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

Maximilian Lam[‡], Jeff Johnson[†], Wenjie Xiong[§], Kiwan Maeng[¶], Udit Gupta[**], Yang Li[†], Liangzhen Lai[†], Ilias Leontiadis[†], Minsoo Rhu[†], Hsien-Hsin S. Lee[‖], Vijay Janapa Reddi[‡], Gu-Yeon Wei[‡], David Brooks[‡], G. Edward Suh[†**]
[†]Meta AI, [‡]Harvard University, [§]Virginia Tech, [¶]Penn State, [‖]Intel, [**]Cornell University

Abstract

On-device machine learning (ML) inference can enable the use of private user data on user devices without revealing them to remote servers. However, a pure on-device solution to private ML inference is impractical for many applications that rely on embedding tables that are too large to be stored on-device. In particular, recommendation models typically use multiple embedding tables each on the order of 1-10 GBs of data, making them impractical to store on-device. To overcome this barrier, we propose the use of private information retrieval (PIR) to efficiently and privately retrieve embeddings from servers without sharing any private information. As off-the-shelf PIR algorithms are usually too computationally intensive to directly use for latency-sensitive inference tasks, we 1) propose novel GPU-based acceleration of PIR, and 2) co-design PIR with the downstream ML application to obtain further speedup. Our GPU acceleration strategy improves system throughput by more than 20× over an optimized CPU PIR implementation, and our PIR-ML co-design provides an over 5× additional throughput improvement at fixed model quality. Together, for various on-device ML applications such as recommendation and language modeling, our system on a single V100 GPU can serve up to 100,000 queries per second—a > 100× throughput improvement over a CPU-based baseline—while maintaining model accuracy.

CCS Concepts: • **Security and privacy** → *Cryptography*; • **Hardware**; • **Computing methodologies** → **Parallel computing methodologies**; **Machine learning**;

Keywords: privacy, security, cryptography, machine learning, GPU, performance

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0372-0/24/04…$15.00
https://doi.org/10.1145/3617232.3624855

ACM Reference Format:
Maximilian Lam, Jeff Johnson, Wenjie Xiong, Kiwan Maeng, Udit Gupta, Yang Li, Liangzhen Lai, Ilias Leontiadis, Minsoo Rhu, Hsien-Hsin S. Lee, Vijay Janapa Reddi, Gu-Yeon Wei, David Brooks, G. Edward Suh. 2024. GPU-based Private Information Retrieval for On-Device Machine Learning Inference. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3617232.3624855

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 [29, 15] and platform policies [9, 37], 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 [40, 72, 22, 70, 97]. 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)-based PIR, in which private embedding lookups are performed by constructing and evaluating DPFs

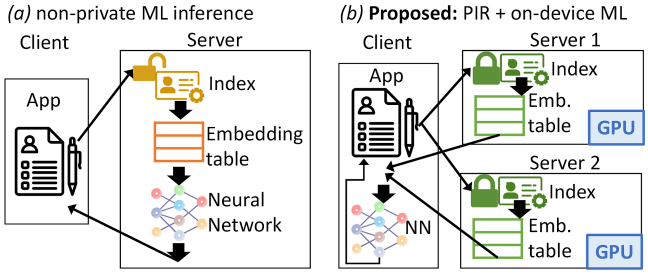


Figure 1. (a) The traditional non-private approach to ML inference, and (b) the proposed approach for private on-device ML inference. Using PIR, a CPU-based client privately obtains embeddings from two GPU-accelerated non-colluding servers; these embeddings are subsequently used as inputs to the client's on-device neural network.

on two non-colluding servers (Figures 1 and 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 [31, 61]. The two-server model is also widely used in the previous work on secure multi-party computation (MPC) for privacy-preserving machine learning [89, 80, 57, 56] or private analytics [14, 50].

Despite their advantages, DPF-based PIR protocols still exhibit massive computational overhead [32, 12], 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 [32, 12]. 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 [40, 41]. The computation and communication requirements of DPF-based PIR make deploying it to real-world ML applications a considerable challenge.

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 [41, 52]. 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 [61, 23, 30, 19], 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 [38], 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 [32], 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 [51, 44, 8], 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 [62], Movielens recommendation [42], and Taobao recommendation [88]. The results show that the proposed optimizations utilizing application-specific data access patterns can increase the ML inference throughput by up to $100\times$ 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.

2 Private On-Device ML Inference

2.1 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 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 [56, 57, 86, 30, 19, 67]. 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 [18]). We also note that incorrect PIR responses only lead to non-optimal suggestions in recommendation models; selective failure attacks [48] 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) [30, 19, 23, 89, 80, 57, 56, 14, 50], 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 [69]. See Section 6 for further discussions.

Table 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

2.2 Key Challenge: Large Embedding Tables

Unfortunately, the embedding tables in machine learning models, especially for recommendation models, are often too large for individual devices [40, 72, 22, 70, 97], 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 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 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 [72, 97]. These models may use user data such as the list of products the user recently purchased [97]. As the number of products can be on the order of millions, the corresponding embedding table can reach several GB to TB in size [40, 70, 35]. Compressing the table is difficult for many real-world models, as it leads to significant accuracy drop [96]. 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 [62]. 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 [24, 73]. Although there are alternative techniques to compress the embeddings

Table 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

(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 [24]. 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 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 [68]. Embedding tables, especially for recommendation models, can easily exceed this range, which makes deploying them on-device impractical [35].

2.3 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*. Removing 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 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 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.

2.4 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) [17, 23]. 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 1).

We use a PIR protocol based on a distributed point function (DPF) [32, 12], which protects accesses using two non-colluding servers. We choose PIR rather than oblivious RAM (ORAM) [34, 84, 92, 85, 49, 7, 10, 27, 78, 90, 91, 76, 60, 75, 13, 94], 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 [61, 20, 31, 59]. 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.

3 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.

3.1 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.

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

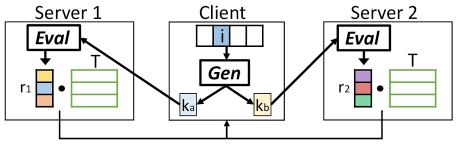


Figure 2. DPF based PIR scheme. The client computes *Gen* to obtain two keys (k_a, k_b) that represent a secret index and sends them to the servers. The servers individually compute *Eval* to obtain secret shares of the answer, from which the client can later retrieve the desired embedding. *Eval* is computationally expensive and is our main target for acceleration.

The server, upon receiving the key, performs expensive cryptographic operations to service the user's query (Figure 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 [83]. 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 $O(L)$.

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) [32, 12], and can be quickly computed even on resource-constrained client devices as shown in Figure 3.

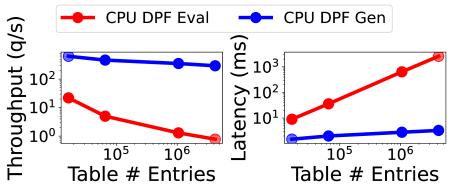


Figure 3. *Gen* vs *Eval* performance. *Gen* is highly efficient and is not our target for optimization.

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 [32, 12] and is the major bottleneck (see Figure 3). Our work focuses on accelerating the *Eval* function. Figure 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 [32], 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 [33] 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 4) to get the secret shares of the indicator vector, using a recursively-defined helper function P:

$$Eval(k, j) = P(d = log(L), j) \tag{1}$$

$$P(0, 0) = C_0[0, 0] \tag{2}$$

$$P(d, j) = PRF_{P(d-1,\lfloor \frac{j}{2} \rfloor)}(j \bmod 2) + C_{P(d-1,\lfloor \frac{j}{2} \rfloor) \bmod 2}[j \bmod 2, d] \tag{3}$$

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 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,

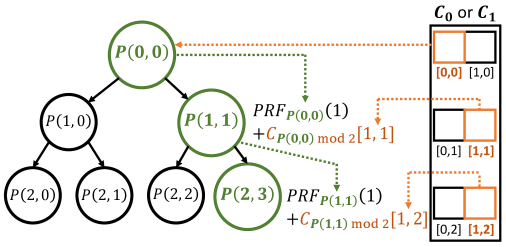


Figure 4. Example of the DPF computation using tree expansion. DPF expansion involves computing the leaves of a binary computation tree which evaluate to a secret-share of a one-hot vector. Computing each node requires evaluating its parent node which involves calling a PRF and adding to it a a codeword value indexed by the height and parity of the node.

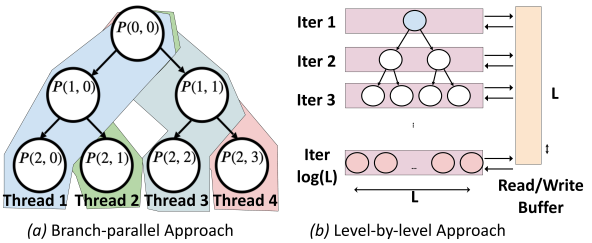


Figure 5. Two naive approaches for parallelizing DPF computation.

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 1), the server needs to calculate $P(1, 1)$, calculating which in turns requires $P(0, 0)$ (Euqation 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 [32] for details on key generation.

3.2 Accelerating PIR with GPU

3.2.1 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, 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 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.

3.2.2 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. 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 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 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.

3.2.3 Memory-bounded Tree Traversal. The tradeoff between computation and memory usage in Section 3.2.2 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 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 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 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 complications. We empirically set $K = 128$, which balances compute utilization and memory usage on a V100 GPU (Figure 8b). Memory-bounded tree traversal achieves both optimal work and low memory usage (Figure 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 6, and achieves utilization benefits of a considerably larger batch size as depicted in Figure 9a.

3.2.4 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 3.1). 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

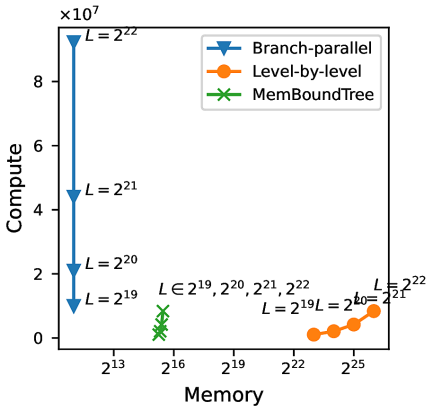
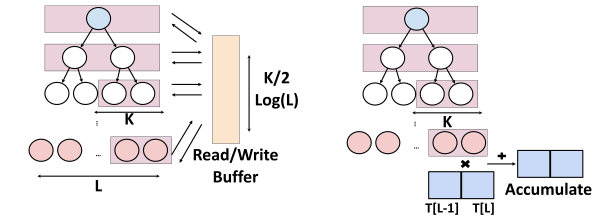


Figure 6. The number of PRFs evaluated (compute) and the peak memory usage (memory) for different parallelization strategies, across different table sizes (L). For both axes, lower is better. The branch-parallel approach redundantly calculates extra PRFs, while the level-by-level approach suffers from 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.

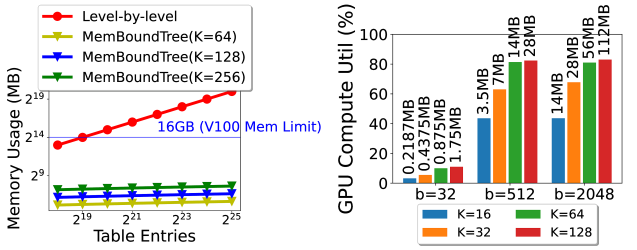


(a) Memory bounded Tree Traversal *(b)* MatMul and DPF fusion

Figure 7. Memory-bounded tree traversal and operator fusion for reducing memory overhead.

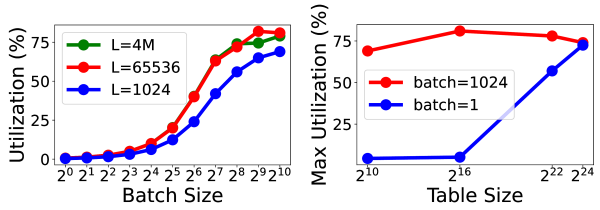
fuse the DPF evaluation operator with the matrix multiplication operator (Figure 7b). Upon reaching a leaf node, a thread immediately performs a dot product between the table entry and the corresponding leaf node output of size K, accumulating the result in local memory. At the end, threads in a single 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.

3.2.5 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



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

Figure 8. The memory usage and the compute resource utilization of the memory-bounded tree traversal.



(a) Batch size vs Util **(b)** Table size vs Util

Figure 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.

to 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 [74] 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 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.

3.2.6 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 performance 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 [16], 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.

3.2.7 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.

4 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 [40] (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) [82], 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.

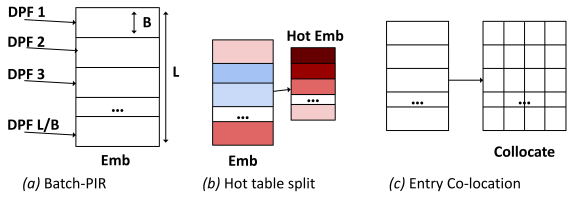


Figure 10. Techniques used to co-design PIR + ML. a) Partial Batch Retrieval, b) splitting the table into a smaller hot table, and c) co-locating frequently accessed entries.

4.1 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 [82], partial batch retrieval (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 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.

4.2 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 [41, 99]. We leverage this observation and add a small *hot table* that holds the top-K frequently accessed indices in addition to the large *full table* that holds all the embedding entries (Figure 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 [58, 21] 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 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 perspective, 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 Evaluation

5.1 Evaluation Setup

Platforms. We evaluate our GPU-based DPF-PIR and compare it with a state-of-the-art CPU implementation [38]. 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 [38], 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 [43] with MovieLens-20M dataset [42], (2) a 2-layer MLP-based recommendation model [43] with Taobao Ads click/display dataset [88], and (3) an LSTM model with Wikitext2 corpus [62]. We protect the user history table [97] 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 [97, 43]. For LSTM, we use perplexity (ppl), a measure of surprise, to measure the model quality. Following the training setup of [62], our model achieves ppl=92.

System Parameters. For application-independent experiments (Figures 13–15, Tables 4–5), unless otherwise stated, we default to an entry size of 2048 bits. Most recommendation models use entries similar or smaller than this [97, 72]. 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).

5.2 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 [38]. 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 + 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 11 shows that the throughput improves by **5–39**× while maintaining the model quality (Acc-eco), and the improvement becomes **40–124**× when small quality degradation is tolerated (Acc-relaxed). GPU optimizations account for 10–20× performance improvement, and PIR-ML co-design can additionally obtain up to 2–5× 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 19.

Table 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

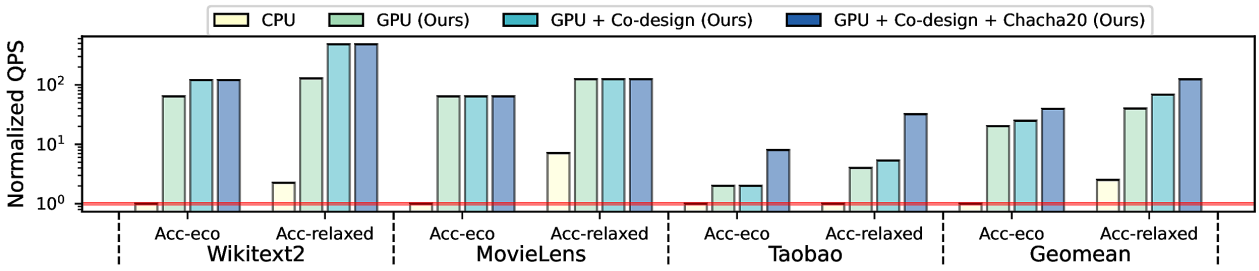


Figure 11. Throughput improvement of our proposed system over the CPU baseline [38]. While preserving accuracy (Acc-eco), our system can improve the throughput on average by 5–39×. When some amount of accuracy degradation is tolerated (Acc-relaxed), the average improvement reaches 40–124×. All configurations searched within the latency (< 300ms) and communication requirement (< 300KB). QPS normalized by the CPU Acc-eco for each benchmark.

Table 3. Unnormalized QPS from Figure 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 5.2

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

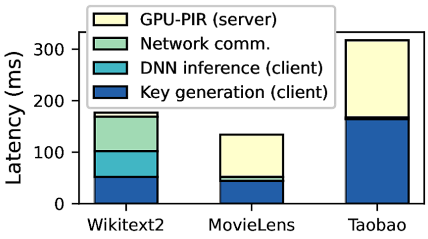


Figure 12. End-to-end latency breakdown of an inference query (i.e: time from client request to receiving and computing the result). Our proposed system makes the PIR latency much lower (Wikitext2) or comparable (MovieLens, Taobao) to the latency of other components. We are able to keep end-to-end latency within a reasonable 500 ms per inference which is acceptable in standard SLAs [41]

benchmarks (e.g., MovieLens queries 72 entries per inference on average).

5.3 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*;

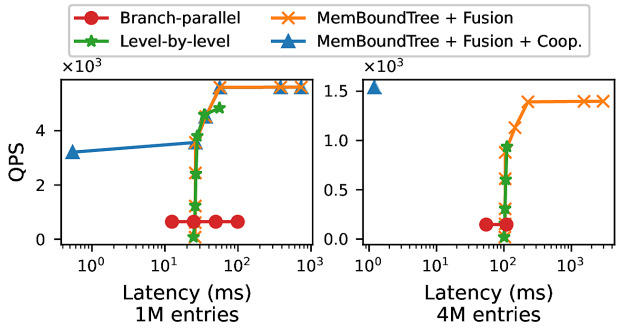


Figure 13. Throughput vs latency for different GPU optimizations: branch-parallel (red), level-by-level (green), memory-bounded tree traversal and operator fusion (orange), and batch/table-size aware scheduling with cooperative groups (blue).

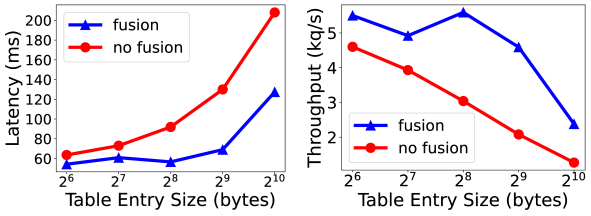
our paper's main focus), (3) client-server network communication (4) client on-device DNN inference. We measure the latency of key generation and DNN inference directly on a single Intel Core i3 CPU. We estimate the network latency assuming 60 Mbit/s bandwidth as in 4G networks [1].

Figure 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 [41].

5.4 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 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 3.2.2, branch-parallel (red) cannot achieve high QPS. Level-by-level (green) is much better, but still



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

Figure 14. Performance impact of table entry size on PIR performance, with and without operator fusion.

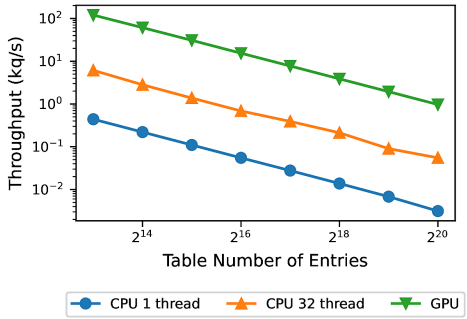


Figure 15. Comparison of throughput performance attained by GPU DPF acceleration compared to an optimized CPU baseline. 1 kq/s = 1,000 queries per second. All methods use the AES-128 PRF.

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 13 (right)), table-size aware scheduling with cooperative groups (blue) obtains significantly better latency without harming throughput.

Performance Impact of Operator Fusion Figure 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×) improvements in both throughput and latency. Figure 14 was obtained with a table with 1M entries; however, the improvement is similar across other table sizes.

Performance Impact of Embedding Entry Size Figure 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 4.2).

Detailed Comparison with CPU We compare our GPU-PIR implementation against an optimized CPU implementation from Google Research [38]. 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 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 different table sizes. Using AES-128 as in the CPU DPF, our GPU implementation consistently achieves > 17× speedup over the 32-threaded CPU implementation. We show the same data in Table 4.

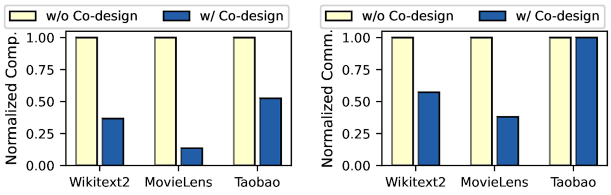
Performance Impact of PRF Table 5 shows the performance of using different PRF functions on a table with 1M entries, a batch size of 512, and a security parameter of 128-bits. Lightweight PRFs can significantly improve the GPU-PIR performance over AES-128. In particular, Chacha20, a well-accepted PRF that is used in high-security applications including TLS 1.3 [16], 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.

Table 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 [38] 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. 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 16. Computation (a) and communication (b) needed to achieve a target model accuracy (Acc-relaxed from Figure 11), with and without ML co-design. Co-design improves computation overhead by 1.9–7.4× and communication overhead by 1–2.6×.

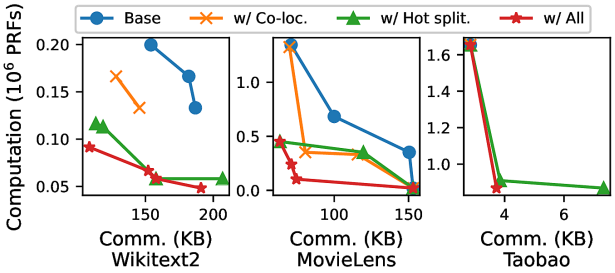


Figure 17. Pareto curve of tradeoff between computation and communication with model accuracy fixed to be within 2% of the baseline.

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

Computation Savings Figure 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 11. Figure 16a shows that co-design reduces the computation significantly, by 1.9×–7.4×.

Communication Savings Figure 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 17 shows this tradeoff across various applications, with model quality fixed to be within 2% of the full precision baseline. Co-design optimizations obtain significantly better tradeoffs than plain batch-PIR.

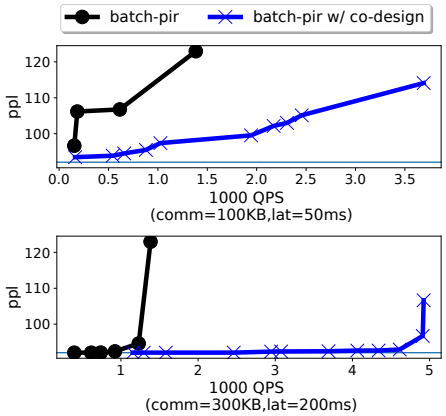


Figure 18. System throughput vs model quality with and without co-design for language model across different budgets.

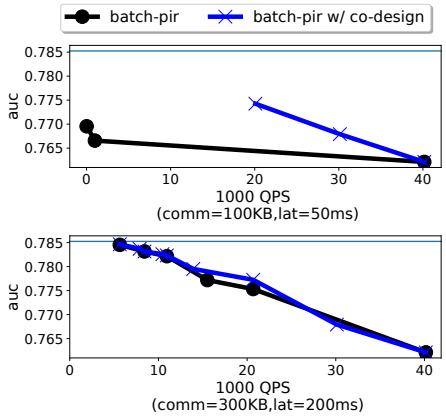


Figure 19. System throughput vs model quality with and without co-design for MovieLens rec across different budgets.

Co-Design Throughput Improvement We show overall co-design throughput improvement over standard batch-PIR across all applications on select budgets in Figures 18, 19, and 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 18 and 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

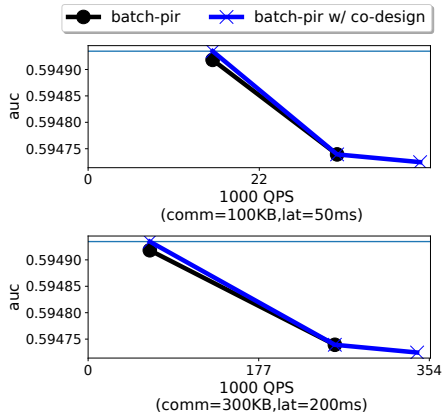


Figure 20. System throughput vs model quality with and without co-design for Taobao rec across different budgets.

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 18) and MovieLens (Figure 19), whose model inputs are entirely sparse features that require embedding table lookups, see much greater improvement with co-design compared to Taobao (Figure 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.

6 Related Work

Privacy-preserving Computation Techniques Prior work on privacy-preserving ML investigated techniques such as fully-homomorphic encryption (FHE) [53, 67], secure multi-party computation (MPC) [56, 57, 89, 80], and trusted execution environments (TEEs) [46, 47, 93]. 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 [2, 79, 26, 55, 98, 25, 95, 63, 54, 77, 81] 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) [61, 31, 20, 59] and n-server (n $\geq$ 2) protocols based on DPFs [30, 19, 23]. We focus on two-server DPF-based PIR protocols, as they are significantly more computation- and communication-efficient than single-server schemes [59, 77, 55, 30, 19, 23]. For example, querying a 1B entry table with a two-server protocol is over 1000× more communication-efficient (2KB vs 3.6MB) [59] and multiple orders of magnitude more computationally-efficient than single-server protocols [45, 8, 4, 3, 71, 61]. For a 1M-entry table, state-of-the-art HE PIR [61] 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 [32, 12] while n>2-server DPF exhibits $O(\sqrt{n})$ communication [11].

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) [56, 57, 89, 80, 30, 19, 23]. 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 [69, 64, 66, 5, 28, 39]. 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 [69, 65] are seeing increasing adoption. Remote attestation capabilities in public cloud TEEs [5, 66, 36, 93] can also be used to further ensure the integrity of two parties.

Batch Private Information Retrieval Various approaches for batch PIR [82, 8, 51, 8, 44] have been proposed. We show that noise tolerance of ML allows the use probabilistic PIR protocols like [82] with minimal accuracy loss.

On-device ML On-device ML has been studied for recommendation [43, 35], speech recognition [6], translation [87], etc. Our work uses on-device ML for privacy, and enables the private use of large server-side embedding tables.

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× speedup over naive system, enabling practical deployment for privacy-sensitive applications.

A Artifact Appendix

A.1 Abstract

This is the artifact appendix for the "GPU-DPF" paper, and our code is publicly available at https://github.com/facebookresearch/GPU-DPF/tree/main. We release code containing optimized GPU kernels that implement distributed point functions (DPF), as presented in our paper. The codebase primarily exposes a PyTorch interface for evaluating DPFs and doing private table lookups. The code demonstrates the kernel is correct (i.e: computes a DPF correctly) and reproduces the main performance results of the GPU kernels that is the core contribution of the paper. Our code is also available at Zenodo: https://zenodo.org/records/10049254.

A.2 Artifact check-list (meta-information)

- **Algorithm: DPF**
- **Program: Kernel**
- **Compilation: bash install.sh (runs python setup.py that compiles using nvcc)**
- **Transformations: N/A**
- **Binary: N/A**
- **Model: N/A**
- **Data set: N/A**
- **Run-time environment: python, pytorch, CUDA**
- **Hardware: V100**
- **Run-time state: N/A**
- **Execution: N/A**
- **Metrics: correctness and QPS performance**
- **Output: QPS performance**
- **Experiments: QPS performance**
- **How much disk space required (approximately)?: not much**
- **How much time is needed to prepare workflow (approximately)?: 10 mins**
- **How much time is needed to complete experiments (approximately)?: 5 mins**
- **Publicly available?: Yes**
- **Code licenses (if publicly available)?: Apache**
- **Data licenses (if publicly available)?: N/A**
- **Workflow framework used?: N/A**
- **Archived (provide DOI)?: N/A**

A.3 Description

A.3.1 How to access. https://github.com/facebookresearch/GPU-DPF/

A.3.2 Hardware dependencies. Please use Linux with V100 GPU.

A.3.3 Software dependencies. python, pytorch, numpy CUDA GPU (tested on cuda > 11.4)

A.3.4 Data sets. N/A

A.3.5 Models. N/A

A.4 Installation

bash install.sh

A.5 Experiment workflow

python benchmark.py outputs performance of DPF across different numbers of table entries across different PRFs, outputting the QPS achieved.

A.6 Evaluation and expected results

Our open-source repo contains installation instructions and tests for reproducing the main result of our paper, which validates the high performance of our GPU DPF kernel. Note 1) we do not include the co-design algorithms (these require downloading large datasets and training that is cumbersome), 2) we do not include Google's CPU DPF implementation which requires custom installation for machines and 3) we only expose the "memory-bounded-tree-traversal" kernel approach (though the code for the other approaches like cooperative groups are still released, just not exposed through PyTorch). The repository primarily reproduces the numbers for Table 4: with 16K entries, we obtain around 50K-60K QPS; with 1M entries, we obtain around 900-1.2K QPS; these numbers amount to a >15× speedup over CPU. Note that there are some differences between the released code and Table 4 of the paper, particularly, in the paper we used a different AES kernel than the one released on Github due to licensing issues (the Github one uses OpenSSL's implementation) – this accounted for a 10% performance difference on the 16K table; additionally Table 4 1M entries used cooperative groups approach which was 30% faster than the memory-bounded-tree-traversal approach as presented in the code. Finally, our code reproduces Table 5 QPS numbers for the Chacha PRF at around 4K QPS. Overall these numbers represent a > 15× speedup over CPU, and constitute the main results of the paper.

A.7 Experiment customization

N/A

A.8 Notes

N/A

A.9 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

References

[1] 4G network throughput. https://en.wikipedia.org/wiki/4G.
[2] Agrawal, R., de Castro, L., Yang, G., Juvekar, C., Yazicigil, R., Chandrakasan, A., Vaikuntanathan, V., and Joshi, A. FAB: An

FPGA-based accelerator for bootstrappable fully homomorphic encryption. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)* (2023).

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

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

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

[6] Amazon on device speech recognition. https://www.amazon.science/blog/how-to-make-on-device-speech-recognition-practical.

[7] AMD SEV. https://developer.amd.com/sev/.

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

[9] Apple app tracking transparency. https://developer.apple.com/documentation/apptrackingtransparency.

[10] ARM trustzone. https://www.arm.com/technologies/trustzone-for-cortex-a.

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

[12] Boyle, E., Gilboa, N., and Ishai, Y. Secure computation with preprocessing via function secret sharing. Cryptology ePrint Archive, Paper 2019/1095, 2019.

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

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

[15] CCPA. https://www.oag.ca.gov/privacy/ccpa.

[16] Chacha20 in TLS. https://www.rfc-editor.org/rfc/rfc7905.

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

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

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

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

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

[22] Deep learning recommendation model. https://www.adityaagrawal.net/blog/dnn/dlrm.

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

[24] Dufter, P., Zhao, M., Schmitt, M., Fraser, A., and Schütze, H. Embedding learning through multilingual concept induction, 2018. arXiv:1801.06807.

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

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

[27] Fletcher, C. W., Ren, L., Kwon, A., Van Dijk, M., and Devadas, S. 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* (2015).

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

[29] GDPR. https://gdpr.eu/what-is-gdpr/.

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

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

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

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

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

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

[36] Google confidential computing. https://cloud.google.com/confidential-computing.

[37] Google cross to restrict cross app tracking. https://www.pcmag.com/news/google-to-restrict-cross-app-tracking-of-users-on-android.

[38] Google research distributed point function. https://github.com/google/distributed_point_functions.

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

[40] Gupta, U., Hsia, S., Saraph, V., Wang, X., Reagen, B., Wei, G.-Y., Lee, H.-H. S., Brooks, D., and Wu, C.-J. 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* (2020).

[41] 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., and Zhang, X. The architectural implications of Facebook's DNN-based personalized recommendation. In *2020 IEEE International Symposium on High-Performance Computer Architecture (HPCA)* (2020).

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

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

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

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

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

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

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

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

[50] Interoperable private attribution. https://github.com/patcg-individual-drafts/ipa/.

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

[52] 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., and Patterson, D. A. 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* (2023).

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

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

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

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

[57] Kumar, N., Rathee, M., Chandran, N., Gupta, D., Rastogi, A., and Sharma, R. CrypTFlow: Secure tensorflow inference. In *2020 IEEE Symposium on Security and Privacy* (2020).

[58] Lee, Y., Seo, S. H., Choi, H., Sul, H. U., Kim, S., Lee, J. W., and Ham, T. J. 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* (2021).

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

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

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

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

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

[64] Meta multi-party computation. https://privacytech.fb.com/multi-party-computation/.

[65] Meta privacy enhancing technologies. https://www.facebook.com/business/news/our-progress-on-developing-and-incorporating-privacy-enhancing-technologies.

[66] Microsoft Azure confidential computing. https://learn.microsoft.com/en-us/azure/architecture/example-scenario/confidential/healthcare-inference.

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

[68] Mobile application average file size. https://sweetpricing.com/blog/index.html%3Fp=4250.html#:~:text=Average%20Android%20and%20iOS%20file%20size&text=And%20the%20average%20iOS%20app%20file%20size%20is%2034.3MB.

[69] MPC alliance. https://www.mpcalliance.org/.

[70] 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., and Rao, V. 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* (2022).

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

[72] 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., and Smelyanskiy, M. Deep learning recommendation model for personalization and recommendation systems. arXiv:1906.00091.

[73] 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., and Wang, J. No language left behind: Scaling human-centered machine translation, 2022. arXiv:2207.04672.

[74] NVIDIA cooperative groups. https://developer.nvidia.com/blog/cooperative-groups/.

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

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

[77] Reagen, B., Choi, W.-S., Ko, Y., Lee, V. T., Lee, H.-H. S., Wei, G.-Y., and Brooks, D. Cheetah: Optimizing and accelerating homomorphic encryption for private inference. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)* (2021).

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

[79] Riazi, M. S., Laine, K., Pelton, B., and Dai, W. 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 (2020).

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

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

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

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

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

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

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

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

[88] Taobao ad dataset. https://www.kaggle.com/datasets/pavansanagapati/ad-displayclick-data-on-taobaocom.

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

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

[91] Wang, R., Zhang, Y., and Yang, J. 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)* (2018).

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

[93] 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., and Lee, H.-H. S. SecNDP: Secure near-data processing with untrusted memory. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)* (2022).

[94] 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. SecNDP: Secure near-data processing with untrusted memory. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)* (2022).

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

[96] Zhao, W., Xie, D., Jia, R., Qian, Y., Ding, R., Sun, M., and Li, P. 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* (2020).

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

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

[99] Zipf's law. https://en.wikipedia.org/wiki/Zipf%27s_law.