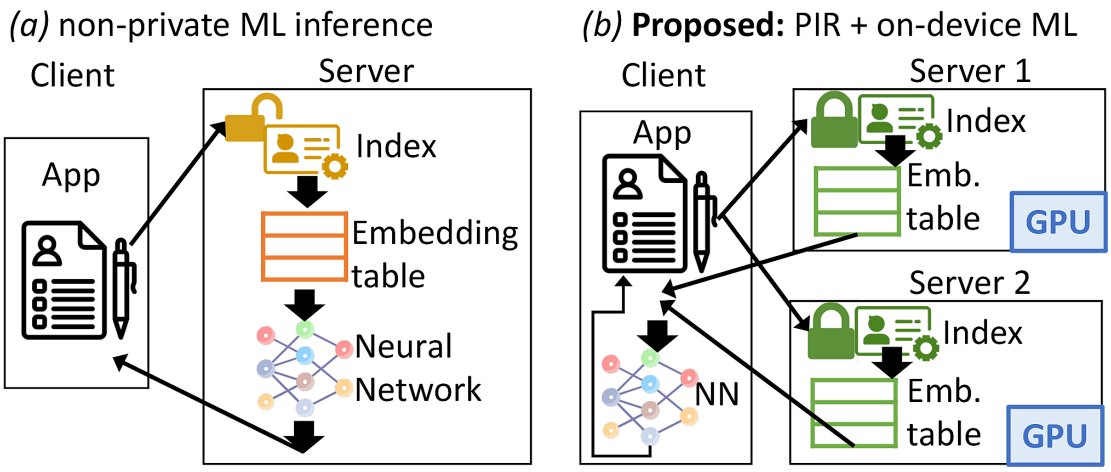


Figure 5.1: (a) The traditional non-private approach to ML inference, and (b) the proposed approach for private on-device ML inference. Using PIR, a CPU-based client privately obtains embeddings from two GPU-accelerated non-colluding servers; these embeddings are subsequently used as inputs to the client's on-device neural network.

based PIR, in which private embedding lookups are performed by constructing and evaluating DPFs on two non-colluding servers (Figures 5.1 and 5.2). A two-server DPF-PIR scheme is attractive as it is much more efficient in terms of computation and communication compared to single-server PIR schemes[73,161]. The two-server model is also widely used in the previous work on secure multi-party computation (MPC) for privacy-preserving machine learning[224,198,192,133] or private analytics[35,116].

Despite their advantages, DPF-based PIR protocols still exhibit massive computational overhead[75,32], making them difficult to deploy in large-scale applications that require high throughput. The computational overhead stems from evaluating the DPFs on the servers, which entails executing a significant number of expensive cryptographic operations[75,32]. For example, expanding a typical DPF for a table with one million entries requires performing at least one million AES-128 encryption operations. The cost is amplified during ML inference where a model may access multiple embedding entries[92,93]. The computation and communication requirements of DPF-based PIR make deploying it to real-world ML applications a considerable challenge.

5.1.1 Our Contributions

We develop a system to efficiently and privately serve embeddings for on-device ML, with the primary focus on on-device recommendation models that require privately accessing large server-side embedding tables. Note that recommendation models represent an important application that account for a significant portion of the computational resources for ML in practice[93,123]. While our work primarily targets private on-device recommendation, the proposed PIR system can also be applied to other on-device ML models that need private access to server-side embedding tables. Embedding accesses for on-device ML, particularly on-device recommendation, have several unique properties and requirements compared to other applications that might use PIR: 1) embedding table entries are often short, between 64-1024 bytes, 2) multiple embedding table entries are often accessed together in a batch as part of a single model inference, and 3) throughput, latency, and model quality are all critical to an application's success. We leverage these properties to design a novel GPU acceleration scheme for efficiently performing PIR on GPUs, and, additionally, co-design PIR with the ML application to facilitate better trade-offs between model quality and system performance. Similar to other systems work in the PIR domain [161,59,72,43], our contributions focus on performance improvements.. Our specific contributions are listed below.

GPU-accelerated PIR We develop a set of novel optimizations to efficiently perform PIR on GPUs. Our optimizations enable high-throughput, low-latency DPF execution, allowing us to scale to tables with millions of entries. We observe that DPF evaluation is compute-bound due to their heavy cryptographic instruction mix, and leverage the fact that GPUs are especially well suited to perform these computationally heavy operations. Yet, performing PIR on a GPU requires exploiting multiple types of parallelism in PIR while carefully balancing computation, communication, and memory usage. Our GPU acceleration, over an optimized CPU baseline[85], obtains $> 1,000\times$ speedup over single-threaded CPU execution, and $> 20\times$ speedup over multi-core execution. To the best

of our knowledge, this work represents the first to explore high-performance GPU implementations of DPFs. We note that our GPU implementation accelerates the state-of-the-art DPF algorithm [75], which exhibits an optimal communication cost of $O(\log(n))$ and an optimal computation complexity of $O(n)$. Beyond private embedding table accesses for ML, our GPU PIR can be used to accelerate any PIR applications such as checking compromised passwords. Our code is open sourced at `https://github.com/facebookresearch/GPU-DPF`.

ML + PIR Co-Optimization To further improve performance, we develop strategies utilizing application-specific data access patterns to co-optimize PIR with the ML application. Traditional batch PIR algorithms [118,102,12], which allow privately obtaining multiple entries together, may impact ML inference quality because they only retrieve entries probabilistically and may drop some queries. We co-design a new batch PIR algorithm for ML tasks to obtain a better trade-off between model quality and system performance. We comprehensively evaluate the resulting performance improvements and model quality of the new batch PIR scheme on applications including WikiText2 language model [163], Movielens recommendation [100], and Taobao recommendation [217]. The results show that the proposed optimizations utilizing application-specific data access patterns can increase the ML inference throughput by up to 100× over a straightforward PIR system design on a multi-core CPU, while maintaining the model quality and limiting inference communication and latency within 300 KB and 300 ms, respectively.

5.2 Related Work

Privacy-preserving Computation Techniques Prior work on privacy-preserving ML investigated techniques such as fully-homomorphic encryption (FHE) [125,171], secure multi-party computation (MPC) [133,192,224,198], and trusted execution environments (TEEs) [108,109,234]. Unlike these prior studies, which primarily focus on protecting dense computation in neural networks, we investigate how

to privately access large embedding tables in recommendation and language models.

Recent work on FHE acceleration [5,195,64,131,246,63,237,164,130,112,199] suggests that FHE-based CNN models can run with low latency. Yet, they still suffer from low throughput. Due to the high computation demand of FHE, FHE accelerators typically use the entire chip (ASIC/FPGA/GPU) to run one inference at a time. While FHE has the potential to enable private inference for any model in the cloud, it is not yet efficient enough for high-throughput use cases.

Private Information Retrieval PIR can be categorized into single-server protocols based on homomorphic encryption (HE) [161,73,44,147] and n-server ($n \geq 2$) protocols based on DPFs [72,43,59]. We focus on two-server DPF-based PIR protocols, as they are significantly more computation- and communication-efficient than single-server schemes [147,112,131,72,43,59]. For example, querying a 1B entry table with a two-server protocol is over 1000× more communication-efficient (2KB vs 3.6MB) [147] and multiple orders of magnitude more computationally-efficient than single-server protocols [103,12,8,6,177,161]. For a 1M-entry table, state-of-the-art HE PIR [161] requires 14KB-60MB communication whereas our DPF-based system requires only 1.25 KB. HE PIR's advantage over a DPF-based PIR system is that it only requires one server, rather than two non-colluding servers, enabling PIR under a stronger threat model. Compared to n>2 DPF approaches, two-server DPF-based PIR protocols are more communication-efficient: 2-server DPF exhibits $O(log(n))$ communication [75,32] while n>2-server DPF exhibits $O(\sqrt{n})$ communication [30].

The two-server PIR protocols require the two participating servers hosting the (embedding) tables to be non-colluding. This threat model with two (or more) non-colluding servers is commonly used in a large body of work on secure multi-party computation (MPC) [133,192,224,198,72,43,59]. Different from other computation with MPC, in DPF, no communication is required between the servers, and thus, the two servers can be hosted by different cloud providers with minimal performance overhead. Further, recent advances in MPC platforms make such a system increasingly realistic [175,167,169,9,67,86]. One realistic scenario is for the companies that want to provide

strong privacy standards to form a consortium to act as each others' non-colluding second party; these efforts [175,168] are seeing increasing adoption. Remote attestation capabilities in public cloud TEEs [9,169,83,234] can also be used to further ensure the integrity of two parties.

Batch Private Information Retrieval Various approaches for batch PIR [201,12,118,12,102] have been proposed. We show that noise tolerance of ML allows the use probabilistic PIR protocols like [201] with minimal accuracy loss.

On-device ML On-device ML has been studied for recommendation [101,79], speech recognition [10], translation [216], etc. Our work uses on-device ML for privacy, and enables the private use of large server-side embedding tables.

5.3 Private On-Device ML Inference

5.3.1 Threat Model

The goal of private on-device inference is to perform ML inference using data on a user device without revealing them to a server owned by a service/cloud provider. In the context of recommendation systems, on-device inference can allow private user data only available on a client device to be used to provide more relevant recommendations, while ensuring that no private data leaves the device. To reduce the burden on user devices, a server-side recommendation model can send a set of candidate recommendations based on less sensitive user features available on the server, then a smaller on-device model can more accurately rank the candidates leveraging private on-device user data without revealing them to the server. In our study of a real-world model, we found that even a small (several MB) on-device MLP model can noticeably improve recommendation accuracy when combined with server-side embedding tables.

We assume that the computation part of the ML model can run on the user device given the increasing trend of hardware accelerators and optimizations for client SoCs, but that *embedding tables* of

categorical/sparse features (described below) are too large to be placed on individual devices and hence are accessed remotely (Figure 5.1). We further assume that only a very small fraction of the table is used per-inference.

As the indices to embedding tables represent private categorical feature values, private on-device inference must ensure the confidentiality of table indices while allowing the use of server-side embedding tables. For this purpose, we leverage private information retrieval (PIR) protocols under the honest-but-curious threat model. The user/client device and its software are trusted. While remote servers are untrusted, they are assumed to follow the protocol. The honest-but-curious threat model is widely used in previous private inference work [133,192,215,72,43,171]. The model may be extended to a malicious setting by using PIR protocols that protect against a malicious server deviating from the protocol and produce wrong answers (e.g. authentication for PIR [42]). We also note that incorrect PIR responses only lead to non-optimal suggestions in recommendation models; selective failure attacks [111] are difficult to perform because failures are not visible to attackers.

Like previous work on privacy preserving ML and analytics using multi-party computation (MPC) [72,43,59,224,198,192,133,35,116], we further assume a two-server model where the two servers are non-colluding. This two-server setup can be practically realised by having two different cloud vendors host and manage the two servers or having another industry actor host the second server. Forming such a privacy consortium among companies is emerging in industry [175]. See Section 5.2 for further discussions.

5.3.2 Key Challenge: Large Embedding Tables

Unfortunately, the embedding tables in machine learning models, especially for recommendation models, are often too large for individual devices [92,179,58,176,242], making a pure on-device inference solution impractical. An embedding table is a large table that maps categorical features into dense vectors that encode semantic information. For example, categorical (sparse) features may include a

Table 5.1: Embedding table sizes for popular public datasets and models spanning across language and recommendation.

Application	# of Embedding Entries	Entry Size	Embedding Table Size
Criteo 1 TB Rec.	>100,000,000	~128B	>90 GB
Criteo Rec.	~10,000,000	~128B	~5 GB
FastText Emb. (Language Model)	~2,000,000	~1024B	>1.9 GB
Taobao Rec.	~900,000	~128B	~109 MB
WikiText2 (Language Model)	~131,000	~512B	~64 MB
Movielens-20M Rec.	~27,000	~128B	~3 MB

user's click or search history. The value of a categorical feature is used as an index to an embedding table where each row of the table holds the vector corresponding to that categorical feature value (Figure 5.1). Embedding tables can have as many rows as the number of possible values in the categorical feature space so their size can grow quickly.

Recommendation models use several user and product input features to predict whether a user is likely to interact (e.g., click or purchase) with the product [179,242]. These models may use user data such as the list of products the user recently purchased [242]. As the number of products can be on the order of millions, the corresponding embedding table can reach several GB to TB in size [92,176,79]. Compressing the table is difficult for many real-world models, as it leads to significant accuracy drop [240]. Recommendation models represent our primary target use case given their reliance on large server-side tables.

Language models are another potential example of an ML application that may require access to server-side embedding tables. Language models empower applications such as next-word prediction,

language translation, and speech recognition. Language models map words into a latent embedding space using word embedding tables [163]. As there may be hundreds of thousands of different words, with each embedding vector being hundreds of bytes long, it quickly becomes impractical to store the entire word embedding table on-device, especially for natural language translation models supporting multiple languages [61,182]. Although there are alternative techniques to compress the embeddings (e.g., character embeddings, sentence level representations, etc.), word embeddings are considered to be more efficient to train in a regime with less training data [61]. We discuss the language model as a potential example in our study to show that our system can be adopted for multiple types of on-device models that need large server-side embedding tables. However, we note that on-device inference for language models is limited to smaller language models that can run on a client device. Private inference for large language models need additional computation beyond embedding table accesses to be securely offloaded to cloud servers. Also, the embedding tables for language models are typically much smaller compared to the tables for recommendation models.

Table 5.1 summarizes the size of the embedding tables of some popular datasets/models. The size ranges from several MBs to hundreds of GBs. On the other hand, the mobile app size is on average 34MB, and seldom exceeds 200MB even in extreme cases [173]. Embedding tables, especially for recommendation models, can easily exceed this range, which makes deploying them on-device impractical [79].

5.3.3 Example: Real-World Recommendation Model

As a concrete use case for private on-device ML inference with sparse features, we studied a real-world recommendation model where some of its input (user) features can only to be used on a client device for strong privacy protection. For this model, such "device-only" sparse features represent 7 out of top 25 features when the input features are ranked by their feature importance score*. Re-

*This score measures the change in the accuracy when a particular feature is changed to a random value.

Table 5.2: The embedding tables for a real-world recommendation model, showing the number of entries, the table size, and the average number of entries accessed per inference. The numbers are shown for the top 5 device-only sparse features with highest importance.

# Entries	Avg Queries Per Inference	Table Size (# of entries * 144B)
7,614,589	13.9	1.02GB
20,000,000	47.3	2.68GB
20,000,000	25.7	2.68GB
2,989,943	3.2	400MB
20,000,000	14.9	2.68GB

moving the device-only features significantly degrade the model's utility (accuracy), and a small (several MB) on-device model can provide good accuracy if the embedding tables can be accessed privately.

Table 5.2 shows the embedding table size and the number of accesses per inference for the top 5 sparse features that are only accessible on-device. Similar to the public datasets, the embedding tables are too large to be sent and stored on a client device, and each table entry is relatively small (144 bytes) – on average only at most 1-10KB of entries are fetched from the table for each inference. Our study also found that the user features change relatively slowly; the sparse user features mostly stay the same for two consecutive recommendations for one user. If a client device keeps recently fetched embedding table entries, only 2.44% of sparse features are new and need to access embedding tables on a server. Even though Table 5.2 shows that several tens of embedding table entries are used for each inference, the temporal locality means that only a few new entries need to be read from the server.

5.3.4 Our Approach: On-Device ML Inference with PIR

To enable private on-device ML applications that require access to large embedding tables, we propose using private information retrieval (PIR)[41,59]. PIR allows a user to query a table without revealing which index was accessed to the table holder, i.e., the server that hosts the embedding table. We propose to keep large embedding tables on the cloud servers, and use PIR to query the table upon an embedding table access by a client's device (Figure 5.1).

We use a PIR protocol based on a distributed point function (DPF)[75,32], which protects accesses using two non-colluding servers. We choose PIR rather than oblivious RAM (ORAM)[77,210,229,211,115,11,14,66,194,226,227,189,149,187,34,235], another popular cryptographic technique to hide an access pattern to memory, because ORAM is designed to protect accesses from a single entity. In the on-device ML scenario, multiple users simultaneously send query requests. DPF-based PIR methods are more efficient in terms of communication and computation compared to single-server PIR schemes that employ homomorphic encryption [161,44,73,147]. A key challenge in employing DPF-based PIR is its high computational intensity due to heavy cryptographic operations. In the following section, we describe how the DPF-based PIR can be efficiently accelerated on GPUs.

5.4 Accelerating PIR using GPUs

Algorithms for PIR exhibit significant overhead due to their heavy cryptographic operations and cannot be immediately adopted for private on-device inference. Below, we 1) briefly introduce PIR and DPF, 2) analyze their characteristics to understand how GPUs may accelerate them, and 3) describe our optimizations for GPU acceleration.

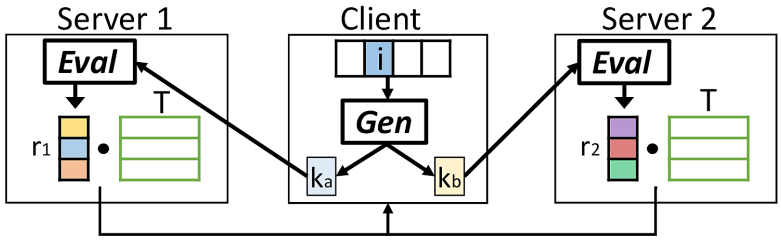


Figure 5.2: DPF based PIR scheme. The client computes *Gen* to obtain two keys (k_a, k_b) that represent a secret index and sends them to the servers. The servers individually compute *Eval* to obtain secret shares of the answer, from which the client can later retrieve the desired embedding. *Eval* is computationally expensive and is our main target for acceleration.

5.4.1 Fundamentals of PIR and DPF

Private information retrieval (PIR) based on distributed point functions (DPF) allows a user to access an index in a table, shared across two non-colluding servers, without leaking the index to the table holders. In DPF-PIR, the client sends a key that represents the index it wants to privately query. The server, upon receiving the key, performs expensive cryptographic operations to service the user's query (Figure 5.2).

Naive PIR Assume a client C seeks to privately access entry $T[i] \in \mathbb{F}_p^D$ from a table $T \in \mathbb{F}_p^{L \times D}$ that is duplicated across two non-colluding servers, S_1 and S_2. Here, L is the number of entries in the table, D is the vector length of each entry, and $\mathbb{F}_p$ is an integer field with modulus p. A simple but highly inefficient approach is for the client C to generate and send a random vector $r_1 \in \mathbb{F}_p^{1 \times L}$ and a second vector $r_2 \in \mathbb{F}_p^{1 \times L}$ to S_1 and S_2, such that they add up to a one-hot indicator vector $I(i)$ whose entries are all 0's except at the i^{th} position where it is 1 ($r_1 + r_2 = I(i)$). Upon receiving the vectors, the servers individually compute and return $r_1 \times T$ and $r_2 \times T$ to the client, from which the client can retrieve $T \times (r_1 + r_2) = T \times I(i) = T[i]$. Information theoretic privacy is ensured as r_1 and r_2 are *secret shares* of the indicator vector that do not leak any information about i individually[204]. This simple approach incurs large communication overhead because the size of r_1 and r_2 is proportional to the size of table T, making the communication overhead $O(L)$.

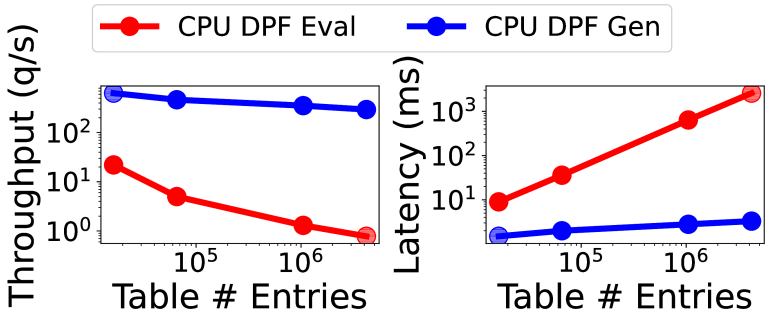


Figure 5.3: *Gen* vs *Eval* performance. *Gen* is highly efficient and is not our target for optimization.

DPF-PIR The generalization of the approach described above is a cryptographic primitive known as a *distributed point function* (DPF). DPF is an algorithmic construct that allows a client to *generate* two compact keys k_a, k_b, such that when the keys are *expanded* across a set of indices, they yield secret shares of the indicator vector $I(i)$.

Formally, a DPF consists of two algorithms,

- $Gen(1^\lambda, i \in 0..L-1) \rightarrow (k_a, k_b)$, which takes security parameter λ and input i, and generates two keys k_a, k_b.
- $Eval(k, j) \rightarrow \mathbb{F}_p$, which takes a key k and an evaluation index j and outputs a field element.

such that, $Eval(k_a, j) + Eval(k_b, j) = \begin{cases} 1 & j = i \\ 0 & j \neq i \end{cases}$.

Gen is a key generation process where a client encrypts the index it wishes to query into two keys k_a and k_b, which are respectively sent to the two non-colluding servers. *Gen* is relatively lightweight compared to *Eval* ($O(log(L)$ computation)[75,32], and can be quickly computed even on resource-constrained client devices as shown in Figure 5.3.

Eval is the key evaluation process that is performed on the servers. Upon receiving k_a or k_b, the servers respectively compute $T \times Eval(k_a, \{0 \ldots, L-1\})$ and $T \times Eval(k_b, \{0 \ldots, L-1\})$ and return the result, from which the client can obtain $T \times (Eval(k_a, \{0 \ldots, L-1\}) + Eval(k_b, \{0 \ldots, L-$

$1\}))) = T \times I(i) = T[i]$. *Eval* requires at least $O(L)$ computation[75,32] and is the major bottleneck (see Figure 5.3). Our work focuses on accelerating the *Eval* function. Figure 5.2 depicts the overall DPF-PIR scheme.

A DPF should be computationally secure, meaning that given just one of the keys and no other information, it should be difficult to recover the client-queried index i without doing computation proportional to $O(2^\lambda)$. There are many different implementations of DPFs, each with a different computation/communication trade-off. We consider the DPF construct described in[75], which provides optimal asymptotic communication complexity of $O(\lambda \log(L))$ and optimal evaluation computation complexity of $O(\lambda L)$.

In this DPF algorithm, the evaluation of DPF involves expanding a GGM-style[76] computation tree. Keys k_a and k_b each consists of two two-dimensional codewords, $\{C_0 \in \mathbb{F}_{2^\lambda}^{2\times(\log(L)+1)}, C_1 \in \mathbb{F}_{2^\lambda}^{2\times(\log(L)+1)}\}$. The server uses the codewords and expand them into a tree (Figure 5.4) to get the secret shares of the indicator vector, using a recursively-defined helper function P:

$$Eval(k,j) = P(d = log(L), j) \tag{5.1}$$

$$P(0,0) = C_0[0,0] \tag{5.2}$$

$$\begin{aligned} P(d,j) = {} & PRF_{P(d-1,\lfloor \frac{j}{2} \rfloor)}(j \bmod 2) \\ & + C_{P(d-1,\lfloor \frac{j}{2} \rfloor) \bmod 2}[j \bmod 2, d] \end{aligned} \tag{5.3}$$

Here, d is the depth of the node (0 for the root, $log(L)$ for the leaves), j is the index of the node within each depth (0 being leftmost), and $PRF_s(x)$ is a *pseudorandom function* that encrypts a message x with an encryption key s, such as AES-128.

Figure 5.4 illustrates how *Eval* works with an example. Assume the client wants to query a table of $L = 4$. The client generates and sends a key to each server, where each key consists of two 2×3

codewords, C_0 and C_1. Using the keys, the server must calculate $Eval(k, 0)..Eval(k, 3)$ and multiply them to the table. To calculate, *e.g.*, $Eval(k, 3)$ (which is $P(2, 3)$ from Equation 5.1), the server needs to calculate $P(1, 1)$, calculating which in turns requires $P(0, 0)$ (Euqation 5.3). The calculation can be seen as an evaluation of each node in a binary tree from the root to the leaf; a child node is computed using the result from the parent node and C_0, C_1.

Evaluating a single node requires a single *PRF* call and an addition, requiring $O(\lambda L)$ computation for the entire tree. Communication overhead is proportional to the size of the keys, resulting in $O(\lambda \log(L))$ total communication. In practice, λ is typically a 128-bit field integer to ensure sufficient computational security. After computing all the leaf nodes of the tree, the output is a vector of λ-bit (128-bit) field values; the final secret shares of the entry are obtained by performing an integer dot product between the computed 128-bit field values and the table. Note that tables with *arbitrary* sized entries (i.e: much greater than 128-bits) may be supported with no additional DPF evaluation, as we can view these large-entried tables as a 2-D matrix, with the large entries subdivided into groups of 128-bit values; we may then perform a matrix-vector-multiplication with the prior DPF output to obtain secret shares of the table lookup. This works as performing a matrix-vector-multiplication between the DPF vector and the 2-D table selects the entire set of entries that corresponds to the selected index. In practice, the dot products for multiple queries to a single table are batched together as a single matrix-matrix multiplication to enhance performance. We refer to[75] for details on key generation.

5.4.2 Accelerating PIR with GPU

Starting Point: Batched DPF Execution

We begin by observing that parallelism in DPF computation can be exposed in two dimensions: 1) parallelizing the evaluation of a *single* DPF; and 2) evaluating *multiple* DPFs in parallel. The latter,

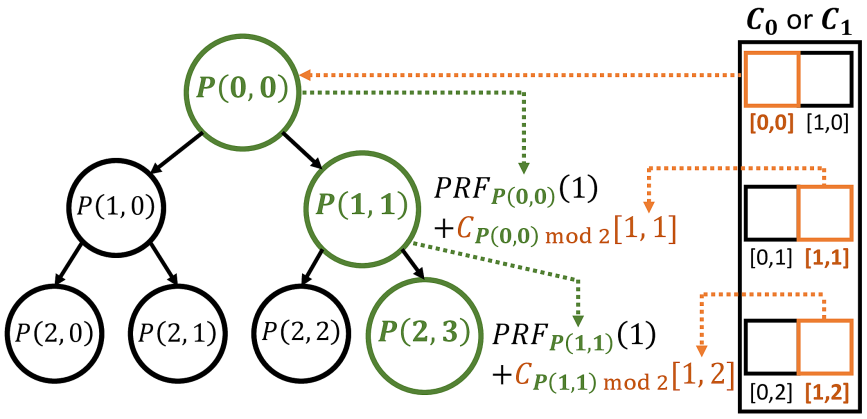
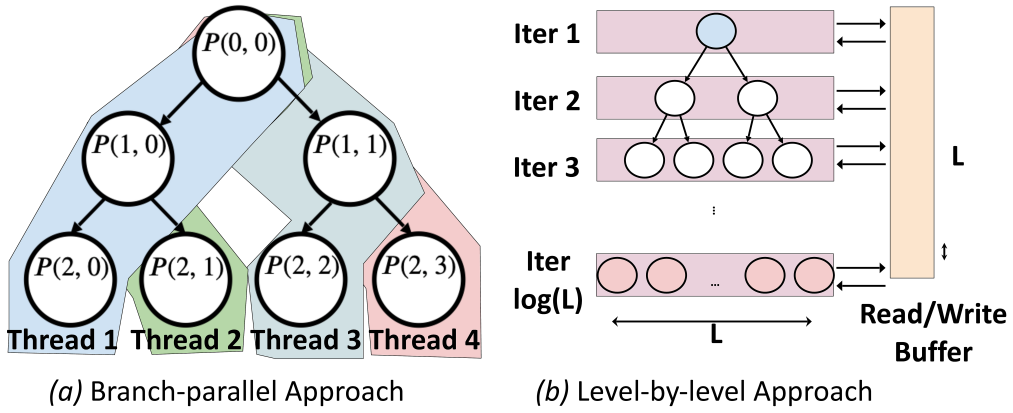


Figure 5.4: Example of the DPF computation using tree expansion. DPF expansion involves computing the leaves of a binary computation tree which evaluate to a secret-share of a one-hot vector. Computing each node requires evaluating its parent node which involves calling a PRF and adding to it a a codeword value indexed by the height and parity of the node.

evaluating multiple DPFs in parallel, is understood as standard *batched execution* and is an implicit starting point inherent to our proposed optimizations. At the GPU level, parallelizing the evaluation of a single DPF is done via thread-level parallelism, and batched-execution is performed by evaluating multiple DPFs on multiple blocks via block-level parallelism. Under this framework, approaches falling under the two categories can be applied jointly with minimal interaction, and hence, unless otherwise noted, batched-execution with batch-size B is assumed in all subsequent parallelization approaches. While batching itself is not a novel component of our proposed approach, batching is indeed important for high utilization of GPU resources (Figure 5.9a). We also found that the batch size needs to be carefully selected based on the size of the table and the DPF paralleization strategy to balance latency, throughput, and memory requirement.



(a) Branch-parallel Approach *(b)* Level-by-level Approach

Figure 5.5: Two naive approaches for parallelizing DPF computation.

Tradeoffs between Branch-parallel and Level-by-level DPF Parallelization Approaches

Two naive approaches to parallelizing the execution of individual DPFs are branch-parallel and level-by-level approaches, shown in Figure 5.5. A branch-parallel approach has each thread independently compute one branch/leaf (or a subset of branches/leaves) of the DPF, while a level-by-level parallelization approach has each thread evaluate the nodes of a single level of the DPF tree in parallel, writing outputs to global memory to be used for computing the next level.

Unfortunately, these two naive parallelization approaches suffer from a major tradeoff between computational redundancy and memory usage, making neither truly efficient nor scalable. A branch-parallel approach suffers from *computational redundancy*. As computing each leaf node requires evaluating all nodes up to the root, each thread in branch-parallel execution re-computes intermediate nodes unnecessarily, as shown in Figure 5.5a. As a result, the overall amount of work becomes $O(L \cdot log(L))$, instead of the optimal $O(L)$.

The level-by-level parallelization approach eliminates this computational redundancy by storing and reusing intermediate node outputs. However, this approach suffers from *memory overhead* as

storing intermediate results consumes significant amount of memory when the batch size and the table size is large ($O(BL)$ for a batch size B). Hence, there is a fundamental tradeoff between these two approaches in balancing computation and memory usage. Figure 5.6 shows that the branch-parallel approach suffers from high number of PRF calls, while the level-by-level approach suffers from high peak memory usage.

Memory-bounded Tree Traversal

The tradeoff between computation and memory usage in Section 5.4.2 motivates a different parallelization strategy. We emphasize that memory usage is a critical factor in accelerating DPFs on GPUs, as memory limitations bound the effective batch size that may be used; consequently, reducing memory usage allows for the use of larger batch sizes which significantly increases throughput. In other words, reducing memory usage while ensuring efficient parallel execution is the key to efficient DPF acceleration on a GPU. To this end, we develop *Memory-bounded tree traversal* (Figure 5.7a), a parallelization scheme that is: 1) optimal in terms of computation ($O(L)$ work); and 2) exhibits memory overhead that scales *logarithmically* with the size of the table, instead of linearly as in the level-by-level approach.

Memory-bounded tree traversal performs a depth-first evaluation of the DPF tree, with chunks of K nodes evaluated at once in parallel for each level (Figure 5.7a). Unlike the level-by-level approach that computes and saves *all* nodes in each level, the new approach only evaluates K nodes per level, then immediately re-uses these node outputs by recursively computing the nodes at the next level that require these outputs, and subsequently discarding the previous node outputs. Thus, at each level, only K more nodes need to be cached to memory. Hence, this approach reduces memory overhead from $O(BL)$ to $O(BKlog(L))$, making the memory overhead affordable even for large tables ((Figure 5.8a)). K, which is a hyperparameter that determines how many nodes to expand in parallel, must be large enough to expose sufficient parallelism but small enough to avoid out-of-memory

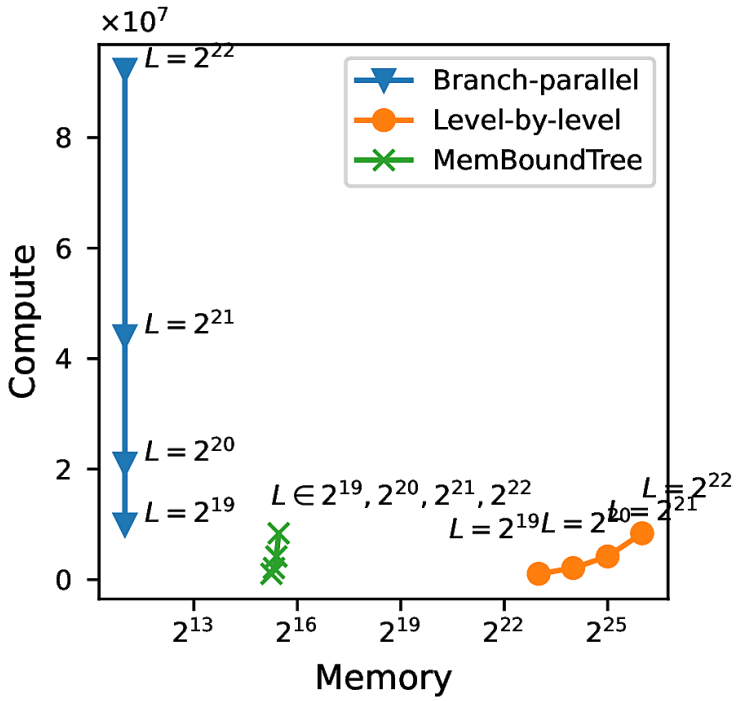


Figure 5.6: The number of PRFs evaluated (compute) and the peak memory usage (memory) for different parallelization strategies, across different table sizes (L). For both axes, lower is better. The branch-parallel approach redundantly calculates extra PRFs, while the level-by-level approach suffers from high memory usage. Our proposed approach, memory-bounded tree traversal (MemBoundTree), simultaneously performs less work while requiring much less memory – MemBoundTree can significantly (i.e., up to 10x) improve performance by reducing memory consumption and allowing the use of larger batch sizes, which increases utilization.

complications. We empirically set $K = 128$, which balances compute utilization and memory usage on a V100 GPU (Figure 5.8b). Memory-bounded tree traversal achieves both optimal work and low memory usage (Figure 5.6). As a result of achieving optimal work, low memory usage, and maximizing parallelism, the memory-bounded tree traversal method can scale to larger batch sizes and hence increase throughput and utilization up to an order of magnitude greater than a naive level-by-level approach. The memory advantage of the memory-bounded tree traversal approach is depicted in Figure 5.6, and achieves utilization benefits of a considerably larger batch size as depicted in Figure 5.9a.

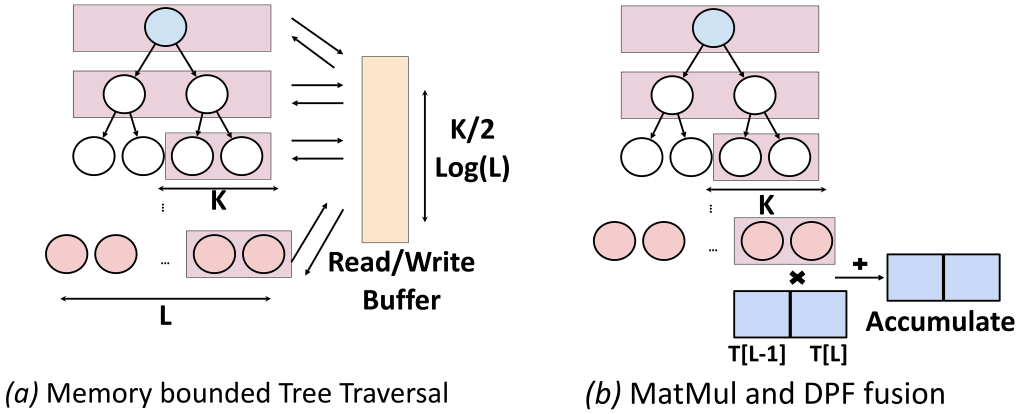


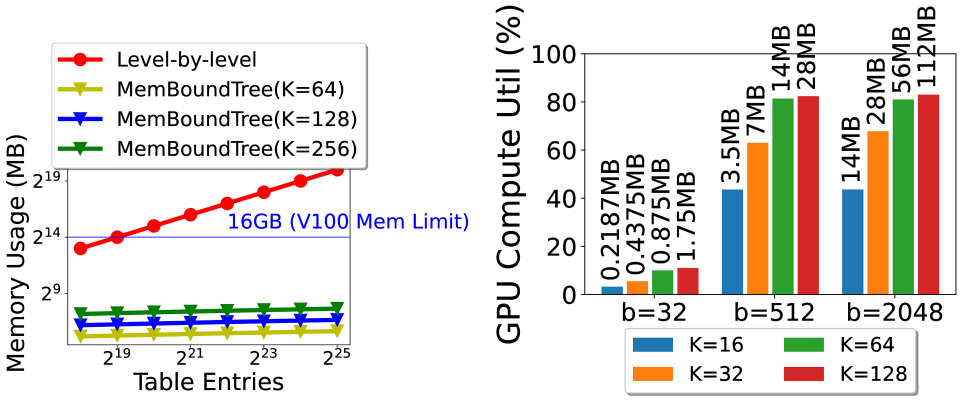
Figure 5.7: Memory-bounded tree traversal and operator fusion for reducing memory overhead.

DPF and Matrix-Multiplication Operator Fusion

After evaluating the DPF, the server needs to perform a matrix multiplication between the large table and the DPF output (Section 5.4.1). If we naively compute the entire output before performing a matrix multiplication, the memory must hold the entire output of the DPF and requires $O(BL)$ space. To keep the memory overhead to $O(BKlog(L))$, we *fuse* the DPF evaluation operator with the matrix multiplication operator (Figure 5.7b). Upon reaching a leaf node, a thread immediately performs a dot product between the table entry and the corresponding leaf node output of size K, accumulating the result in local memory. At the end, threads in a single thread-block coordinate to perform a cross-thread sum of the local registers to obtain the final result, using tree-summation. Fusing DPF has additional performance benefits as it reduces the number of accesses to global memory and allows interleaving between matrix-multiplication and DPF computation.

Batch and Table-Size Aware Scheduling

On large tables ($> 2^{22}$ entries), we observe that a single DPF (batch size of 1) may have enough parallelism to sufficiently saturate GPU resources. Hence, for very-large tables, it is preferable to



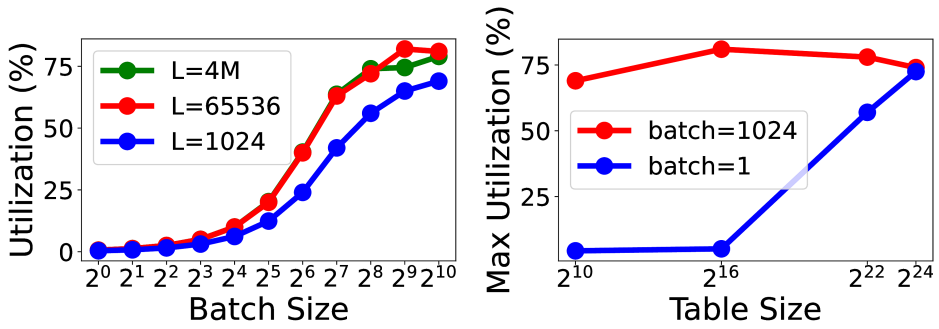
(a) Memory Usage **(b)** GPU Utilization vs K

Figure 5.8: The memory usage and the compute resource utilization of the memory-bounded tree traversal.

use all GPU resources for the computation of a single DPF at a time, which significantly reduces latency, rather than perform batched-execution. We additionally develop a parallelization strategy based on cooperative groups [184] to coordinate all GPU blocks when computing a single DPF. This single-batch strategy is selectively applied only when the table size is very large. Figure 5.9b shows that using cooperative groups with a batch size of 1 can indeed achieve high GPU utilization on extremely-large tables (with a lower latency, which is not shown), while it suffers from low resource utilization if incorrectly applied to smaller tables. We empirically use a threshold of 2^{22} entries to choose between batched execution and cooperative groups.

GPU-Aware PRF Selection

CPUs typically come with built-in hardware for popular PRFs such as AES and SHA-256 (e.g., AES-NI instructions). AES is a natural choice for the PRF on a CPU given built-in CPU hardware primitives. However, unlike CPUs, GPUs do not offer hardware acceleration for cryptographic primitives. As a result, AES computation on a GPU is far more computationally expensive compared to a CPU. Hence, a more careful PRF selection has the potential to provide higher perfor-



(a) Batch size vs Util

(b) Table size vs Util

Figure 5.9: Effect of batch size (a) and table size (b) on GPU utilization. For figure (b), batch=1 utilizes cooperative groups to coordinate all available GPU resources towards computing a single DPF.

mance on a GPU. In this context, we evaluate multiple PRFs including block ciphers (AES), hash functions (SHA-256), stream ciphers (ChaCha20), and others. We mainly show results of PIR performance based on AES-128 to match the standard PRFs used in the CPU PIR baseline. However, we found that PRF selection has a significant impact on GPU PIR performance, and we report these results in the evaluation as well. Particularly, Chacha20, which is a standard stream cipher used in TLS[37], provides noticeable performance gains. Other non-standard PRFs, such as SipHash, can provide even more speed-up, but their security assurance may be weaker as they are not yet widely analyzed or proven in practice. One must consider the performance and security tradeoff of a PRF to determine whether that PRF is suitable for the application at hand.

Note on Scaling to Multiple GPUs

We note that our DPF execution strategies may be applied to multiple GPUs in the case where a single embedding table is too large to fit in a single GPU's memory. A single DPF can be computed across multiple GPUs by having each of the N GPUs evaluate the DPF on a subset of the table indices, then summing the result across GPUs at the end. This approach works because the final DPF

reduction operation (a summation) is linear. Hence, we can linearly scale our DPF execution strategies across multiple GPUs by simply dividing the work in an embarrassingly parallel approach. We note that, in this scenario, each GPU effectively evaluates a DPF on a table of size $\frac{L}{N}$, hence, performance is the same as if evaluating a DPF on a smaller table size. Additionally, with more GPUs, a larger batch size may be needed to fully utilize GPU compute resources since the table sizes are proportionally smaller. Thus, for multi-GPU execution, it becomes more important to maximize batch size by using the memory-bounded tree traversal execution strategy, and a cooperative-groups approach would be less effective.

5.5 Accelerating Batch-PIR with ML Co-Design

Many recommendation/language models require multiple lookups to the same embedding table. For example, recommendation models may lookup the same table tens of times to perform a single inference[92] (e.g., a user can have multiple clicked items, if the clicked-item history is used as a feature). Multiple lookups linearly increase the cost of PIR as simple DPF-PIR only retrieves one entry at a time.

To support multiple tabe lookups more efficiently, we adopt partial batch retrieval (PBR)[201], an algorithm that accelerates the retrieval of multiple entries. PBR comes at a cost; with some probability (when multiple queries map to the same internal bin), queries are dropped, which may negatively affect model quality. Hence, we co-design PBR with ML inference to improve system performance while maintaining the model quality.

5.5.1 Background: Batch Private Information Retrieval

Batch private information retrieval (batch-PIR) is a set of techniques to retrieve multiple private entries from a single table. In this work, we adopt the method proposed in[201], partial batch retrieval

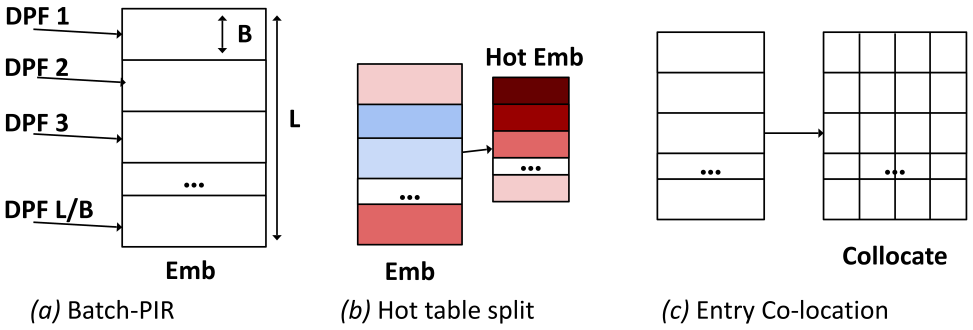


Figure 5.10: Techniques used to co-design PIR + ML. a) Partial Batch Retrieval, b) splitting the table into a smaller hot table, and c) co-locating frequently accessed entries.

(PBR), which operates by segmenting table T into $\frac{L}{I}$ bins of size I, and issuing individual DPF-PIR queries to each bin (Figure 5.10a). This approach saves computation by a factor of $\frac{L}{I}$ in the best-case scenario where the client retrieves $\frac{L}{I}$ entries that are spread across different bins. However, a single PBR can fetch only one query from each bin. If more than one query index fall into the same bin, the rest of the queries except for the one must be dropped.

This limitation leads to a complex tradeoff between the communication efficiency and the accuracy of the retrieval. A large I can reduce the accuracy of the retrieval if multiple desired entries map to the same bin. Conversely, a smaller I yields fewer conflicts, but increases communication costs. This tradeoff naturally affects model quality as dropped queries affect the model's inference.

5.5.2 Co-Designing the ML Model and Batch-PIR

To improve batch-PIR efficiency while minimizing effect of retrieval failures, we propose PIR-ML co-optimizations that improve the tradeoff between model accuracy and performance.

Frequency-Based Hot Table Split Many ML applications access embedding tables following a power-law distribution, where a small number of *hot* indices account for the majority of lookups[93,247]. We leverage this observation and add a small *hot table* that holds the top-K frequently accessed in-

dices in addition to the large *full table* that holds all the embedding entries (Figure 5.10b). The hot table is constructed statically using the observed statistics from the training dataset as part of a preprocessing phase ahead of model deployment, and a small hash table is placed on a client device to provide the hot table index for the categorical feature values that are in the hot table; as this hot table is designed to be small, this index mapping can reasonably reside on client devices. At inference time, a client looks up whether the index they wish to query is in the hot table, and issues two sets of keys: one set that queries the hot table and the other for the full table.

Simply using the hot table as a traditional cache is insecure as it leaks the number of queries to the hot/full tables. To avoid this information leakage, we predetermine a fixed number of queries Q_{hot} and Q_{full} to issue to the hot and full tables, respectively, during preprocessing. These parameters are chosen based on the historical query request patterns for the training data, balancing the impact of dropped requests / model accuracy and performance costs. The queries issued to the hot table benefit from the lower PIR cost for accessing the small table rather than a large full table. We emphasize that this design is necessary to eliminate data leakage through the number of queries that a user issues to each table. For example, the number of queries to the hot table can reveal whether the user accesses the indices that are in the hot table. The total number of table entries that a user accesses in both hot and full tables may also leak private information such as the number of items purchased, the number of websites visited, etc. To remove such information leakage through the number of accesses to each table, for each inference, we require a user to issue exactly Q_{hot} and Q_{full} queries to the tables. If the user needs to read more table entries than the allocated budget, these requests are dropped; the dropped requests may impact model accuracy. If the user has fewer queries, then dummy queries are added to ensure that the user makes the fixed number of PIR requests.

Access Pattern-Aware Embedding Co-location Embedding table access patterns in ML applications tend to exhibit co-occurrence [142,50] as some indices are often accessed together in a single ML inference. We co-locate the entries that are frequently accessed together in the same row of

the table so that a single query can retrieve multiple embeddings that might be accessed together (Figure 5.10c). Co-location is done by profiling the training dataset and co-locating the top-C embeddings that are most frequently retrieved with each embedding. C is empirically selected. In the best-case scenario, co-location can reduce the number of queries by $C + 1$.

Co-design Parameter Selection The parameters involving these two co-design techniques (frequency-based hot table splitting and embedding co-location), which involve parameters such as Q_{hot}, Q_{full}, C, and bin-size, as well as kernel parameters such as DPF execution batch size and DPF execution strategy are selected after sweeping the parameter space using grid search and evaluating the corresponding performance (i.e: communication and computational costs, as well as accuracy) for the target application. Note we separate training and test datasets, selecting parameters based on the training dataset, and showing results on the test dataset. Broadly, our experimental results show the pareto frontier of the performance achieved across a complete sweep of the parameter space. Generally, across applications, we found that a good choice of Q_{hot} is typically 10%-20% of the size of the full embedding table. On the other hand, a good choice of C, the number of entries to collocate, depended on the application: a higher C at around 4-5 (i.e: more collocation) was more beneficial for the language model task, as words in a sentence have natural associations, whereas a lower C at 1-3 was better for the recommendation application. A good choice of bin size and other parameters such as DPF execution batch size and strategy, generally vary and depend on performance or accuracy constraints which may be imposed by service expectations. In summary, our co-design and kernel parameters are determined by performing a grid search across the space of possible parameters in order to find parameters that balance computation, communication and model accuracy.

Changes to Embedding Table Updates to the embedding table (i.e., updates/insertions/deletions) may occur over time as embedding tables can change when the model is re-trained. Note that updates to table entries without changing indexing (no insertion/deletion) can be done under the hood (transparent to the clients) by updating the table entries on both servers. From the client perspec-

tive, the tables are read-only. Full updates of embedding tables that include deletions and insertions, on the other hand, require the indexing functions on the client to be also updated. An updated hash table for the hot table needs to be sent to the client. If the full table size is changed, the hash function for indexing the full embedding table is also updated on the client. However, this cost of a full update is only incurred when the model itself is changed or fully re-trained, which is infrequent for typical recommendation models or language models. In this paper, we study the overhead of our system assuming that full embedding table updates are infrequent enough. More efficient handling of table updates for other use cases that require frequent updates is left as future work.

5.6 Evaluation

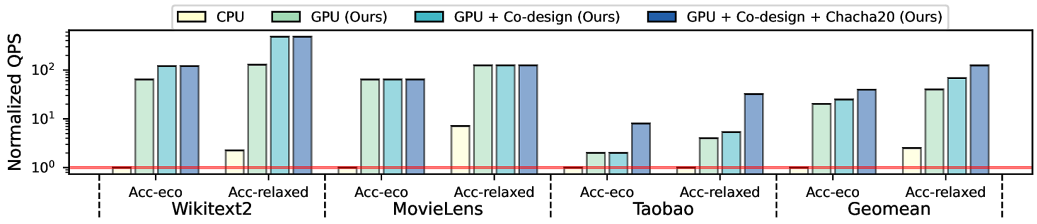


Figure 5.11: Throughput improvement of our proposed system over the CPU baseline[85]. While preserving accuracy (Acc-eco), our system can improve the throughput on average by 5–39×. When some amount of accuracy degradation is tolerated (Acc-relaxed), the average improvement reaches 40–124×. All configurations searched within the latency ($<$ 300ms) and communication requirement ($<$ 300KB). QPS normalized by the CPU Acc-eco for each benchmark.

5.6.1 Evaluation Setup

Platforms We evaluate our GPU-based DPF-PIR and compare it with a state-of-the-art CPU implementation[85]. We run all GPU experiments on an NVIDIA V100 GPU, and all CPU experiments on an Intel(R) Xeon(R) Gold 6230 CPU @ 2.10GHz with 28 cores. The CPU baseline is an optimized DPF-PIR implementation from Google Research[85], which uses AES-NI CPU hardware acceleration.

Datasets and Models We evaluate our system and the baseline by running a couple of recommendation models and a language model on open-source datasets. We run (1) a 2-layer MLP-based recommendation model [101] with MovieLens-20M dataset [100], (2) a 2-layer MLP-based recommendation model [101] with Taobao Ads click/display dataset [217], and (3) an LSTM model with Wikitext2 corpus [163]. We protect the user history table [242] for recommendation models and the word embedding for the LSTM using PIR. The baseline model quality of the models we study are as follows. For recommendation models, we use *area under the receiver operating characteristic curve* (ROC-AUC or AUC) metric, where a higher AUC means better quality. Our model achieves AUC=0.7845 for MovieLens and AUC=0.58 for Taobao, similar to prior works [242,101]. For LSTM, we use perplexity (ppl), a measure of surprise, to measure the model quality. Following the training setup of [163], our model achieves ppl=92.

System Parameters For application-independent experiments (Figures 5.13–5.15, Tables 5.4–5.5), unless otherwise stated, we default to an entry size of 2048 bits. Most recommendation models use entries similar or smaller than this [242,179]. Also, by default, we use a security parameter of 128 bits as standard (AES-128), and apply all proposed GPU acceleration optimizations, with a memory optimization factor $K = 128$. Batch size is tuned for each experiment separately to maximize throughput while meeting latency and communication budgets (300ms and 300KB, unless stated otherwise).

5.6.2 End-to-End System Throughput on Applications

First, we show that our proposed design *significantly improves system throughput on various applications*, compared to the baseline CPU system [85]. We evaluate key portions of our proposed design separately: 1) Applying all GPU acceleration techniques (**GPU (Ours)**), 2) Adding ML co-design (**GPU + Co-design (Ours)**), and 3) Using Chacha20 instead of AES-128 (**GPU + Co-design +**

Table 5.3: Unnormalized QPS from Figure 5.11. Among our proposed design, we only show the best one (GPU + Co-design + Chacha20). Acc-eco specifies that each approach must reach the full-precision accuracy; Acc-relaxed indicates the approaches must reach within some range of full precision accuracy; see Section 5.6.2

Dataset	CPU	Ours	
		Acc-eco	Acc-relaxed
Wikitext2	5	577	2,306
MovieLens	44	2,821	5,476
Taobao	8k	64k	256k

Chacha20 (Ours)). For each design, we conducted an extensive parameter sweep across kernel hyperparameters like batch size and K, and across co-design hyperparameters like hot table and cold table sizes, the number of entries co-located, and the number of queries issued to each table. We first show throughput achieved requiring a fixed model quality. Then, we additionally show throughput improvement tolerating some model quality degradation. We set the tolerated degradation to $<$0.5% for MovieLens and Taobao and $<$5% for Wikitext2.

Figure 5.11 shows that the throughput improves by **5–39**$\times$ while maintaining the model quality (Acc-eco), and the improvement becomes **40–124**$\times$ when small quality degradation is tolerated (Acc-relaxed). GPU optimizations account for 10–20$\times$ performance improvement, and PIR-ML co-design can additionally obtain up to 2–5$\times$ improvement. These cumulative improvements result in significant overall gains. Co-design does not show improvement for MovieLens for this particular setup; however, the co-design is more effective for the cases with a tighter communication budget. We discuss this later in Figure 5.19.

Table 5.3 additionally shows the unnormalized numbers for some representative points. Our proposed design improves performance from an impractical throughput (e.g., 5 QPS) to an acceptable range of hundreds of QPS. Taobao has much higher QPS in general, because each user queries much fewer entries per inference (2.68 on average), compared to other benchmarks (e.g., MovieLens queries 72 entries per inference on average).

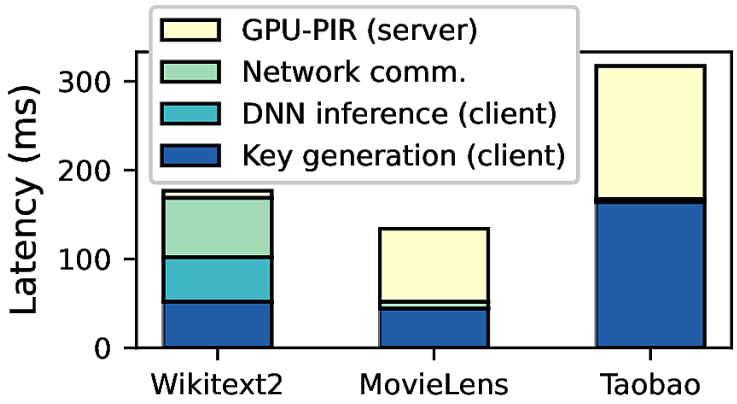


Figure 5.12: End-to-end latency breakdown of an inference query (i.e: time from client request to receiving and computing the result). Our proposed system makes the PIR latency much lower (Wikitext2) or comparable (MovieLens, Taobao) to the latency of other components. We are able to keep end-to-end latency within a reasonable 500 ms per inference which is acceptable in standard SLAs[93]

5.6.3 End-to-End System Latency

We subsequently show the impact of our system on end-to-end inference latency to show that the latency overhead of our GPU-PIR results in acceptable standards for real-world applications. Four components that affect inference latency include: (1) client-side key generation (*Gen*), (2) PIR (*Eval*; our paper's main focus), (3) client-server network communication (4) client on-device DNN inference. We measure the latency of key generation and DNN inference directly on a single Intel Core i3 CPU. We estimate the network latency assuming 60 Mbit/s bandwidth as in 4G networks[3]. Figure 5.12 shows that PIR is not the sole dominating latency bottleneck anymore, costing comparable or less latency compared to other sources. While the overall end-to-end latency is much larger than a no-privacy system, the end-to-end latency still falls under the typical service level requirement (SLA) of many real-world applications[93].

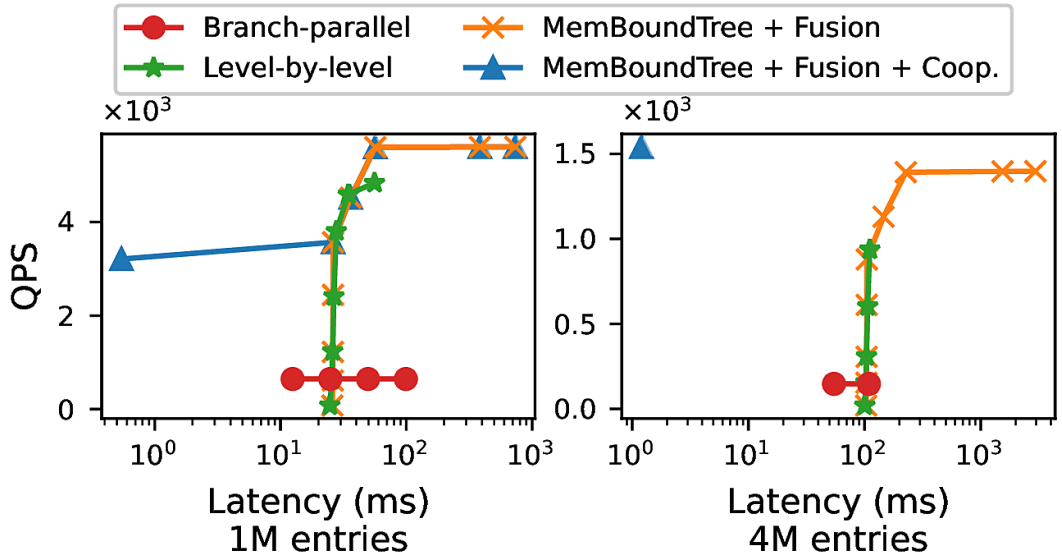


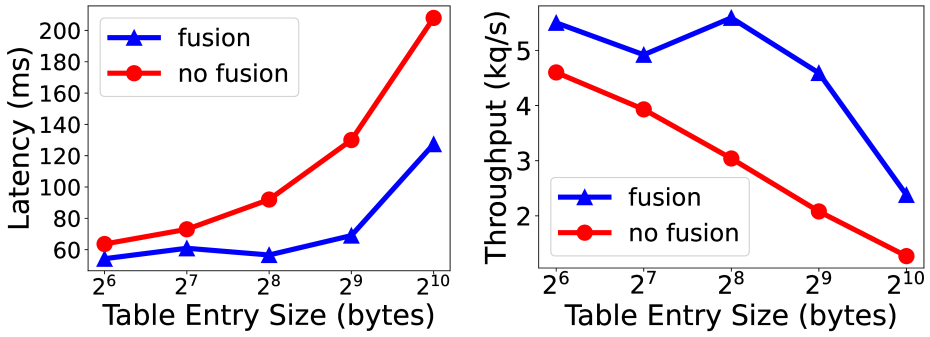
Figure 5.13: Throughput vs latency for different GPU optimizations: branch-parallel (red), level-by-level (green), memory-bounded tree traversal and operator fusion (orange), and batch/table-size aware scheduling with cooperative groups (blue).

5.6.4 Detailed Analysis of System Optimizations

Here, we evaluate and isolate the effects of our proposed system optimizations, starting with GPU kernel optimizations, and concluding with ML co-design optimizations.

Performance Impact of Each GPU Optimization Figure 5.13 plots the latency-throughput tradeoff for each GPU optimization. As shown, our proposed optimizations increase the latency-throughput pareto frontier significantly. As discussed in Section 5.4.2, branch-parallel (red) cannot achieve high QPS. Level-by-level (green) is much better, but still limited, as it is bottlenecked by the memory capacity. The proposed memory-bounded tree traversal and operator fusion (orange) is able to increase the throughput further when some latency degradation is tolerated, by using less memory and allowing additional batching. For very large tables (Figure 5.13 (right)), table-size aware scheduling with cooperative groups (blue) obtains significantly better latency without harming throughput.

Performance Impact of Operator Fusion Figure 5.14 shows the performance benefits of fusing the subsequent matrix multiplication with DPF evaluation, across different table entry sizes. Generally, fusing and interleaving the two kernels offer significant ($> 1.5\times$) improvements in both throughput and latency. Figure 5.14 was obtained with a table with 1M entries; however, the improvement is similar across other table sizes.



(a) Latency **(b)** Throughput

Figure 5.14: Performance impact of table entry size on PIR performance, with and without operator fusion.

Performance Impact of Embedding Entry Size Figure 5.14 also shows the impact of different table entry sizes on latency and throughput. Tables with entry sizes of <512 bytes do not significantly degrade performance, especially with operator fusion. This is because the memory operations are tightly interwoven with the subsequent matrix operations with operator fusion. As the latency and throughput does not linearly degrade with increasing entry size, co-locating and retrieving multiple entries at once becomes efficient (Section 5.5.2).

Detailed Comparison with CPU We compare our GPU-PIR implementation against an optimized CPU implementation from Google Research [85]. Note that, Google Research's CPU implementation of DPFs uses AES-128 for its PRF, and utilizes AES-NI hardware intrinsics to accelerate PRF computation. Figure 5.15 compares the throughput attained by the memory-efficient GPU DPF acceleration strategy against a 1-threaded and 32-threaded (fully-utilized) CPU version on

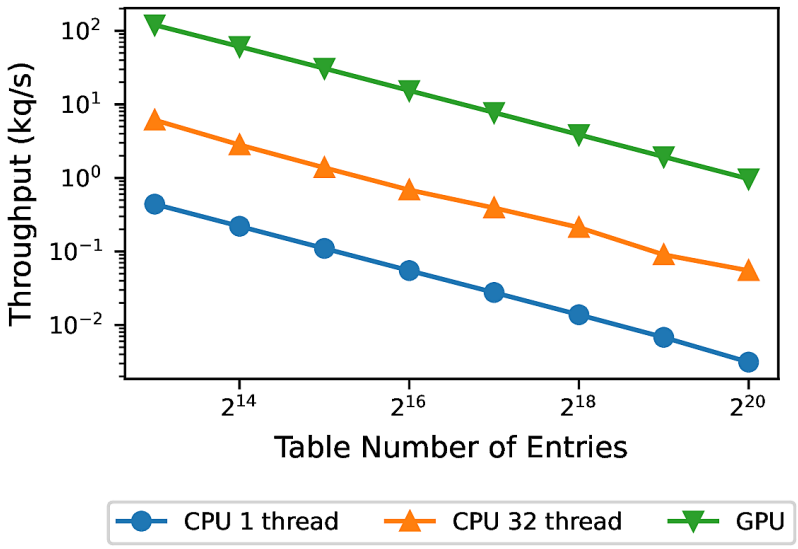


Figure 5.15: Comparison of throughput performance attained by GPU DPF acceleration compared to an optimized CPU baseline. 1 kq/s = 1,000 queries per second. All methods use the AES-128 PRF.

different table sizes. Using AES-128 as in the CPU DPF, our GPU implementation consistently achieves $>$ 17$\times$ speedup over the 32-threaded CPU implementation. We show the same data in Table 5.4.

Performance Impact of PRF Table 5.5 shows the performance of using different PRF functions on a table with 1M entries, a batch size of 512, and a security parameter of 128-bits. Lightweight PRFs can significantly improve the GPU-PIR performance over AES-128. In particular, Chacha20, a well-accepted PRF that is used in high-security applications including TLS 1.3 [37], improves the latency and throughput significantly compared to AES-128. Other lightweight PRFs can improve the throughput even more if their security is acceptable for the target use case.

5.6.5 PIR + ML Co-Design

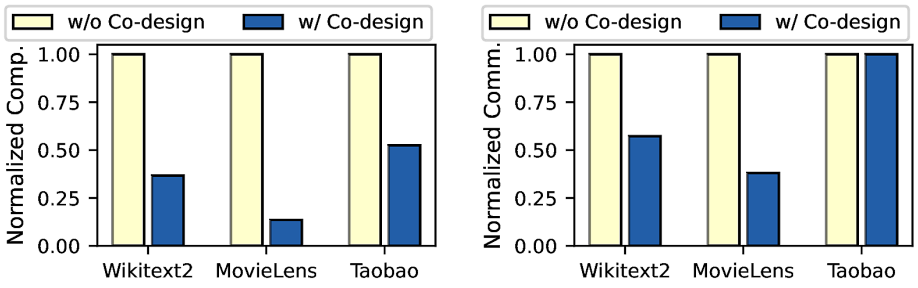
Private on-device ML inference often requires the private retrieval of a batch of embeddings from the same table. We evaluate our techniques that co-design ML inference and batch PIR, and demon-

Table 5.4: Throughput / latency comparison of our GPU acceleration (all optimizations) vs single and multi-threaded CPU implementations, on tables with an entry size of 2048 bits. Both use AES-128 as their PRF. The CPU DPF baseline is taken from [85] and is an optimized CPU implementation that uses AES-NI hardware intrinsics. Bytes indicates the size of the DPF key that is transferred between client and server for that table size.

# Entries	Bytes	Strategy	QPS	Latency (ms)
16K	896	GPU	60,347	3.2
		CPU 1-thread	22	9
		CPU 32-thread	2,810	.71
1M	1280	GPU	1,358	1.4
		CPU 1-thread	1.3	638
		CPU 32-thread	21.2	36
4M	1408	GPU	468	4.18
		CPU 1-thread	0.78	2579.8
		CPU 32-thread	12	160.1

Table 5.5: Performance evaluation of memory-efficient GPU DPF with different PRF functions, on a table of size 1,048,576, with batch size 512, and a security parameter of 128 bits.

PRF	Type	Latency (ms)	QPS
AES-128	Block Cipher (Ctr Mode)	591	965
SHA-256	Hash (HMAC)	659	921
Chacha20	Stream Cipher	174	3,640
SipHash	PRF	82.3	7,447
HighwayHash	PRF	320	1,973



(a) Computation overhead

(b) Communication overhead

Figure 5.16: Computation (a) and communication (b) needed to achieve a target model accuracy (Acc-relaxed from Figure 5.11), with and without ML co-design. Co-design improves computation overhead by 1.9–7.4× and communication overhead by 1–2.6×.

strate how our co-design techniques significantly improve model quality vs system performance tradeoffs.

Computation Savings Figure 5.16a shows the computation needed to reach a target accuracy with and without ML co-design. We fixed the communication below 300KB, and target Acc-relaxed from Figure 5.11. Figure 5.16a shows that co-design reduces the computation significantly, by 1.9×–7.4×.

Communication Savings Figure 5.16b shows the communication needed to reach a target accuracy (Acc-relaxed) with and without ML co-design. We fixed the computation to be less than 100K PRFs per batched inference for Wikitext2 and MovieLens, and 5M PRFs for Taobao. With a fixed computation budget, the result shows that co-design improves the communication overhead by 1.7× and 2.6× for Wikitext2 and MovieLens, respectively. Taobao's communication overhead was already too small (<3KB) and did not improve. Co-design can be especially useful when the communication is expensive, e.g., when using 3G/4G network.

Communication vs Computation We show the tradeoff between computation and communication with the fixed model quality. Figure 5.17 shows this tradeoff across various applications, with model quality fixed to be within 2% of the full precision baseline. Co-design optimizations obtain

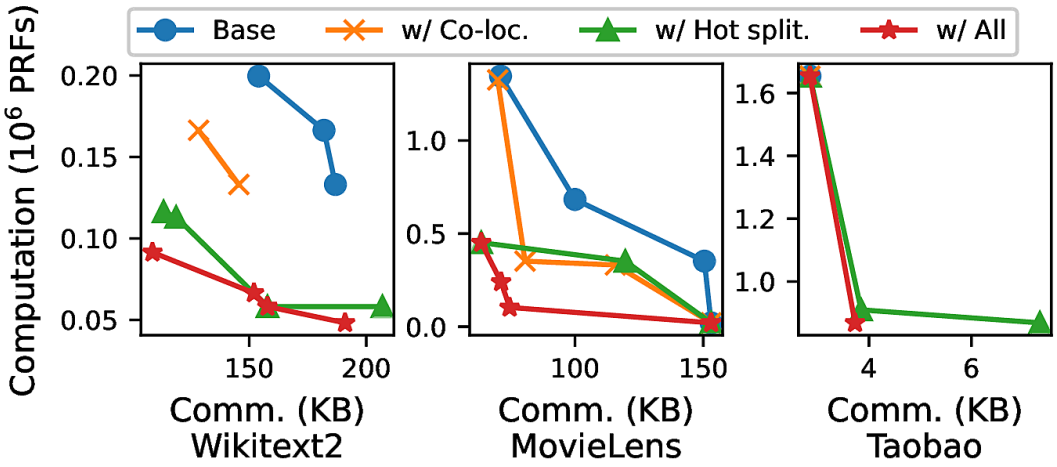


Figure 5.17: Pareto curve of tradeoff between computation and communication with model accuracy fixed to be within 2% of the baseline.

significantly better tradeoffs than plain batch-PIR.

Co-Design Throughput Improvement We show overall co-design throughput improvement over standard batch-PIR across all applications on select budgets in Figures 5.18, 5.19, and 5.20. As shown, the PIR-ML co-design can result in significant improvements to the tradeoffs between model-quality and system throughput. Co-design is most effective when a) the budget is small enough to be sufficiently restrictive, and b) the impact of dropping queries has a significant impact on model quality. To expand on a), the budget plays a major role in the relative improvement that co-design sees as shown in Figures 5.18 and 5.19; there is increasingly smaller difference between batch-PIR and batch-PIR with co-design when the budgets are large enough. This makes intuitive sense as with a larger budget both batch-PIR schemes with and without co-design converge on the optimal pareto curve. Expanding on b), co-design is less helpful for applications where dropping the sparse features does not impact model quality – this is natural since co-design optimizes for model quality and if the sparse features has less impact, the relative gains of co-design would also be less. This phenomenon is best demonstrated by the observation that language model (Figure 5.18) and

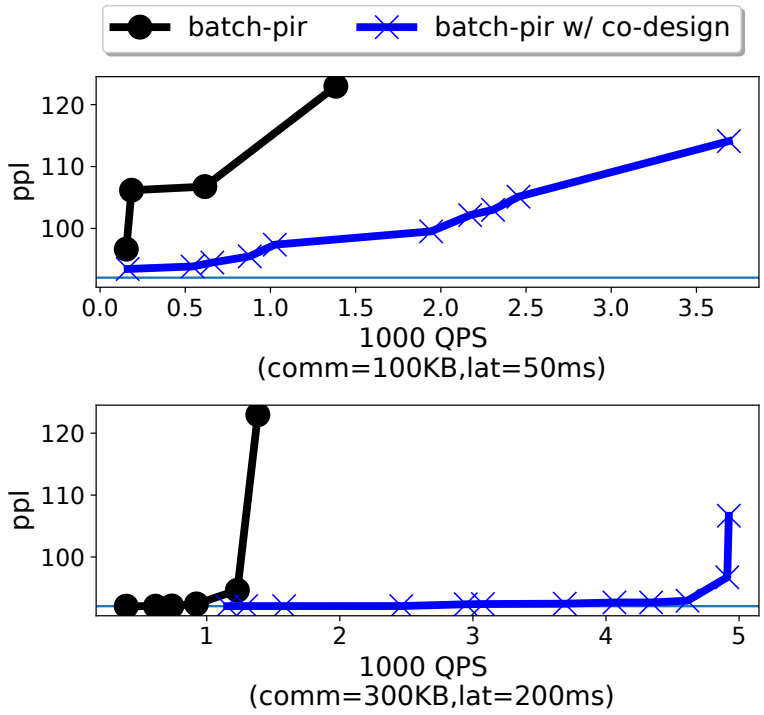


Figure 5.18: System throughput vs model quality with and without co-design for language model across different budgets.

MovieLens (Figure 5.19), whose model inputs are entirely sparse features that require embedding table lookups, see much greater improvement with co-design compared to Taobao (Figure 5.20), whose sparse categorical features are only a fraction of model inputs. Overall, the results show that PIR-ML co-design can significantly improve the system throughput beyond what just batch-PIR can support, especially under tight computation and/or communication budgets.

5.7 Conclusion

We present a system for efficiently and privately serving embeddings for on-device ML application. Our system on a single V100 GPU can serve up to 100,000 queries per second—a $>100\times$ speedup over naive system, enabling practical deployment for privacy-sensitive applications.

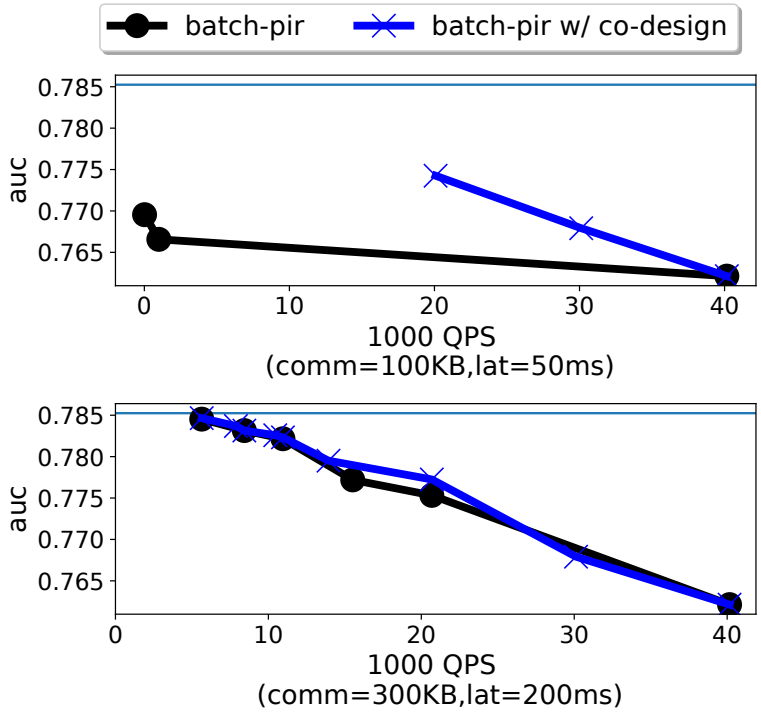


Figure 5.19: System throughput vs model quality with and without co-design for MovieLens rec across different budgets.

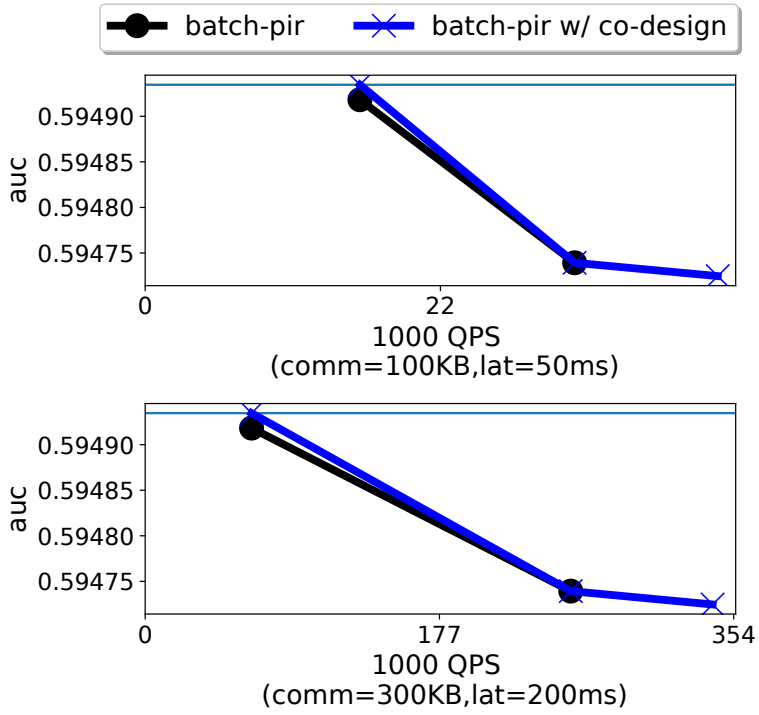


Figure 5.20: System throughput vs model quality with and without co-design for Taobao rec across different budgets.

6

Breaking Privacy in Federated Learning by Reconstructing the User Participant Matrix

In this final chapter we show how ensuring the security of the machine learning system, namely in federated learning, is essential for ensuring data privacy. We develop an attack on federated learning that recovers individual gradient updates from sums, which may lead to the full recovery of the user's input data breaking privacy. This work shows that the security and privacy of a machine

learning systems must be thought about from both a defensive and offensive angle and motivates new privacy defenses in the federated learning paradigm.

6.1 Introduction

Federated learning is a method for collaboratively learning a shared model across multiple participants and enhances privacy by limiting data sharing [155,99,27,135,134]. Participants' data privacy is preserved by sending model updates rather than raw data, which limits the amount of information that is exposed to the central server. In the context of applications, federated learning participants are edge devices such as users' smart phones or wearables, and maintaining the integrity of their data is a critical issue. Already, federated learning has been deployed by many major companies in various privacy sensitive applications including sentiment learning, next word prediction, health monitoring, content suggestion, and item ranking [99,145,27]. Guaranteeing data privacy in these scenarios is becoming increasingly important as the topic of privacy becomes more heavily scrutinized by the greater public and by government regulations [155,153,68].

Recent research has shown that model updates may unintentionally leak information about their respective training examples [71,158,245]. A central server that obtains participants' model updates may perform inference attacks to learn significant information about participants' training data, violating the core privacy principles of the federated learning paradigm. To address this critical privacy flaw, researchers have introduced methods leveraging secure multiparty computation to limit the central server's visibility into individual participants' model updates. Notably, secure aggregation [200,208] has emerged as a standard security protocol which ensures that the central server may see only the final sum of model updates, rather than any individual update by itself. Thus, information learned from the aggregated model update may not be attributed to a specific user, which offers a layer of privacy against the central server. Additionally, by aggregating updates over tens to hundreds or

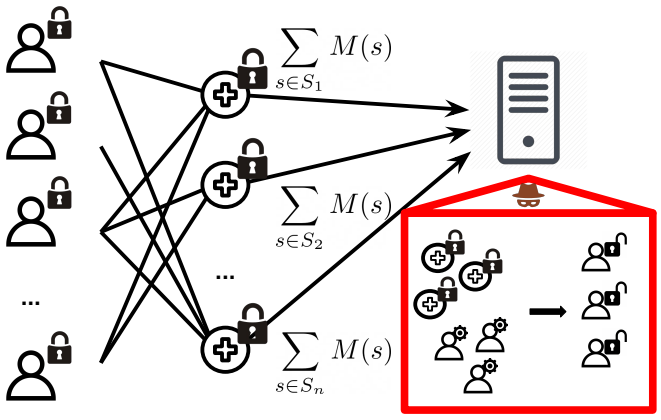


Figure 6.1: Our gradient disaggregation attack observes multiple rounds of aggregated model updates and leverages side channel information in the form of summary analytics collected by federated learning systems (how often users participated across certain training rounds) to uncover individual users' private model updates, undermining the secure aggregation protocol. Code: https://github.com/gdisag/gradient_disaggregation.

thousands of users, updates are obfuscated to a point where most inference attacks are rendered ineffectual [158,71,245].

The secure aggregation protocol is secure only to the degree that it hides individual participants' model updates. A procedure that disaggregates individual participants' updates or gradients from their sum would undermine the secure aggregation protocol and unveil the aforementioned privacy vulnerability. In this work, we develop a method for gradient disaggregation, showing that secure aggregation offers little privacy protection against an adversarial server seeking to undermine individual users' data privacy. Our key insight is that participant information (e.g: which rounds of training users participated in) is derivable from aggregated model updates, when observing multiple rounds of training and leveraging summary analytics. We can reconstruct this information and use it to recover participants' individual model updates (see Figure 6.1). Our contributions are as follows:

- We introduce and formulate the gradient disaggregation problem as a constrained binary matrix factorization problem. Leveraging summary analytics collected by federated learning systems, we demonstrate that our disaggregation attack can exactly recover the user partic-

ipant matrix on up to thousands of participants, revealing the model update of each user. Additionally, we show that gradient disaggregation works even in the presence of significant noise and allows us to disaggregate aggregated model updates that were generated by federated averaging.

- We leverage gradient disaggregation to significantly improve the quality of traditional inference attacks on model updates. We show that without gradient disaggregation, inference attacks often fail to recover meaningful information on updates aggregated across tens to hundreds of users; with gradient disaggregation, we show successful recovery of users' privileged data from their disaggregated model updates.

6.2 Related Work

6.2.1 Secure Aggregation

Secure aggregation is a method based on secure multiparty computation and is a key privacy measure deployed in federated learning systems. Secure aggregation ensures that the central server sees only the final aggregate of model updates across users while guaranteeing that no participants' updates are revealed in the clear[200]. The secure aggregation protocol enhances privacy by obfuscating a user's model update with many other users' updates, limiting inference attacks such as those in [158,71,245]. This obfuscation also ensures that information learned from the aggregated model update may not be attributed to an individual user. Concretely, on the issue of attribution, the secure aggregation paper states: "Using a Secure Aggregation protocol to compute these weighted averages would ensure that the server may learn only that one or more users in this randomly selected subset wrote a given word, but not which users"[200].

Our work on gradient disaggregation undermines the secure aggregation protocol by showing that, through observing multiple rounds of collected data and leveraging side channel informa-

tion (specifically, user participation frequency as collected by federated learning systems), individual updates may be reconstructed from their overall sums. While secure aggregation has been proven to be cryptographically secure, leaking no information which is not leaked by the aggregated model update itself[200], the key insight of our attack is that participant information (e.g: which rounds each user participated in) is derivable from the aggregated model updates and reconstructing it allows us to in turn recover individual model updates.

6.2.2 Analytics in Federated Learning Systems

Infrastructure to support, debug, and manage federated learning systems is critical to their functioning. [27] outlines the design of Google's federated learning systems and describes its core components and protocols. A key aspect of their infrastructure is the collection of device analytics. Notably, [27] collect several important device metrics such as how often devices performed training, how much memory devices used during training, etc [27]. These metrics ensure that users' devices are not oversubscribed (draining battery) and may be used to debug device performance issues. Device analytics play a critical role in maintaining user experience quality: "Device utility to the user is mission critical, and degradations are difficult to pinpoint and easy to wrongly diagnose. Using accurate analytics to prevent federated training from negatively impacting the device's utility to the user accounts for a substantial part of our engineering and risk mitigation costs." [27]

In our work, we leverage summary information from device analytics – specifically how often a user performed training – to assist disaggregating gradients, breaking privacy. Note while [27] points out that device analytics contain no personally identifiable information, these reports nevertheless provide crucial information that links gradient information collected across rounds, facilitating our attack on disaggregating gradients.

6.2.3 Product Identification by Solving Linear Inverse Problem

Recently, independent of our work, research has shown that individual item product prices may be recovered given customer's transaction history by optimizing a linear inverse problem[65]. Under certain conditions (e.g: assuming that in the transaction history each item was purchased by itself at least once) their approach recovers these item prices with high precision and allows them to reveal customers' spending habits. Specifically, given a corpus of sums of item prices from customers' transaction histories,[65] utilizes a subset sum algorithm to uncover the individual prices of the transaction and to identify the products themselves.

Our work on gradient disaggregation and the work in[65] solve the same core problem: uncovering individual values given observations of their sums. While their work recovers prices of items, our work analogically reconstructs participants' model gradients. However, a key distinction in[65] is the assumption that each item must be purchased individually at least once. This makes their approach unsuitable for disaggregating aggregated model updates as, under the secure aggregation protocol, each aggregated update is composed of more than one participant's model updates.

6.2.4 Data Leakage from Model Updates

Recent research has shown that model updates and gradients leak significant amounts of information. Information leaked by model updates ranges from specific properties to entire data samples[158,245,71,206,186,230,152,16,159,104]. Methods to recover this information from gradients are broadly categorized as inference attacks, and prior works have demonstrated the effectiveness of inference attacks on small batches of gradients, across various modalities ranging from image to text[206], on both shallow and deep networks[71].

In the context of federated learning, these methods suffer decreased efficacy with larger aggregates (> 100)[158,245,71]. Our work on gradient disaggregation facilitates these attacks by de-obfuscating

these updates and by enabling attribution of learned properties to specific users.

6.2.5 Privacy Attacks in Federated Learning

Recent works have introduced various privacy attacks on federated learning. Broadly, these attacks are performed by a malicious central server or by participants with influence over model training[152]. Threats from an adversarial central server typically involve extracting private information via inference attacks as described in the previous section. Attacks by adversarial participants, on the other hand, involve influencing the model training process to alter the behavior of the trained model (e.g: model poisoning, backdoors)[225,17,69,25].

Our work on gradient disaggregation falls under the category of an attack performed by a malicious central server. Specifically, gradient disaggregation breaks the secure aggregation protocol and enables a central server to perform inference attacks on individual participants' model updates.

6.3 Gradient Disaggregation

6.3.1 Problem Statement, Threat Model and Assumptions

Gradient disaggregation involves uncovering individual participants' model updates given observations of their sums. Concretely, on round r the central server receives

$$G_{aggregated_{[r,:]}} = \sum_{s \in S_r} M(s)$$

where S_r is the selected participants on round r, and $M(s)$ are the model updates. The goal of gradient disaggregation is, acting as an adversarial central server, to recover $M(s)$ given $G_{aggregated}$ (aggregated gradients across n rounds).

Our threat model and assumptions are as follows:

- The central server is adversarial but is limited in its ability to modify the training protocol. Specifically, we assume the central server may fix its model across rounds. Such a scenario is realistic in a case where an attacker has read access to corporation servers (e.g: to collect round model update data) and limited influence over when the global model is updated (e.g: to fix the model across rounds). An adversarial central server is a major threat model in federated learning [152,145,126]

- Client selection / device participation (S_r) is somewhat random and is a subset of the total number of users. This matches the federated learning protocol which selects a random fraction of devices to participate in each round of training [27,155,145].

- The central server has access to side channel information in the form of summary analytics (specifically, how often users participated across certain federated learning rounds). Device and summary analytics are a core part of federated learning systems and infrastructure [27].

6.3.2 Gradient Disaggregation by Reconstructing the User Participant Matrix

A central server that observes aggregates of users' updates that are constant across rounds obtains

$$G_{aggregated} = PG_{individual} \tag{6.1}$$

where $G_{aggregated} \in \mathbb{R}^{n \times d}$ are the final aggregated dimension d gradients the server collected across n rounds; $P \in \{0,1\}^{n \times u}$ is the user participant matrix across the n rounds with u total participants specifying which users participated in which rounds; and $G_{individual} \in \mathbb{R}^{u \times d}$ contains per user individual gradients. Hence, recovering $G_{individual}$ may be viewed as a matrix factorization problem where the left term is binary.

To approach this matrix factorization problem, we start with the method introduced in [207], which, to the best of our knowledge, is one of the only works to address matrix factorization where the left term is binary. [207] first reconstructs the binary user participant matrix P, then recovers $G_{individual}$ by inverting P from $G_{aggregated}$. As observed by [207], columns of P lie in the image of $G_{aggregated}$. Hence, with $Nul(M)$ as the kernel of a matrix, an approach to solving this factorization problem would be to recover each column p_k of P:

$$\begin{aligned} &\text{Find } p_k \text{ s.t } Nul(G_{aggregated}^T)p_k = 0 \\ &p_k \in \{0,1\}^n \end{aligned} \tag{6.2}$$

In the context of federated learning, this attempts to recover individually for each user which rounds they participated in. Note that such an optimization procedure can be solved using standard mixed-integer programming frameworks such as [94] and can additionally be parallelized across each user. However, this approach is not sufficient for gradient disaggregation due to three issues: 1) failure to distinguish between the numerous binary vectors in the image of $G_{aggregated}$, 2) inability to distinguish between user solutions and 3) computational difficulties due to the exponential nature of the optimization problem (recovering p_k is NP hard and [207] reports only being able to solve up to $n = 30$ vectors). To address these issues, we incorporate summary analytics to assist factorization.

6.3.3 Leveraging Summary Analytics to Reconstruct P

We leverage summary information from device analytics as collected in [27] to assist reconstructing P. Specifically, summary analytics that are collected periodically by the central server log how often a specific user participated in training and can be used to narrow down p_k by limiting the total number of participations across certain training rounds (see our Related Works section for details). We capture partial information on participations across rounds by introducing linear constraints: the

i'th constraint $C_k^i \in \{0,1\}^n$ specifies for the k'th participant the training rounds for which total number of participations c_k^i is known. For example, knowing that a user participated in training 3 times between rounds 1-5 and 2 times between rounds 6-10 yields $C_1^1 = [1,1,1,1,1,0,0,0,0,0]$, $c_1^1 = 3$, $C_1^2 = [0,0,0,0,0,1,1,1,1,1]$, $c_1^2 = 2$.

We therefore add the individual constraints.

$$C_k^i p_k - c_k^i = 0 \tag{6.3}$$

After collecting all j constraints and counts across all users, we combine them into

$$C_k = \begin{bmatrix} — & C_k^1 & — \\ & \vdots & \\ — & C_k^j & — \end{bmatrix}, c_k = \begin{bmatrix} c_k^1 \\ \vdots \\ c_k^j \end{bmatrix} \tag{6.4}$$

Incorporating them into the optimization, we obtain

$$\begin{aligned} &\text{Find } p_k \text{ s.t. } Nul(G_{aggregated}^T) p_k = 0 \\ &p_k \in \{0,1\}^n \\ &C_k p_k - c_k = 0 \end{aligned} \tag{6.5}$$

We note that it is possible that devices timestamp the exact moment they perform a round of training; in this case, P may be revealed directly through the specificity of the constraints (making the disaggregation problem solvable through a simple linear regression). However, even if devices log only the total number of times they performed training (with no timestamped data) and send these analytics back to the server once every few rounds of participation, the central server may piece together these constraints and incorporate them into the formulation above. In other words, just knowing

the number of times particular users performed training and collecting this information periodically (both of which are reasonable based on [27]), the central server may obtain enough information to carry out the gradient disaggregation attack. Incorporating summary analytics into the gradient disaggregation attack is significant as it greatly reduces the problem space, allowing a solution to a previously intractable problem.

6.3.4 Disaggregating Noisy Model Updates

Previously, we assumed users submitted the same model update across every round. However, participants may perform updates composed of multiple steps (e.g: FedAvg) or their data may change, leading to differences in the updates they submit across rounds. We treat these differences as a form of injected noise.

Accounting for noise, our formulation becomes

$$G_{aggregated} = PG_{individual\_avg} + noise \tag{6.6}$$

and our goal is to recover for each user the average model update they submitted across rounds $G_{individual\_avg}$. We introduce two changes to reconstruct P in the presence of noise: 1) we use hard-threshold SVD with u singular values to approximate the low rank product $PG_{individual\_avg}$ and 2) we relax our constraint satisfaction problem to minimize the distance of the user participant column to the image of $G_{aggregated}$:

$$\begin{aligned} &\min ||Nul(G_{aggregated}^T)p_k||^2 \\ &p_k \in \{0,1\}^n \\ &C_k p_k - c_k = 0 \end{aligned} \tag{6.7}$$

These two changes allow reconstructing P even when the updates user submit across rounds are

noisy.

Note that we may have incomplete information for each user; for example, we may have constraints for a user over certain rounds but not others if the infrastructure only provides that information sporadically (or hides it). Additionally, if round participations are inexact (e.g: off by some small error), we may relax the hard constraint $C_k p_k - c_k = 0$ to be a soft constraint: $min||C_k p_k - c_k||^2$ and reweight the objective accordingly. Additionally, we can check whether our solution exactly recovers P by probing the number of optimal solutions returned by the mixed integer programming solver; if the solver returned only one optimal solution (and proved that it is the only one), then this indicates that our reconstruction of P is exact. Our full gradient disaggregation attack which works both for noisy and non-noisy updates is presented in Algorithm 4.

Algorithm 4: Gradient Disaggregation

Input: Aggregated gradients $G_{aggregated}$; constraint windows C; constraint sums c, number of users u
Output: Disaggregated gradients $G_{individual\_avg}$

$U, \Sigma, V \longleftarrow SVD(G_{aggregated})$
$G_{denoised} \longleftarrow U\Sigma[0:u]V$
for $i = 1$ **to** u **do**
  $p_i \longleftarrow \min ||Nul(G_{denoised}^T)p_i||^2$ s.t. $p_i \in \{0,1\}^n$ and $C_i p_i - c_i = 0$
end for

$P \longleftarrow [p_1, ..., p_u]$
return $LeastSquares(P, G_{aggregated})$

6.4 Results

6.4.1 Capabilities and Limitations of Disaggregation

We experimentally validate the capabilities and limitations of our gradient disaggregation procedure across various parameter settings. Note that unlike prior works performing server side attacks on privacy in federated learning, our method leverages participant information and rounds of aggregated gradients. Hence in our experiments we generate this information (across various settings) to understand how our attack behaves under different conditions. We evaluate the following parameters:

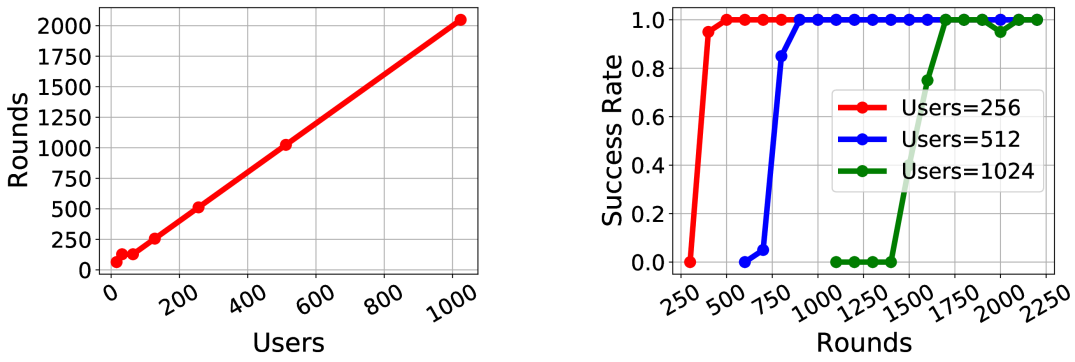
- **Number of Rounds:** Number of rounds of training n
- **Number of Users:** Number of users in system u.
- **Participation Rate:** Fraction of participants chosen to participate in each round.
- **Constraint Granularity:** Granularity of windows across rounds with known participation sums, per user. (E.g: granularity of 10 means we know how many times each user participated across every 10 rounds).
- **Gradient Noise:** Noise of user model updates across rounds.

We run all experiments on a 64-core cpu and use the Gurobi optimizer[94].

Number of Users

We validate the maximum number of users and rounds we can disaggregate on synthetically generated matrices. $G_{individual}$ is sampled from $\mathcal{N}(0,1)$, P is sampled with sparsity = participation rate = .1, and constraint granularity=10, with no noise between submitted gradients. For users $\in \{16, 32, 64, 128, 256, 512, 1024\}$ we scan over rounds $\in \{16, 32, 64, 128, 256, 512, 1024, 2048\}$ and report the minimum number of rounds to successfully disaggregate P with 100% accuracy over 30 trials.

Figure 6.2a shows the number of rounds required to exactly recover P across number of users; data shows that we can disaggregate matrices with thousands of user participants with enough observed gradients. Additionally, we plot success rate of reconstructing an individual column for users $\in \{256, 512, 1024\}$ which is shown in Figure 6.2b which furthermore reinforces that more rounds of observed gradients can increase reconstruction success rate. We also evaluate the relation that rounds vs users has on the runtime of the solver, shown in Figure 6.3, where we measure the runtime to exactly recover columns of P (with a maximum time limit of 180 seconds per column). Results show that larger P require more time to solve. Additionally fewer rounds leads to slower reconstruction as there are fewer constraints, while too many rounds leads to slower optimization due to large matrix sizes. Note we report time per column, as each column is solved in parallel.



(a) Rounds required to reconstruct P with 100% accuracy vs number of users.

(b) Success rate of recovering columns of P versus rounds.

Figure 6.2: Relationship between rounds vs number of users in gradient disaggregation. We successfully disaggregate settings with thousands of users with enough observed aggregated updates.

Participation Rate

We evaluate the effect of participation rate – the probability that a user is selected to take part in a round of training – on gradient disaggregation. We use the same parameter settings as in the previous section and scan participation rate $\in \{.10, .20, .30, .40, .50\}$ across various numbers of user

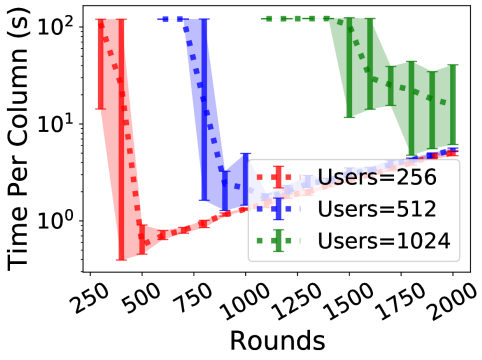
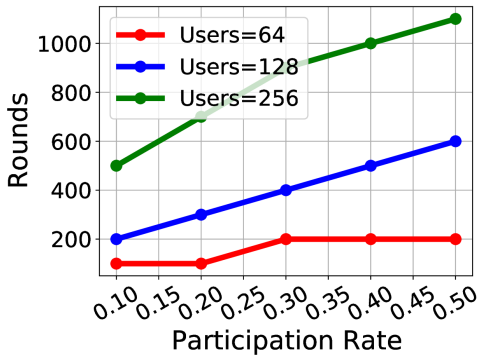


Figure 6.3: Mean, min, and max times to reconstruct columns of P vs rounds. More users require more time to recover P; too few rounds slows optimization due to lack of constraints; too many rounds slows optimization due to large vector sizes. We disaggregate thousands of users' gradients in minutes on a 64-core cpu.

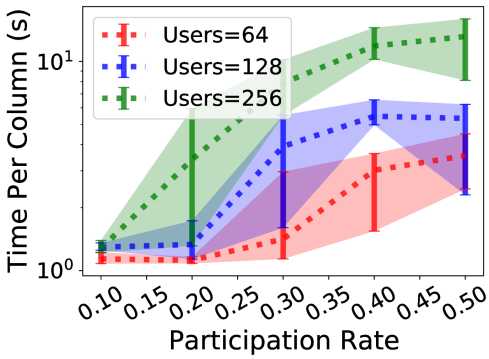
participants, measuring number of rounds of observed aggregated gradients required to successfully reconstruct P with 100% accuracy across 30 trials. As shown in Figure 6.4a, higher participation rate requires more rounds to reconstruct P. Intuitively, more participants per round leads to higher obfuscation of user updates, requiring more rounds to decode. However, as indicated, by observing more rounds of collected gradients, P is eventually reconstructed exactly. We additionally evaluate participation rate's effect on runtime which is shown in 6.4b. Higher participation rate makes the reconstruction problem more difficult and hence requires longer to solve. Note that federated learning settings have between tens to hundreds of round participants [145,27] and we have chosen these points to reflect this as accurately as possible.

Constraint Granularity

We evaluate the effect of constraint granularity on gradient disaggregation. We consider granularities $\in \{10, 20, 30, 40, 50\}$. Figure 6.5a shows that coarser constraints make reconstruction more difficult, requiring more rounds of observed aggregated gradients. Additionally, for reference Figure 6.5b shows the histogram of the number of times a user participates within a granularity window at different constraint granularities. Eventually, with enough observed rounds of aggregated gradients, the participant matrix P is exactly recoverable. Our results indicate that less detailed analytics may be



(a) Rounds required to reconstruct P with 100% accuracy vs participation rate.



(b) Mean, min, and max runtime to reconstruct columns of P vs participation rate.

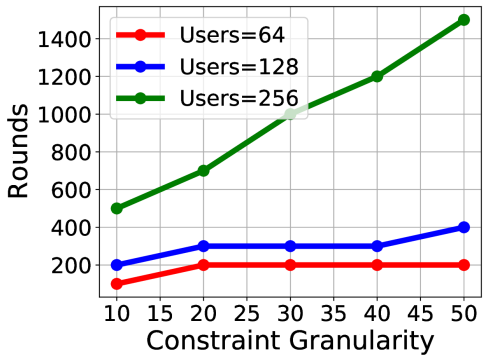
Figure 6.4: Effect of participation rate in gradient disaggregation. Higher participation rate can be compensated for by observing more rounds of aggregated gradients. We recover P even in the presence of many round participants.

compensated for by observing more rounds of aggregated model updates.

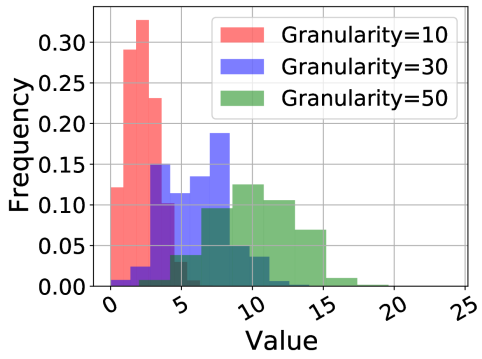
Noisy Model Updates / FedAvg

We address the scenario where model updates submitted by users are noisy across rounds, which may be due to the stochasticity of the optimization (e.g: the FedAvg algorithm). Initial experiments synthetically generate user ground truth gradients and inject noise into them at aggregation time. We initialize user vectors sampled from $\mathcal{N}(0,1)$ then inject noise sampled from $\mathcal{N}(0,\sigma)$ to each user's vector at aggregation time, measuring the gradient dimension required to exactly reconstruct P. We perform the experiment with 100 users, a participation rate of .1 and constraint granularity of 10, with a 600 second time limit on reconstructing each column of P.

Figure 6.6a shows the minimum gradient dimension (d) that is required to exactly reconstruct P with 100% success rate. Note that unlike prior experiments, increased noise may be compensated for by incorporating a higher number of the parameters of the model update (rather than observing more rounds of gradients). As even the smallest neural network models contain thousands or millions of parameters[96,97,107], this indicates that the attack may handle significant levels of noise.



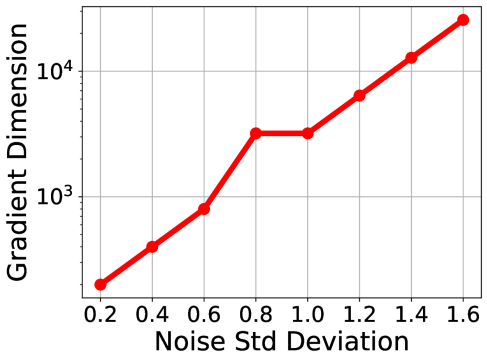
(a) Rounds required to reconstruct P with 100% accuracy vs constraint granularity.



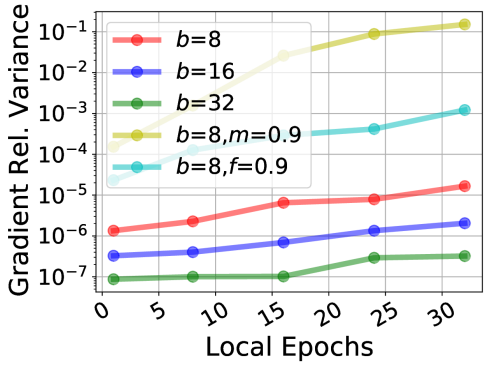
(b) Histogram of number of user participations across constraint windows.

Figure 6.5: Effect of constraint granularity in gradient disaggregation. Coarser constraints can be compensated by observing more rounds of aggregated gradients.

Furthermore, note that the dimension of the model update does not significantly affect solver time as the nullspace of $G_{aggregated}$ is computed only once and reused across users.



(a) Number of parameters of the gradient dimension required to recover P exactly.



(b) Relative noise of FedAvg updates on Cifar10 LeNet; batch size b, momentum m, dataset fraction f.

Figure 6.6: Effect of noise on gradient disaggregation.

Additionally, we perform experiments on gradient disaggregation using model updates generated by the FedAvg algorithm [156], on Cifar10 [138] with a LeNet neural network (SGD lr=.01). FedAvg performs multiple epochs of training over the participant's dataset before sending the final model

Dataset Size D	Batch Size b	Local Epochs e						
		1	2	4	8	16	32	64
64	8	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	16	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	32	1.0	1.0	1.0	1.0	1.0	1.0	1.0
128	8	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	16	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	32	1.0	1.0	1.0	1.0	1.0	1.0	1.0
64 (momentum=.9)	8	.99	1.0	1.0	1.0	1.0	1.0	1.0
	16	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	32	1.0	1.0	1.0	1.0	1.0	1.0	1.0
128 (momentum=.9)	8	1.0	1.0	1.0	1.0	1.0	1.0	.96
	16	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	32	1.0	1.0	1.0	1.0	1.0	1.0	1.0
64 (fraction=.9)	8	.06	.66	.97	.99	.98	.99	.85
	16	1.0	1.0	1.0	1.0	1.0	1.0	.99
	32	1.0	1.0	1.0	1.0	1.0	1.0	1.0
128 (fraction=.9)	8	1.0	1.0	1.0	1.0	1.0	1.0	.90
	16	.59	.96	1.0	1.0	1.0	1.0	.99
	32	1.0	1.0	1.0	1.0	1.0	1.0	1.0

Table 6.1: Fraction of P reconstructed with FedAvg model updates (users=100, rounds=200, Cifar10 LeNet, participant rate=.1, granularity=10, time limit per column=10 min). We exactly reconstruct P in the majority of FedAvg settings.

difference back to the central server. We evaluate gradient disaggregation on updates generated by FedAvg over various parameter settings: local batchsize b, epochs e, user dataset size D (see [156] for more details on these parameters); additionally, we simulate a shift in data distribution by randomly sampling a fraction f of participants' total data set during computation of model updates; finally we test disaggregation on updates generated with and without SGD momentum m. Figure 6.6b shows that relative variance of model updates ($D = 128$) increases with epochs of training, with momentum and with a shifting data distribution. However, as Table 6.1 shows we can reconstruct P exactly in nearly all cases. The failure cases happen at lower (≤ 1) or higher epochs (≥ 64) of training. At lower epochs, we believe parameters of the update are smaller and less distinguished from each other, making reconstruction more difficult; at higher epochs, reconstruction is more difficult as updates are more noisy. With $2 - 32$ epochs, we are generally able to exactly recover P across the

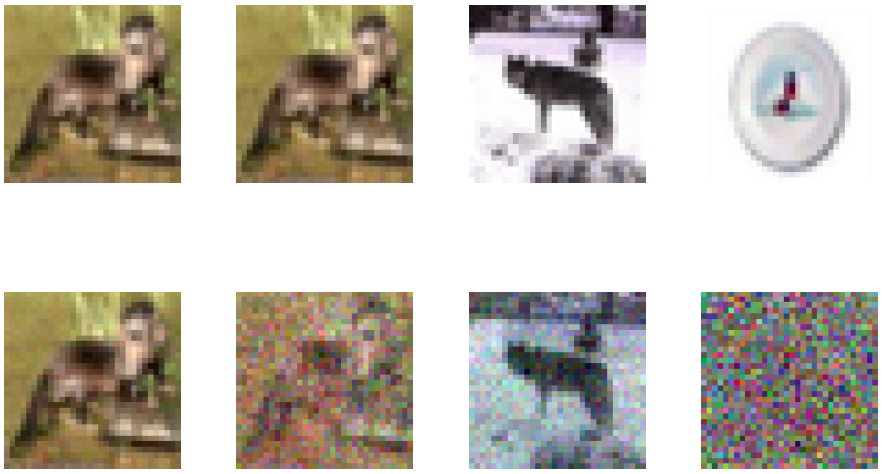
settings.

6.4.2 Gradient Inversion Attacks with Disaggregation

We evaluate the benefits of gradient disaggregation on two methods to invert images from their gradients. Generally, gradient inversion methods optimize image data x', y' to match the target gradient ∇W: $\arg\min_{x',y'} || \frac{\partial l(F(x',W),y')}{\nabla W} - \nabla W||^2$ [245]. This optimization grows exponentially more difficult with larger aggregates [71,245]; we use gradient disaggregation to reduce the aggregate and improve the quality of the inverted images. To quantitatively measure quality, we use PSNR as in [71]. In our results we only show the reconstructed image with the smallest corresponding PSNR to a ground truth image for space.

We perform the attack in [245] on an MLP network on Cifar100 and show the effect of inversion with and without gradient disaggregation across multiple users with each user having 1 image in their dataset (submitting full gradients of that image). Figure 6.7 shows the closest reconstructed image to a user's data example and Table 6.2 shows the corresponding PSNR achieved. With gradient disaggregation, we recover the target user's exact gradient and hence the reconstructed image is high quality. Without disaggregation, reconstruction quality degrades significantly.

We furthermore perform the attack in [71] to invert noisy FedAvg updates. Figure 6.8 and Table 6.3 show the results of inverting fedavg updates with local epochs = 4, batch size = 16, user data set size = 64, with and without gradient disaggregation (100 users, 2 layer MLP). With gradient disaggregation we achieve similar quality as inverting a single model update, whereas inverting an update aggregated over multiple users (users=10) significantly degrades reconstruction quality.



(a) users=1 (or with disaggregation) **(b)** users=2 (no disaggregation) **(c)** users=4 (no disaggregation) **(d)** users=32 (no disaggregation)

Figure 6.7: Recovered images from gradients across users (top image is the closest ground truth). Gradient disaggregation recovers individual users' exact gradients, hence, performing the gradient inversion attack with gradient disaggregation on multiple users yields the same quality as performing the attack on just one user. Without disaggregation, gradient inversion fails on gradients aggregated across more users.

	users=1	users=2	users=4	users=32
PSNR	36.5	18.8	13.9	6.1

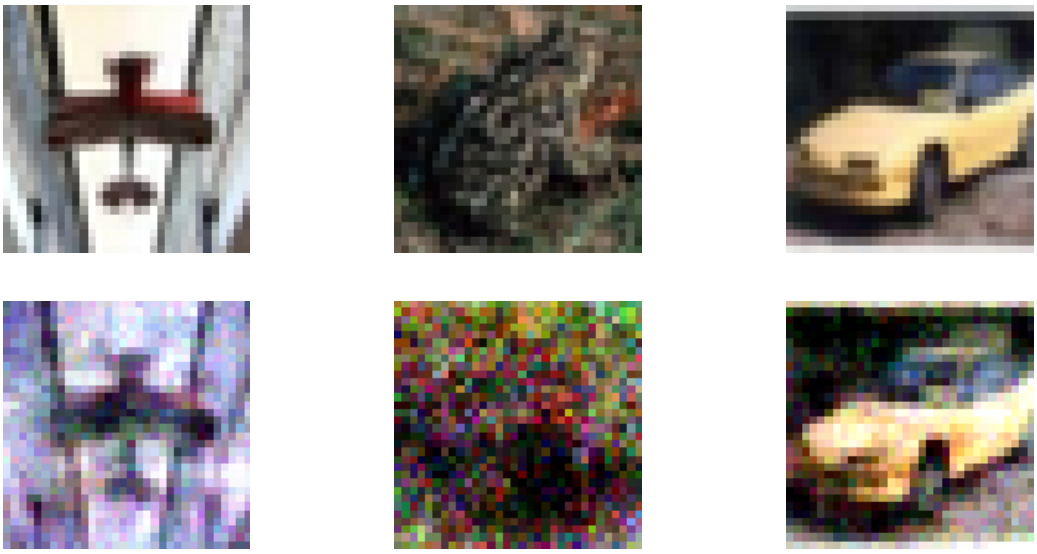
Table 6.2: Corresponding PSNR scores against ground truth for Figure 6.7

	users=1	users=10	users=100 (disaggregated)
PSNR	16.0	13.3	18.6

Table 6.3: Corresponding PSNR scores against ground truth for Figure 6.8.

6.4.3 Property Inference Attacks with Disaggregation

We demonstrate gradient disaggregation on property inference attacks as in [158]. We train a gender model on the LFW dataset [110] and a model to predict whether participants' FedAvg updates (local epochs=4, batchsize=8, data size per user=32) on the gender model contain people of a specific race (hence the attacker's goal may be to learn a participants' images' race from the application). As in [158]



(a) users=1 (no disaggregation)

(b) users=10 (no disaggregation)

(c) users=100 (disaggregated)

Figure 6.8: Recovered images from FedAvg updates across users (top image is closest ground truth). Gradient disaggregation enables high quality inversion on noisy FedAvg updates aggregated across many users; unlike disaggregation on exact gradients, disaggregation on noisy updates recovers the average update submitted across rounds, and we are able to reconstruct high quality images on noisy updates aggregated across many users. Without disaggregation, inversion on updates aggregated over multiple users (users=10) significantly degrades quality.

only the target's dataset contains a significant proportion (p=.5) of images with the specific race and the goal is to determine whether the target's update is present in the aggregated updates over various numbers of users.

Figure 6.9 shows the AUC score of the attack across various numbers of users with and without gradient disaggregation. AUC score quickly degrades with more users; however, with gradient disaggregation high AUC score is maintained across increased numbers of participants as each user's model update is disaggregated exactly, allowing the property inference attack to be performed on each user separately. We note that the requirement in [158] that only the target has the particular data distribution is a limiting assumption, as many participants' data may exhibit the property of interest. With gradient disaggregation, learned properties are attributed to individual participants, enabling

the central server to build profiles of users, violating anonymity.

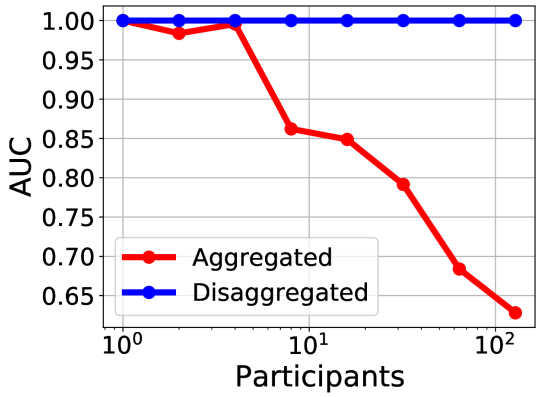


Figure 6.9: Property inference with and without gradient disaggregation (main task: gender classification, auxiliary task: identifying images of specific race) on FedAvg updates. Gradient disaggregation enables property inference on individual model updates and maintains high AUC score across increased number of users.

6.5 Discussion

We introduce gradient disaggregation, a method to disaggregate model updates from sums of model updates given repeated observations and access to summary information from device analytics. Our attack is capable of disaggregating model updates over thousands of users and we apply it to augment existing attacks such as gradient inversion and property inference. Our attack undermines the secure aggregation protocol.

Our findings show that summary metrics such as participation frequency may, when combined with gradient information, be used as an attack vector to undermine individual users' data privacy in federated learning systems. Ways to mitigate this attack include: injecting noise into model updates to reduce efficacy of disaggregation, using differential privacy on the collected device metrics to make reconstruction more difficult, and reducing or eliminating the collection of device analytics. These mitigation strategies may hinder the management of federated learning systems, and employing these techniques to increase privacy must be balanced with the costs to utility. We hope

that bringing awareness to the privacy risks of side channel information in federated learning infrastructure will assist in designing secure federated learning systems.

7

Conclusion

In this dissertation I have asserted that data privacy will be essential in the future of machine learning systems as machine learning and AI become ubiquitous. I have furthermore established that the key challenges to private machine learning systems of the future are primarily computational and have identified three key requirements for these next generation systems: **efficiency**, **privacy** and **security**. Finally, I have presented the major works of my PhD that have pushed this frontier to achieve more deploy-able private machine learning systems: these works accelerate privacy pre-

serving machine learning applications like on-device inference and secure neural network inference by leveraging techniques specific to neural networks, like quantization, combining them with hardware acceleration, like GPU acceleration, and tailoring them toward the specific machine learning privacy technique to achieve maximal efficiency, all while remaining aware of and defending against potential threat vectors like data leakage (i.e: privacy leakage from gradients).

In Chapter 3 we develop PrecisionBatching, which utilizes two key aspects of neural networks and machine learning systems to achieve more efficient non-private machine learning inference: neural network quantization and GPU acceleration. This work presents two key insights that are critical for the rest of the works, namely the importance of neural network specific optimizations like quantization, and the power of hardware acceleration through GPU computation.

In Chapter 4 we develop Tabula which accelerates private neural network inference by using quantization to enable fast table-lookup based secure nonlinear activation function computation, achieving over 10× reduction in communication and runtime while maintaining neural network accuracy.

In Chapter 5 we present GPU-DPF which enables on-device private machine learning inference systems that require access to embedding tables too large to store on-device, by accelerating the cryptographically heavy distributed-point-function operation by over 30× over a CPU, for the purpose of enabling private information retrieval for on-device private machine learning inference systems.

In Chapter 6 we construct GradientDisaggregation, an attack that breaks privacy in federated learning by inverting individual gradient updates from sums of updates given device analytics (i.e: participation counts), showing that federated learning over thousands of users may be insecure, motivating new techniques for ensuring privacy in federated learning.

7.1 Future Research Directions

Machine learning systems of the future will need to be efficient, private, and secure, and considerable amounts of work are still needed to reach this target.

In terms of **efficiency**, private neural network inference still faces high computation, communication and storage costs imposed by the linear and nonlinear layers of the network, particularly **preprocessing costs**; additionally, the issue of **latency** may be an issue in applications where response time is important. Finally, scaling these private neural network inference frameworks to large neural networks or state-of-the-art LLMs may pose considerable differences due to the differences in instruction mix of these larger architectures.

Privacy wise, this thesis has focused primarily on ensuring that the computation private, but did not focus on ensuring that the model outputs were private. Ensuring that the model outputs are private requires **differential privacy** which imposes considerable constraints and tradeoffs on the accuracy of the model by requiring the addition of oftentimes considerable amounts of added noise at training time. Achieving satisfactory tradeoffs between model privacy and model accuracy is a tremendous undertaking that remains to be properly understood and addressed.

Finally, in terms of **security**, security analyses on existing machine learning system infrastructure is largely absent due to the opaque nature of what machine learning system infrastructure looks like in real settings. Clarity and insight into how machine learning infrastructure works in real settings will make a more complete privacy/security analysis more achievable and spur more research into security analyses into machine learning systems that are deployed in the real world.

References

[1] (2021). AES performance. https://openwrt.org/docs/guide-user/perf_and_log/benchmark.openssl.

[2] (2022). Fundamental MPC protocols. https://securecomputation.org/docs/ch3-fundamentalprotocols.pdf.

[3] 4G Network Throughput (2022). 4G network throughput. https://en.wikipedia.org/wiki/4G.

[4] Agarwal, A., Peceny, S., Raykova, M., Schoppmann, P., & Seth, K. (2022). Communication-efficient secure logistic regression. Cryptology ePrint Archive, Paper 2022/866. https://eprint.iacr.org/2022/866.

[5] Agrawal, R., de Castro, L., Yang, G., Juvekar, C., Yazicigil, R., Chandrakasan, A., Vaikuntanathan, V., & Joshi, A. (2023). FAB: An FPGA-based accelerator for bootstrappable fully homomorphic encryption. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*.

[6] Ahmad, I., Yang, Y., Agrawal, D., Abbadi, A. E., & Gupta, T. (2021). Addra: Metadata-private voice communication over fully untrusted infrastructure. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*.

[7] Albericio, J., Delmás, A., Judd, P., Sharify, S., O'Leary, G., Genov, R., & Moshovos, A. (2017). Bit-pragmatic deep neural network computing. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17 (pp. 382–394). New York, NY, USA: ACM.

[8] Ali, A., Lepoint, T., Patel, S., Raykova, M., Schoppmann, P., Seth, K., & Yeo, K. (2021). Communication–Computation trade-offs in PIR. In *30th USENIX Security Symposium (USENIX Security 21)*.

[9] Amazon Enclave (2022). Amazon MPC using enclaves. https://d1.awsstatic.com/events/Summits/reinvent2022/CMP403_Enabling-multi-party-analysis-of-sensitive-data-using-AWS-Nitro-Enclaves-.pdf.

[10] Amazon On Device Speech Recognition (2021). Amazon on device speech recognition. https://www.amazon.science/blog/how-to-make-on-device-speech-recognition-practical.

[11] AMD SEV (2023). AMD SEV. https://developer.amd.com/sev/.

[12] Angel, S., Chen, H., Laine, K., & Setty, S. (2018). PIR with compressed queries and amortized query processing. In *2018 IEEE Symposium on Security and Privacy*.

[13] Apple ATT (2022). Apple app tracking transparency. https://developer.apple.com/documentation/apptrackingtransparency.

[14] ARM Trustzone (2022). ARM trustzone. https://www.arm.com/technologies/trustzone-for-cortex-a.

[15] Arun, A., Setty, S., & Thaler, J. (2023). Jolt: Snarks for virtual machines via lookups. Cryptology ePrint Archive, Paper 2023/1217. https://eprint.iacr.org/2023/1217.

[16] Athalye, A., Carlini, N., & Wagner, D. (2018). Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018*.

[17] Bagdasaryan, E., Veit, A., Hua, Y., Estrin, D., & Shmatikov, V. (2019). How to backdoor federated learning.

[18] Baidu (2017). Baidu deepbench.

[19] Bansal, M., Krizhevsky, A., & Ogale, A. (2018). Chauffeurnet: Learning to drive by imitating the best and synthesizing the worst.

[20] Beaver, D. (1991). Efficient multiparty protocols using circuit randomization. volume 576 (pp. 420–432).

[21] Beaver, D. (1992). Efficient multiparty protocols using circuit randomization. In J. Feigenbaum (Ed.), *Advances in Cryptology — CRYPTO '91* (pp. 420–432). Berlin, Heidelberg: Springer Berlin Heidelberg.

[22] Beaver, D. (1995). Precomputing oblivious transfer. volume 963 (pp. 97–109).

[23] Bengio, Y., Ducharme, R., Vincent, P., & Janvin, C. (2003). A neural probabilistic language model. *J. Mach. Learn. Res.*, 3(null), 1137–1155.

[24] Berner, C., Brockman, G., Chan, B., & et al., V. C. (2019). Dota 2 with large scale deep reinforcement learning.

[25] Blanchard, P., El Mhamdi, E. M., Guerraoui, R., & Stainer, J. (2017). Machine learning with adversaries: Byzantine tolerant gradient descent. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, & R. Garnett (Eds.), *Advances in Neural Information Processing Systems*, volume 30 (pp. 119–129).: Curran Associates, Inc.

[26] Bonawitz, K., Ivanov, V., Kreuter, B., Marcedone, A., McMahan, H. B., Patel, S., Ramage, D., Segal, A., & Seth, K. (2017). Practical secure aggregation for privacy-preserving machine learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17 (pp. 1175–1191). New York, NY, USA: Association for Computing Machinery.

[27] Bonawitz, K. A., Eichner, H., Grieskamp, W., Huba, D., Ingerman, A., Ivanov, V., Kiddon, C. M., Konečný, J., Mazzocchi, S., McMahan, B., Overveldt, T. V., Petrou, D., Ramage, D., & Roselander, J. (2019). Towards federated learning at scale: System design. In *SysML 2019*. To appear.

[28] Bowman, S. R., Angeli, G., Potts, C., & Manning, C. D. (2015). A large annotated corpus for learning natural language inference. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP)*: Association for Computational Linguistics.

[29] Boyle, E., Chandran, N., Gilboa, N., Gupta, D., Ishai, Y., Kumar, N., & Rathee, M. (2020). Function secret sharing for mixed-mode and fixed-point secure computation. Cryptology ePrint Archive, Paper 2020/1392. https://eprint.iacr.org/2020/1392.

[30] Boyle, E., Gilboa, N., & Ishai, Y. (2015). Function secret sharing. In *International Conference on the Theory and Application of Cryptographic Techniques*.

[31] Boyle, E., Gilboa, N., & Ishai, Y. (2019a). Secure computation with preprocessing via function secret sharing. Cryptology ePrint Archive, Paper 2019/1095. https://eprint.iacr.org/2019/1095.

[32] Boyle, E., Gilboa, N., & Ishai, Y. (2019b). Secure computation with preprocessing via function secret sharing. Cryptology ePrint Archive, Paper 2019/1095.

[33] Boyle, E., Gilboa, N., Ishai, Y., & Kolobov, V. I. (2023). Information-theoretic distributed point functions. Cryptology ePrint Archive, Paper 2023/028. https://eprint.iacr.org/2023/028.

[34] Cao, D., Zhang, M., Lu, H., Ye, X., Fan, D., Che, Y., & Wang, R. (2021). Streamline ring ORAM accesses through spatial and temporal optimization. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*.

[35] Case, B., Jain, R., Koshelev, A., Leiserson, A., Masny, D., Sandberg, T., Savage, B., Taubeneck, E., Thomson, M., & Yamaguchi, T. (2023). Interoperable private attribution: A distributed attribution and aggregation protocol. Cryptology ePrint Archive, Paper 2023/437.

[36] CCPA (2023). CCPA. https://www.oag.ca.gov/privacy/ccpa.

[37] Chacha20 in TLS (2016). Chacha20 in TLS. https://www.rfc-editor.org/rfc/rfc7905.

[38] Cho, M., Ghodsi, Z., Reagen, B., Garg, S., & Hegde, C. (2021). Sphynx: Relu-efficient network design for private inference.

[39] Choi, J., Venkataramani, S., Srinivasan, V. V., Gopalakrishnan, K., Wang, Z., & Chuang, P. (2019). Accurate and efficient 2-bit quantized neural networks. In A. Talwalkar, V. Smith, & M. Zaharia (Eds.), *Proceedings of Machine Learning and Systems*, volume 1 (pp. 348–359).

[40] Choi, J., Wang, Z., Venkataramani, S., Chuang, P. I.-J., Srinivasan, V., & Gopalakrishnan, K. (2018). Pact: Parameterized clipping activation for quantized neural networks.

[41] Chor, B., Goldreich, O., Kushilevitz, E., & Sudan, M. (1995). Private information retrieval. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*.

[42] Colombo, S., Nikitin, K., Corrigan-Gibbs, H., Wu, D. J., & Ford, B. (2023). Authenticated private information retrieval. Cryptology ePrint Archive, Paper 2023/297.

[43] Corrigan-Gibbs, H., Boneh, D., & Mazières, D. (2015). Riposte: An anonymous messaging system handling millions of users. In *2015 IEEE Symposium on Security and Privacy*.

[44] Corrigan-Gibbs, H., Henzinger, A., & Kogan, D. (2022). Single-server private information retrieval with sublinear amortized time. In *Advances in Cryptology - EUROCRYPT 2022*.

[45] Courbariaux, M., Hubara, I., Soudry, D., El-Yaniv, R., & Bengio, Y. (2016a). Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1.

[46] Courbariaux, M., Hubara, I., Soudry, D., El-Yaniv, R., & Bengio, Y. (2016b). Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1.

[47] Cowan, M., Moreau, T., Chen, T., Bornholt, J., & Ceze, L. (2020). Automatic generation of high-performance quantized machine learning kernels. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, CGO 2020 (pp. 305–316). New York, NY, USA: Association for Computing Machinery.

[48] Cowan, M., Moreau, T., Chen, T., & Ceze, L. (2018). Automating generation of low precision deep learning operators.

[49] Crawford, J. L. H., Gentry, C., Halevi, S., Platt, D., & Shoup, V. (2018). Doing real work with fhe: The case of logistic regression. In *Proceedings of the 6th Workshop on Encrypted Computing; Applied Homomorphic Cryptography*.

[50] Dagan, I., Pereira, F., & Lee, L. (1994). Similarity-based estimation of word cooccurrence probabilities. In *Proceedings of the 32nd Annual Meeting on Association for Computational Linguistics*.

[51] Dalskov, A., Escudero, D., & Keller, M. (2020). Secure evaluation of quantized neural networks. *Proceedings on Privacy Enhancing Technologies*.

[52] Damgård, I., Nielsen, J. B., Nielsen, M., & Ranellucci, S. (2017). The tinytable protocol for 2-party secure computation, or: Gate-scrambling revisited. In *CRYPTO*.

[53] Damgård, I. & Zakarias, R. (2016). Fast oblivious aes a dedicated application of the minimac protocol. In *Proceedings of the 8th International Conference on Progress in Cryptology — AFRICACRYPT 2016 - Volume 9646* (pp. 245–264). Berlin, Heidelberg: Springer-Verlag.

[54] Dao, T., Gu, A., Eichhorn, M., Rudra, A., & Ré, C. (2019). Learning fast algorithms for linear transforms using butterfly factorizations. In *Proceedings of the 17th International Conference on Machine Learning (ICML 2019)*.

[55] de Bruin, B., Zivkovic, Z., & Corporaal, H. (2020). Quantization of deep neural networks for accumulator-constrained processors. *Microprocessors and Microsystems*.

[56] Dessouky, G., Koushanfar, F., Sadeghi, A., Schneider, T., Zeitouni, S., & Zohner, M. (2017). Pushing the communication barrier in secure computation using lookup tables. *IACR Cryptol. ePrint Arch.*

[57] Dettmers, T., Lewis, M., Belkada, Y., & Zettlemoyer, L. (2022). Llm. int8 (): 8-bit matrix multiplication for transformers at scale. *arXiv preprint arXiv:2208.07339*.

[58] DLRM (2020). Deep learning recommendation model. https://www.adityaagrawal.net/blog/dnn/dlrm.

[59] Doerner, J. & Shelat, A. (2017). Scaling ORAM for secure computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*.

[60] Dong, Z., Yao, Z., Gholami, A., Mahoney, M., & Keutzer, K. (2019). Hawq: Hessian aware quantization of neural networks with mixed-precision.

[61] Dufter, P., Zhao, M., Schmitt, M., Fraser, A., & Schütze, H. (2018). Embedding learning through multilingual concept induction. arXiv:1801.06807.

[62] European Union (2016). Gdpr. https://gdpr-info.eu/.

[63] Fan, S., Wang, Z., Xu, W., Hou, R., Meng, D., & Zhang, M. (2022). TensorFHE: Achieving practical computation on encrypted data using GPGPU. arXiv:2212.14191.

[64] Feldmann, A., Samardzic, N., Krastev, A., Devadas, S., Dreslinski, R., Eldefrawy, K., Genise, N., Peikert, C., & Sanchez, D. (2021). F1: A fast and programmable accelerator for fully homomorphic encryption (extended version). arXiv:2109.05371.

[65] Fleder, M. & Shah, D. (2020). I know what you bought at chipotle for $9.81 by solving a linear inverse problem. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(3).

[66] Fletcher, C. W., Ren, L., Kwon, A., Van Dijk, M., & Devadas, S. (2015). Freecursive ORAM: [nearly] free recursion and integrity verification for position-based oblivious RAM. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*.

[67] Forbes MPC (2021). Forbes multiparty computation adoption. https://www.forbes.com/sites/forbestechcouncil/2021/10/26/multi-party-computation-private-inputs-public-outputs/?sh=2e2abccd1bb0.

[68] FTC (2019). Ftc privacy restrictions facebook. https://www.ftc.gov/news-events/press-releases/2019/07/ftc-imposes-5-billion-penalty-sweeping-new-privacy-restrictions.

[69] Fung, C., Yoon, C. J. M., & Beschastnikh, I. (2020). Mitigating sybils in federated learning poisoning.

[70] GDPR (2016). GDPR. https://gdpr.eu/what-is-gdpr/.

[71] Geiping, J., Bauermeister, H., Dröge, H., & Moeller, M. (2020). Inverting gradients – how easy is it to break privacy in federated learning?

[72] Gentry, C. (2009). Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*.

[73] Gentry, C. & Halevi, S. (2019). Compressible FHE with applications to PIR. *IACR Cryptol. ePrint Arch.*

[74] Ghodsi, Z., Jha, N. K., Reagen, B., & Garg, S. (2021). Circa: Stochastic relus for private deep learning. In *Advances in Neural Information Processing Systems*.

[75] Gilboa, N. & Ishai, Y. (2014). Distributed point functions and their applications. In *EUROCRYPT*.

[76] Goldreich, O., Goldwasser, S., & Micali, S. (1986). How to construct random functions. *J. ACM*.

[77] Goldreich, O. & Ostrovsky, R. (1996). Software protection and simulation on oblivious RAMs. *J. ACM*.

[78] Gong, Y., Jiang, Z., Feng, Y., Hu, B., Zhao, K., Liu, Q., & Ou, W. (2020a). Edgerec: Recommender system on edge in mobile taobao.

[79] Gong, Y., Jiang, Z., Zhao, K., Liu, Q., & Ou, W. (2020b). EdgeRec: Recommender system on edge in mobile taobao. *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*.

[80] Google (2020). Quantization aware training with tensorflow.

[81] Google (2022a). Federated learning google. https://federated.withgoogle.com/.

[82] Google (2022b). How ai powers great search results. https://blog.google/products/search/how-ai-powers-great-search-results/.

[83] Google Confidential Computing (2020). Google confidential computing. https://cloud.google.com/confidential-computing.

[84] Google Cross App Tracking Restriction (2022). Google cross to restrict cross app tracking. https://www.pcmag.com/news/google-to-restrict-cross-app-tracking-of-users-on-android.

[85] Google Distributed Point Function (2022). Google research distributed point function. https://github.com/google/distributed_point_functions.

[86] Google MPS (2019). Google secure multiparty computation. https://security.googleblog.com/2019/06/helping-organizations-do-more-without-collecting-more-data.html.

[87] Graves, A., rahman Mohamed, A., & Hinton, G. (2013). Speech recognition with deep recurrent neural networks.

[88] Groq (2020). Groq tsp leads in inference performance.

[89] Gupta, K., Kumaraswamy, D., Chandran, N., & Gupta, D. (2022). Llama: A low latency math library for secure inference. Cryptology ePrint Archive, Paper 2022/793. https://eprint.iacr.org/2022/793.

[90] Gupta, U., Hsia, S., Saraph, V., Wang, X., Reagen, B., Wei, G.-Y., Lee, H.-H. S., Brooks, D., & Wu, C.-J. (2020a). Deeprecsys: A system for optimizing end-to-end at-scale neural recommendation inference.

[91] Gupta, U., Hsia, S., Saraph, V., Wang, X., Reagen, B., Wei, G.-Y., Lee, H.-H. S., Brooks, D., & Wu, C.-J. (2020b). Deeprecsys: A system for optimizing end-to-end at-scale neural recommendation inference.

[92] Gupta, U., Hsia, S., Saraph, V., Wang, X., Reagen, B., Wei, G.-Y., Lee, H.-H. S., Brooks, D., & Wu, C.-J. (2020c). DeepRecSys: A system for optimizing end-to-end at-scale neural recommendation inference. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*.

[93] Gupta, U., Wu, C., Wang, X., Naumov, M., Reagen, B., Brooks, D., Cottel, B., Hazelwood, K. M., Hempstead, M., Jia, B., Lee, H. S., Malevich, A., Mudigere, D., Smelyanskiy, M., Xiong, L., & Zhang, X. (2020d). The architectural implications of Facebook's DNN-based personalized recommendation. In *2020 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*.

[94] Gurobi Optimization, L. (2020). Gurobi optimizer reference manual.

[95] Han, S., Liu, X., Mao, H., Pu, J., Pedram, A., Horowitz, M. A., & Dally, W. J. (2016a). Eie: Efficient inference engine on compressed deep neural network. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*.

[96] Han, S., Mao, H., & Dally, W. J. (2016b). Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *International Conference on Learning Representations (ICLR)*.

[97] Han, S., Pool, J., Tran, J., & Dally, W. (2015). Learning both weights and connections for efficient neural network. In *Advances in Neural Information Processing Systems (NIPS)* (pp. 1135–1143).

[98] Hard, A., Kiddon, C. M., Ramage, D., Beaufays, F., Eichner, H., Rao, K., Mathews, R., & Augenstein, S. (2018a). Federated learning for mobile keyboard prediction.

[99] Hard, A., Kiddon, C. M., Ramage, D., Beaufays, F., Eichner, H., Rao, K., Mathews, R., & Augenstein, S. (2018b). Federated learning for mobile keyboard prediction.

[100] Harper, F. M. & Konstan, J. A. (2015). The MovieLens datasets: History and context. *ACM Trans. Interact. Intell. Syst.*

[101] Hejazinia, M., Huba, D., Leontiadis, I., Maeng, K., Malek, M., Melis, L., Mironov, I., Nasr, M., Wang, K., & Wu, C.-J. (2022). Fel: High capacity learning for recommendation and ranking via federated ensemble learning. arXiv:2206.03852.

[102] Henry, R. (2016). Polynomial batch codes for efficient IT-PIR. *Proceedings on Privacy Enhancing Technologies*.

[103] Henzinger, A., Hong, M. M., Corrigan-Gibbs, H., Meiklejohn, S., & Vaikuntanathan, V. (2023). One server for the price of two: Simple and fast single-server private information retrieval. In *32nd USENIX Security Symposium (USENIX Security 23)*.

[104] Hitaj, B., Ateniese, G., & Perez-Cruz, F. (2017). Deep models under the gan: Information leakage from collaborative deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17 (pp. 603–618). New York, NY, USA: Association for Computing Machinery.

[105] Hochreiter, S. & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8), 1735–1780.

[106] Hoffman, M., Shahriari, B., Aslanides, J., Barth-Maron, G., Behbahani, F., Norman, T., Abdolmaleki, A., Cassirer, A., Yang, F., Baumli, K., et al. (2020). Acme: A research framework for distributed reinforcement learning. *arXiv preprint arXiv:2006.00979*.

[107] Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., & Adam, H. (2017). Mobilenets: Efficient convolutional neural networks for mobile vision applications.

[108] Hua, W., Umar, M., Zhang, Z., & Suh, G. E. (2022a). GuardNN: Secure accelerator architecture for privacy-preserving deep learning. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*.

[109] Hua, W., Umar, M., Zhang, Z., & Suh, G. E. (2022b). MGX: Near-zero overhead memory protection for data-intensive accelerators. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*.

[110] Huang, G. B., Ramesh, M., Berg, T., & Learned-Miller, E. (2007). *Labeled Faces in the Wild: A Database for Studying Face Recognition in Unconstrained Environments*. Technical Report 07-49, University of Massachusetts, Amherst.

[111] Huang, Y., Katz, J., & Evans, D. (2013). Efficient secure two-party computation using symmetric cut-and-choose. Cryptology ePrint Archive, Paper 2013/081.

[112] Huang, Z., jie Lu, W., Hong, C., & Ding, J. (2022). Cheetah: Lean and fast secure Two-Party deep neural network inference. In *31st USENIX Security Symposium (USENIX Security 22)* (pp. 809–826). Boston, MA: USENIX Association.

[113] Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., & Bengio, Y. (2016). Binarized neural networks. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, & R. Garnett (Eds.), *Advances in Neural Information Processing Systems 29* (pp. 4107–4115). Curran Associates, Inc.

[114] Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., & Bengio, Y. (2017). Quantized neural networks: Training neural networks with low precision weights and activations. *J. Mach. Learn. Res.*, 18(1), 6869–6898.

[115] Intel SGX (2022). Intel SGX. https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html.

[116] IPA (2023). Interoperable private attribution. https://github.com/patcg-individual-drafts/ipa/.

[117] Ishai, Y., Kushilevitz, E., & Meldgaard, S. (2013). On the power of correlated randomness in secure computation. In *In Proc. TCC 2013* (pp. 600–620).

[118] Ishai, Y., Kushilevitz, E., Ostrovsky, R., & Sahai, A. (2004). Batch codes and their applications. In *Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing*.

[119] Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., Adam, H., & Kalenichenko, D. (2017). Quantization and training of neural networks for efficient integer-arithmetic-only inference.

[120] Jha, N. K., Ghodsi, Z., Garg, S., & Reagen, B. (2021a). Deepreduce: Relu reduction for fast private inference.

[121] Jha, N. K., Ghodsi, Z., Garg, S., & Reagen, B. (2021b). Deepreduce: Relu reduction for fast private inference. In *Proceedings of the 38th International Conference on Machine Learning*.

[122] Johnson, J. (2018). Rethinking floating point for deep learning.

[123] Jouppi, N., Kurian, G., Li, S., Ma, P., Nagarajan, R., Nai, L., Patil, N., Subramanian, S., Swing, A., Towles, B., Young, C., Zhou, X., Zhou, Z., & Patterson, D. A. (2023). TPU v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*.

[124] Judd, P., Albericio, J., Hetherington, T., Aamodt, T. M., & Moshovos, A. (2016). Stripes: Bit-serial deep neural network computing. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (pp. 1–12).

[125] Juvekar, C., Vaikuntanathan, V., & Chandrakasan, A. (2018). Gazelle: A low latency framework for secure neural network inference. In *27th USENIX Security Symposium (USENIX Security 18)*.

[126] Kairouz, P., McMahan, H. B., Avent, B., Bellet, A., Bennis, M., Bhagoji, A. N., Bonawitz, K., Charles, Z., Cormode, G., Cummings, R., D'Oliveira, R. G. L., Rouayheb, S. E., Evans, D., Gardner, J., Garrett, Z., Gascón, A., Ghazi, B., Gibbons, P. B., Gruteser, M., Harchaoui, Z., He, C., He, L., Huo, Z., Hutchinson, B., Hsu, J., Jaggi, M., Javidi, T., Joshi, G., Khodak, M., Konečný, J., Korolova, A., Koushanfar, F., Koyejo, S., Lepoint, T., Liu, Y., Mittal, P., Mohri, M., Nock, R., Özgür, A., Pagh, R., Raykova, M., Qi, H., Ramage, D., Raskar, R., Song, D., Song, W., Stich, S. U., Sun, Z., Suresh, A. T., Tramèr, F., Vepakomma, P., Wang, J., Xiong,

L., Xu, Z., Yang, Q., Yu, F. X., Yu, H., & Zhao, S. (2019). Advances and open problems in federated learning.

[127] Keller, M. (2020). MP-SPDZ: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*.

[128] Keller, M., Orsini, E., Rotaru, D., Scholl, P., Soria-Vazquez, E., & Vivek, S. (2017). Faster secure multi-party computation of aes and des using lookup tables. In *International Conference on Applied Cryptography and Network Security*.

[129] Keller, M. & Sun, K. (2021). Secure quantized training for deep learning. *CoRR*, abs/2107.00501.

[130] Kim, J., Kim, S., Choi, J., Park, J., Kim, D., & Ahn, J. H. (2023). SHARP: A short-word hierarchical accelerator for robust and practical fully homomorphic encryption. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*.

[131] Kim, S., Kim, J., Kim, M. J., Jung, W., Kim, J., Rhu, M., & Ahn, J. H. (2022). BTS: An accelerator for bootstrappable fully homomorphic encryption. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*.

[132] Knott, B., Venkataraman, S., Hannun, A., Sengupta, S., Ibrahim, M., & van der Maaten, L. (2021a). Crypten: Secure multi-party computation meets machine learning. In *arXiv 2109.00984*.

[133] Knott, B., Venkataraman, S., Hannun, A. Y., Sengupta, S., Ibrahim, M., & van der Maaten, L. (2021b). CrypTen: Secure multi-party computation meets machine learning. In *Neural Information Processing Systems*.

[134] Konečný, J., McMahan, B., & Ramage, D. (2015). Federated optimization:distributed optimization beyond the datacenter.

[135] Konečný, J., McMahan, H. B., Yu, F. X., Richtárik, P., Suresh, A. T., & Bacon, D. (2017). Federated learning: Strategies for improving communication efficiency.

[136] Krishnan, S., Chitlangia, S., Lam, M., Wan, Z., Faust, A., & Reddi, V. J. (2019a). Quantized reinforcement learning (quarl).

[137] Krishnan, S., Lam, M., Chitlangia, S., Wan, Z., Barth-Maron, G., Faust, A., & Reddi, V. J. (2019b). Quarl: Quantization for fast and environmentally sustainable reinforcement learning. *arXiv preprint arXiv:1910.01055*.

[138] Krizhevsky, A. (2009). *Learning multiple layers of features from tiny images*. Technical report.

[139] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, & K. Q. Weinberger (Eds.), *Advances in Neural Information Processing Systems 25* (pp. 1097–1105). Curran Associates, Inc.

[140] Launchbury, J., Diatchki, I. S., DuBuisson, T., & Adams-Moran, A. (2012). Efficient lookup-table protocol in secure multiparty computation. *SIGPLAN Not.*

[141] Lee, E. H., Miyashita, D., Chai, E., Murmann, B., & Wong, S. S. (2017). Lognet: Energy-efficient neural networks using logarithmic computation. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*: IEEE.

[142] Lee, Y., Seo, S. H., Choi, H., Sul, H. U., Kim, S., Lee, J. W., & Ham, T. J. (2021). MERCI: Efficient embedding reduction on commodity hardware via sub-query memoization. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.

[143] Lehmkuhl, R., Mishra, P., Srinivasan, A., & Popa, R. A. (2021). Muse: Secure inference resilient to malicious clients. In *30th USENIX Security Symposium (USENIX Security 21)*.

[144] Li, R., Ishimaki, Y., & Yamana, H. (2019). Fully homomorphic encryption with table lookup for privacy-preserving smart grid. In *2019 IEEE International Conference on Smart Computing (SMARTCOMP)*.

[145] Li, T., Sahu, A. K., Talwalkar, A., & Smith, V. (2020a). Federated learning: Challenges, methods, and future directions. *IEEE Signal Processing Magazine*, 37(3), 50–60.

[146] Li, Y., Dong, X., & Wang, W. (2020b). Additive powers-of-two quantization: An efficient non-uniform discretization for neural networks.

[147] Lin, J., Liang, L., Qu, Z., Ahmad, I., Liu, L., Tu, F., Gupta, T., Ding, Y., & Xie, Y. (2022). INSPIRE: In-storage private information retrieval via protocol and architecture co-design. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*.

[148] Lin, Z., Courbariaux, M., Memisevic, R., & Bengio, Y. (2015). Neural networks with few multiplications. *International Conference on Learning Representations (ICLR)*.

[149] Liu, G., Li, K., Xiao, Z., & Wang, R. (2022). PS-ORAM: Efficient crash consistency support for oblivious RAM on NVM. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*.

[150] Liu, J., Juuti, M., Lu, Y., & N., A. (2017). *Oblivious Neural Network Predictions via MiniONN Transformations*.

[151] Lo, H.-K. (1997). Insecurity of quantum secure computations. *Phys. Rev. A*.

[152] Lyu, L., Yu, H., & Yang, Q. (2020). Threats to federated learning: A survey.

[153] McCabe, K. E. (2013). Just you and me and netflix makes three: Implications for allowing "frictionless sharing" of personally identifiable information under the video privacy protection act. Act, 20 J. Intell. Prop. L. 413.

[154] McKinstry, J. L., Esser, S. K., Appuswamy, R., Bablani, D., Arthur, J. V., Yildiz, I. B., & Modha, D. S. (2019). Discovering low-precision networks close to full-precision networks for efficient embedded inference.

[155] McMahan, B. & Ramage, D. (2017). Federated learning google. https://ai.googleblog.com/2017/04/federated-learning-collaborative.html.

[156] McMahan, H. B., Moore, E., Ramage, D., Hampson, S., & y Arcas, B. A. (2017). Communication-efficient learning of deep networks from decentralized data.

[157] Melis, G., Dyer, C., & Blunsom, P. (2017). On the state of the art of evaluation in neural language models. *International Conference on Learning Representations (ICLR)*.

[158] Melis, L., Song, C., De Cristofaro, E., & Shmatikov, V. (2019). Exploiting unintended feature leakage in collaborative learning. In *2019 IEEE Symposium on Security and Privacy (SP)* (pp. 691–706).

[159] Mengkai, S., Wang, Z., Zhang, Z., Song, Y., Wang, Q., Ren, J., & Qi, H. (2020). Analyzing user-level privacy attack against federated learning. *IEEE Journal on Selected Areas in Communications*, PP, 1–1.

[160] Menon, S. J. & Wu, D. J. (2022a). Spiral: Fast, high-rate single-server pir via fhe composition. Cryptology ePrint Archive, Paper 2022/368. https://eprint.iacr.org/2022/368.

[161] Menon, S. J. & Wu, D. J. (2022b). Spiral: Fast, high-rate single-server PIR via FHE composition. Cryptology ePrint Archive, Paper 2022/368.

[162] Merity, S., Xiong, C., Bradbury, J., & Socher, R. (2016). Pointer sentinel mixture models. *International Conference on Learning Representations (ICLR)*.

[163] Merity, S., Xiong, C., Bradbury, J., & Socher, R. (2017). Pointer sentinel mixture models. In *International Conference on Learning Representations*.

[164] Mert, A. C., Aikata, Kwon, S., Shin, Y., Yoo, D., Lee, Y., & Roy, S. S. (2022). Medha: Microcoded hardware accelerator for computing on encrypted data. Cryptology ePrint Archive, Paper 2022/480.

[165] Meta (2020). How does facebook use machine learning to deliver ads. https://www.facebook.com/business/news/good-questions-real-answers-how-does-facebook-use-machine-learning-to-deliver-ads.

[166] Meta (2022). Federated learning meta for mobile devices. https://engineering.fb.com/2022/06/14/production-engineering/federated-learning-differential-privacy/.

[167] Meta Multi-Party Computation (2022). Meta multi-party computation. https://privacytech.fb.com/multi-party-computation/.

[168] Meta Privacy Enhancing Technologies (2022). Meta privacy enhancing technologies. https://www.facebook.com/business/news/our-progress-on-developing-and-incorporating-privacy-enhancing-technologies.

[169] Microsoft Azure Confidential Computing (2022). Microsoft Azure confidential computing. https://learn.microsoft.com/en-us/azure/architecture/example-scenario/confidential/healthcare-inference.

[170] Mishra, P., Lehmkuhl, R., Srinivasan, A., Zheng, W., & Popa, R. A. (2020a). Delphi: A cryptographic inference service for neural networks. In *29th USENIX Security Symposium (USENIX Security 20)* (pp. 2505–2522).: USENIX Association.

[171] Mishra, P., Lehmkuhl, R., Srinivasan, A., Zheng, W., & Popa, R. A. (2020b). Delphi: A cryptographic inference service for neural networks. In *29th USENIX Security Symposium (USENIX Security 20)*.

[172] Mishra, P., Lehmkuhl, R., Srinivasan, A., Zheng, W., & Popa, R. A. (2020c). Delphi codebase. https://github.com/mc2-project/delphi.

[173] Mobile Application Average File Size (2017). Mobile application average file size. https://sweetpricing.com/blog/index.html%3Fp=4250.html.

[174] Mohassel, P. & Zhang, Y. (2017). Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy (SP)*.

[175] MPC Alliance (2020). MPC alliance. https://www.mpcalliance.org/.

[176] Mudigere, D., Hao, Y., Huang, J., Jia, Z., Tulloch, A., Sridharan, S., Liu, X., Ozdal, M., Nie, J., Park, J., Luo, L., Yang, J. A., Gao, L., Ivchenko, D., Basant, A., Hu, Y., Yang, J., Ardestani, E. K., Wang, X., Komuravelli, R., Chu, C.-H., Yilmaz, S., Li, H., Qian, J., Feng, Z., Ma, Y., Yang, J., Wen, E., Li, H., Yang, L., Sun, C., Zhao, W., Melts, D., Dhulipala, K., Kishore, K., Graf, T., Eisenman, A., Matam, K. K., Gangidi, A., Chen, G. J., Krishnan, M., Nayak, A., Nair, K., Muthiah, B., khorashadi, M., Bhattacharya, P., Lapukhov, P., Naumov, M., Mathews, A., Qiao, L., Smelyanskiy, M., Jia, B., & Rao, V. (2022). Software-hardware co-design for fast and scalable training of deep learning recommendation models. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*.

[177] Mughees, M. H., Chen, H., & Ren, L. (2021). OnionPIR: Response efficient single-server PIR. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*.

[178] Naor, M. & Pinkas, B. (1999). Oblivious transfer and polynomial evaluation. In *Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing*, STOC '99 (pp. 245–254). New York, NY, USA: Association for Computing Machinery.

[179] Naumov, M., Mudigere, D., Shi, H. M., Huang, J., Sundaraman, N., Park, J., Wang, X., Gupta, U., Wu, C., Azzolini, A. G., Dzhulgakov, D., Mallevich, A., Cherniavskii, I., Lu, Y., Krishnamoorthi, R., Yu, A., Kondratenko, V., Pereira, S., Chen, X., Chen, W., Rao, V., Jia, B., Xiong, L., & Smelyanskiy, M. (2019). Deep learning recommendation model for personalization and recommendation systems. arXiv:1906.00091.

[180] Ni, R., min Chu, H., Castañeda, O., yeh Chiang, P., Studer, C., & Goldstein, T. (2020). Wrapnet: Neural net inference with ultra-low-resolution arithmetic.

[181] Nielsen, J. B., Nordholt, P. S., Orlandi, C., & Burra, S. S. (2012). A new approach to practical active-secure two-party computation. *IACR Cryptol. ePrint Arch.*, 2011, 91.

[182] NLLB Team, Costa-jussà, M. R., Cross, J., Çelebi, O., Elbayad, M., Heafield, K., Heffernan, K., Kalbassi, E., Lam, J., Licht, D., Maillard, J., Sun, A., Wang, S., Wenzek, G., Youngblood, A., Akula, B., Barrault, L., Gonzalez, G. M., Hansanti, P., Hoffman, J., Jarrett, S., Sadagopan, K. R., Rowe, D., Spruit, S., Tran, C., Andrews, P., Ayan, N. F., Bhosale, S., Edunov, S., Fan, A., Gao, C., Goswami, V., Guzmán, F., Koehn, P., Mourachko, A., Ropers, C., Saleem, S., Schwenk, H., & Wang, J. (2022). No language left behind: Scaling human-centered machine translation. arXiv:2207.04672.

[183] NVIDIA (2018). Cutlass linear algebra library.

[184] NVIDIA Cooperative Groups (2017). NVIDIA cooperative groups. https://developer.nvidia.com/blog/cooperative-groups/.

[185] PyTorch (2023). Pytorch quantization. https://pytorch.org/docs/stable/quantization.html.

[186] Qian, J. & Hansen, L. K. (2020). What can we learn from gradients?

[187] Rajat, R., Wang, Y., & Annavaram, M. (2023). LAORAM: A look ahead ORAM architecture for training large embedding tables. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*.

[188] Rajpurkar, P., Zhang, J., Lopyrev, K., & Liang, P. (2016). SQuAD: 100,000+ questions for machine comprehension of text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing* (pp. 2383–2392). Austin, Texas: Association for Computational Linguistics.

[189] Raoufi, M., Zhang, Y., & Yang, J. (2022). IR-ORAM: Path access type based memory intensity reduction for Path-ORAM. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*.

[190] Rass, S., Schartner, P., & Brodbeck, M. (2015). Private function evaluation by local two-party computation. *EURASIP Journal on Information Security*.

[191] Rathee, D., Rathee, M., Goli, R. K. K., Gupta, D., Sharma, R., Chandran, N., & Rastogi, A. (2021). Sirnn: A math library for secure rnn inference.

[192] Rathee, D., Rathee, M., Kumar, N., Chandran, N., Gupta, D., Rastogi, A., & Sharma, R. (2020). Cryptflow2: Practical 2-party secure inference. *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*.

[193] Reagan, B., Gupta, U., Adolf, B., Mitzenmacher, M., Rush, A., Wei, G.-Y., & Brooks, D. (2018). Weightless: Lossy weight encoding for deep neural network compression. In *Proceedings of the 35th International Conference on Machine Learning*.

[194] Ren, L., Yu, X., Fletcher, C. W., Van Dijk, M., & Devadas, S. (2013). Design space exploration and optimization of path oblivious RAM in secure processors. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*.

[195] Riazi, M. S., Laine, K., Pelton, B., & Dai, W. (2020). HEAX: An architecture for computing on encrypted data. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*.

[196] Rouhani, B. D., Riazi, M. S., & Koushanfar, F. (2017). Deepsecure: Scalable provably-secure deep learning.

[197] Ryffel, T., Tholoniat, P., Pointcheval, D., & Bach, F. (2021). Ariann: Low-interaction privacy-preserving deep learning via function secret sharing.

[198] Ryffel, T., Tholoniat, P., Pointcheval, D., & Bach, F. R. (2020). AriaNN: Low-interaction privacy-preserving deep learning via function secret sharing. *Proceedings on Privacy Enhancing Technologies*.

[199] Samardzic, N., Feldmann, A., Krastev, A., Manohar, N., Genise, N., Devadas, S., Eldefrawy, K., Peikert, C., & Sanchez, D. (2022). CraterLake: A hardware accelerator for efficient unbounded computation on encrypted data. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*.

[200] Segal, A., Marcedone, A., Kreuter, B., Ramage, D., McMahan, H. B., Seth, K., Bonawitz, K. A., Patel, S., & Ivanov, V. (2017). Practical secure aggregation for privacy-preserving machine learning. In *CCS*.

[201] Servan-Schreiber, S., Langowski, S., & Devadas, S. (2022). Private approximate nearest neighbor search with sublinear communication. In *2022 IEEE Symposium on Security and Privacy*.

[202] Setty, S., Thaler, J., & Wahby, R. (2023). Unlocking the lookup singularity with lasso. Cryptology ePrint Archive, Paper 2023/1216. https://eprint.iacr.org/2023/1216.

[203] Shamir, A. (1979a). How to share a secret. *Commun. ACM*, 22(11), 612–613.

[204] Shamir, A. (1979b). How to share a secret. *Commun. ACM*.

[205] Sharma, H., Park, J., Suda, N., Lai, L., Chau, B., Kim, J. K., Chandra, V., & Esmaeilzadeh, H. (2017). Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural networks.

[206] Shokri, R., Stronati, M., Song, C., & Shmatikov, V. (2017). Membership inference attacks against machine learning models.

[207] Slawski, M., Hein, M., & Lutsik, P. (2013). Matrix factorization with binary components. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, & K. Q. Weinberger (Eds.), *Advances in Neural Information Processing Systems*, volume 26 (pp. 3210–3218).: Curran Associates, Inc.

[208] So, J., Guler, B., & Avestimehr, A. S. (2020). Turbo-aggregate: Breaking the quadratic aggregation barrier in secure federated learning.

[209] State of California (2018). California consumer privacy act. https://oag.ca.gov/privacy/ccpa.

[210] Stefanov, E., van Dijk, M., Shi, E., Fletcher, C., Ren, L., Yu, X., & Devadas, S. (2013). Path oram: An extremely simple oblivious ram protocol. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*.

[211] Suh, G. E., Clarke, D., Gassend, B., van Dijk, M., & Devadas, S. (2003). AEGIS: Architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th Annual International Conference on Supercomputing*.

[212] Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to sequence learning with neural networks.

[213] Sze, V., Chen, Y.-H., Yang, T.-J., & Emer, J. (2017). Efficient processing of deep neural networks: A tutorial and survey.

[214] Tambe, T., Yang, E.-Y., Wan, Z., Deng, Y., Reddi, V. J., Rush, A., Brooks, D., & Wei, G.-Y. (2019). Adaptivfloat: A floating-point based data type for resilient deep learning inference. *arXiv preprint arXiv:1909.13271*.

[215] Tan, S., Knott, B., Tian, Y., & Wu, D. J. (2021). CryptGPU: Fast privacy-preserving machine learning on the GPU. In *2021 IEEE Symposium on Security and Privacy (SP)*.

[216] Tan, Z., Yang, Z., Zhang, M., Liu, Q., Sun, M., & Liu, Y. (2022). Dynamic multi-branch layers for on-device neural machine translation. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*.

[217] Taobao Ad Dataset (2020). Taobao ad dataset. https://www.kaggle.com/datasets/pavansanagapati/ad-displayclick-data-on-taobaocom.

[218] Tassa, Y., Doron, Y., Muldal, A., Erez, T., Li, Y., Casas, D. d. L., Budden, D., Abdolmaleki, A., Merel, J., Lefrancq, A., et al. (2018). Deepmind control suite. *arXiv preprint arXiv:1801.00690*.

[219] Twitter (2023). Twitter recommendation algorithm. https://blog.twitter.com/engineering/en_us/topics/open-source/2023/twitter-recommendation-algorithm.

[220] Umuroglu, Y. & Jahre, M. (2017). Streamlined deployment for quantized neural networks.

[221] United States of America (2022). American data privacy and protection act. https://www.congress.gov/bill/117th-congress/house-bill/8152/text.

[222] van der Hagen, M. & Lucia, B. (2021). Practical encrypted computing for iot clients.

[223] Wagh, S. (2022). Pika: Secure computation using function secret sharing over rings. Cryptology ePrint Archive, Paper 2022/826. https://eprint.iacr.org/2022/826.

[224] Wagh, S., Tople, S., Benhamouda, F., Kushilevitz, E., Mittal, P., & Rabin, T. (2020). FALCON: Honest-majority maliciously secure framework for private deep learning. arXiv:2004.02229.

[225] Wang, H., Sreenivasan, K., Rajput, S., Vishwakarma, H., Agarwal, S., yong Sohn, J., Lee, K., & Papailiopoulos, D. (2020). Attack of the tails: Yes, you really can backdoor federated learning.

[226] Wang, R., Zhang, Y., & Yang, J. (2017). Cooperative Path-ORAM for effective memory bandwidth sharing in server settings. In *2017 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*.

[227] Wang, R., Zhang, Y., & Yang, J. (2018). D-ORAM: Path-ORAM delegation for low execution interference on cloud servers with untrusted memory. In *2018 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*.

[228] Wang, S. & Jiang, J. (2015). Learning natural language inference with lstm.

[229] Wang, X., Chan, H., & Shi, E. (2015). Circuit ORAM: On tightness of the goldreich-ostrovsky lower bound. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*.

[230] Wei, W., Liu, L., Loper, M., Chow, K.-H., Gursoy, M. E., Truex, S., & Wu, Y. (2020). A framework for evaluating gradient leakage attacks in federated learning.

[231] Wikipedia (2021). Data security law of the people's republic of china. https://en.wikipedia.org/wiki/Data_Security_Law_of_the_People%27s_Republic_of_China.

[232] Wu, H. (2018). Nvidia quantization deck.

[233] Xiao, G., Lin, J., Seznec, M., Wu, H., Demouth, J., & Han, S. (2023). Smoothquant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning* (pp. 38087–38099).: PMLR.

[234] Xiong, W., Ke, L., Jankov, D., Kounavis, M., Wang, X., Northup, E., Yang, J. A., Acun, B., Wu, C.-J., Peter Tang, P. T., Edward Suh, G., Zhang, X., & Lee, H.-H. S. (2022a). SecNDP: Secure near-data processing with untrusted memory. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*.

[235] Xiong, W., Ke, L., Jankov, D., Kounavis, M., Wang, X., Northup, E., Yang, J. A., Acun, B., Wu, C.-J., Tang, P. T. P., et al. (2022b). SecNDP: Secure near-data processing with untrusted memory. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*.

[236] Xu, C., Yao, J., Lin, Z., Ou, W., Cao, Y., Wang, Z., & Zha, H. (2018). Alternating multi-bit quantization for recurrent neural networks. *International Conference on Learning Representations (ICLR)*.

[237] Yang, Y., Zhang, H., Fan, S., Lu, H., Zhang, M., & Li, X. (2023). Poseidon: Practical homomorphic encryption accelerator. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*.

[238] Yao, A. C.-C. (1986). How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*.

[239] Zhao, R., Hu, Y., Dotzel, J., Sa, C. D., & Zhang, Z. (2019). Improving neural network quantization without retraining using outlier channel splitting. In *Proceedings of the 17th International Conference on Machine Learning (ICML 2019)*.

[240] Zhao, W., Xie, D., Jia, R., Qian, Y., Ding, R., Sun, M., & Li, P. (2020a). Distributed hierarchical GPU parameter server for massive scale deep learning ads systems. In *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*.

[241] Zhao, X., Wang, Y., Cai, X., Liu, C., & Zhang, L. (2020b). Linear symmetric quantization of neural networks for low-precision integer hardware. In *International Conference on Learning Representations*.

[242] Zhou, G., Zhu, X., Song, C., Fan, Y., Zhu, H., Ma, X., Yan, Y., Jin, J., Li, H., & Gai, K. (2018). Deep interest network for click-through rate prediction. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery Data Mining*.

[243] Zhou, S., Wu, Y., Ni, Z., Zhou, X., Wen, H., & Zou, Y. (2016). Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients.

[244] Zhu, L., Liu, Z., & Han, S. (2019a). Deep leakage from gradients. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, & R. Garnett (Eds.), *Advances in Neural Information Processing Systems*, volume 32: Curran Associates, Inc.

[245] Zhu, L., Liu, Z., & Han, S. (2019b). Deep leakage from gradients.

[246] Zhu, Y., Wang, X., Ju, L., & Guo, S. (2023). FxHENN: Fpga-based acceleration framework for homomorphic encrypted CNN inference. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*.

[247] Zipf's Law (2022). Zipf's law. https://en.wikipedia.org/wiki/Zipf%27s_law.