cuOT: Accelerating Oblivious Transfer on GPUs for Privacy-preserving Computation

Andrew Gan
Purdue University
West Lafayette, USA
gan35@purdue.edu

Setsuna Yuki
Purdue University
West Lafayette, USA
hou125@purdue.edu

Timothy Rogers
Purdue University
West Lafayette, USA
timrogers@purdue.edu

Zahra Ghodsi
Purdue University
West Lafayette, USA
zahra@purdue.edu

***Abstract*—Privacy and security requirements are gaining importance rapidly as new applications emerge in several domains that work with sensitive user information. Various Secure Multiparty Computation (MPC) techniques in cryptography allow working with data in a privacy-preserving manner. A core building block in many popular MPC protocols is the Oblivious Transfer (OT). The main challenge in deploying MPC protocols is the significant overhead they incur compared to computing over plaintext data. Given the fundamental role of OT in constructing MPC protocols, efficiency improvements in generation of OTs would directly translate into improvements for a wide range of MPC protocols. This paper proposes cuOT, a novel framework for accelerating OT on the GPU based on a CUDA implementation. cuOT implements OT variants based on recent silent OT extension constructions which significantly reduce communication at the cost of higher computation. cuOT implements core building blocks of silent OT extension, incorporating several GPU specific optimizations to improve parallelization and reduce memory transfer overheads. Our evaluations show that cuOT is able to obtain up to 75× and 15× speedup in generating millions of OTs compared to CPU counterparts.**

***Index Terms*—Oblivious Transfer, Privacy-preserving Computation, Graphical Processing Unit**

I. Introduction

Applications that work with user data have increased in scope, enabling new capabilities in health monitoring, traffic management, and finance [1]. Such data typically reports on individual's sensitive information, such as location or health status, and requires techniques to provide strong data protection guarantees [2], [3]. Privacy requirements can be satisfied by tools in cryptography which allow for privacy-preserving computation. For example, Secure Multiparty Computation (MPC) allows two or more parties to compute a function on their inputs and obtain the output, without revealing anything else about their inputs to each other. Similarly, Homomorphic Encryption (HE) allows computing directly on encrypted data. While each of these techniques have their unique advantages and drawbacks, they all share the downside of impractical overhead for modern applications. While building more efficient protocols is an important step in reducing cost and has resulted in significant improvements over the years, we posit that hardware acceleration is another promising direction to bring private computing closer to practicality [4]–[6].

In this paper, we focus on a core primitive in MPC, namely, Oblivious Transfer (OT), which is a fundamental building block in many cryptographic protocols. In a nutshell, OT allows one party (sender) holding two inputs, to send one of their inputs to another party (receiver) based on a choice bit that the receiver holds. At the end of the protocol, the sender will not learn which input was chosen by the receiver, and the receiver will not learn anything about sender's other input. Protocols that use OT typically require a large number of OTs for performing a computation on a single input (e.g., one OT per AND gate in the Boolean representation of the function [7]). As an example use case, consider privacy-preserving machine learning (PPML) which is a popular application of secure computation [8]–[11]. State-of-the-art protocols for PPML combine HE for linear layers and OT for non-linear layer evaluations [8], [10]. However, PPML computations (e.g., performing one private inference) are still orders of magnitude more expensive than their non-private counterparts [10], [11]. Our benchmarks show that one private inference computation (of ImageNet [12] over DenseNet121 [13] model) based on the Cheetah framework [10] requires over 1.17 billion OTs. For this benchmark, non-linear computations are 60% of total runtime, with OT generation constituting the largest computation taking 59% of total non-linear evaluation runtime.

Prior work has explored accelerating cryptographic primitives such as HE on specialized hardware [14], [15], including Graphics Processing Units (GPU) [16]–[19]. GPUs provide an attractive target for hardware acceleration due to their ability to concurrently execute thousands of threads, and their wide-range availability as a computing platform. In this work, we provide the first framework to accelerate OT on the GPU.

Early constructions of OT [20] required a significant amount of communication. Recent developments on "silent" OT extensions have introduced protocols where short random seeds can be silently expanded without any interaction to produce a large number of pseudorandom correlations for constructing OT. We target two silent OT protocols, namely, SOT [21] and Ferret [22]. The two protocols have the same high level operations as we will discuss later: a base oblivious transfer step (Base OT), a binary tree expansion which computes an AES block cipher encryption for each child node, and a large binary matrix vector multiplication (LPN). The tree expansion and LPN steps are the most time consuming operations, while base OT is a one-time step with small overhead.

a) Contributions: This paper presents cuOT, a library for OT acceleration on CUDA-enabled GPUs incorporating various design optimizations. Our contributions are as follows.

- We present implementations of computing blocks for silent OT construction, including bit-wise matrix transpose and tree expansion on the GPU for the first time. We introduce a batched design for tree expansion that places neighboring tree nodes into a coalesced memory block on the GPU to be expanded in parallel, with careful partitioning at each tree level to ensure efficient memory access. The bit-wise summation performed at each tree level is optimized through the use of a warp-synchronized block reduction kernel. Additionally, we implement the LPN step using CUDA's cuFFT library and ensuring optimized distribution of work across several GPUs.
- We utilize our core components summarized above to create cuOT, a GPU library implementing SOT [21] and Ferret [22] silent on GPUs. Our end-to-end implementation incorporates memory optimizations that minimize expensive memory transfers. Additionally, we present multi-GPU implementations of both protocols, and discuss trade-off points. Specifically, we show that SOT scales better with number of GPUs and can outperform Ferret under low bandwidth scenarios. On the other hand, Ferret is more suitable to high bandwidth and lower computing power settings.
- We provide microbenchmarks as well as end-to-end performance evaluations , and compare the runtime of cuOT with state-of-the-art software libraries that implement SOT and Ferret on CPU. Our results show that our tree expansion and LPN for cuOT SOT over GPU obtains up to $25\times$ and $90\times$ speedup over CPU runtime, resulting in $75\times$ improvement in total runtime. Similarly, Ferret's Tree expansion and LPN operations are $10\times$ and $35\times$ faster than their CPU counterparts, resulting in $18\times$ overall runtime speedup.

II. Background

A. Oblivious Transfer

Oblivious transfer is a fundamental building block for MPC protocols including Yao's Garbled Circuits [23] and Goldreich, Micali, and Wigderso [24]. Several variations of the OT protocol exist. In the simplest OT setting, one party (sender) holds two *random* strings (s_0, s_1) and the other party (receiver) holds a *random* choice bit c. At the end of the OT protocol, the receiver obtains one of the strings s_c according to their choice bit. The sender will not learn anything about c, and the receiver will not learn anything about the other string s_{1-c}. This setting is referred to as 1-out-of-2 random OT (choice bit is random), and can be used to implement 1-out-of-2 OT (when choice bit is not random), and 1-out-of-m OT (where one string is selected by the receiver out of m sender strings [25]). One important variant of random OT (ROT) is random correlated OT (RCOT), where the two inputs of the sender are correlated, e.g., $(r, r \oplus \Delta)$ where r is a random string and Δ is a fixed constant. In this case, the receiver obtains $r \oplus c\Delta$ at the end. ROT can be reduced to RCOT using correlation robust hash functions [20]. Therefore, prior work focuses on building optimized constructions for RCOT, which is also the focus of our work.

While oblivious transfer requires *public-key* assumptions [26], a few *Base* OTs can be "extended" to generate many OTs using symmetric-primitives only [27]. While performing Base OTs requires more expensive primitives, they are limited to a small size and combined with OT extension [20] which uses cheaper primitives to obtain a large number of OTs. Earlier solutions for OT extension incurred a high communication cost [20]. Recently, new directions in OT extension have proposed constructions where small correlated seeds can be exchanged between two parties through Base OT. The parties can then expand those seeds "silently" without requiring any interaction into a large number of OTs [21], [22], [28]. These silent OT constructions significantly reduce the communication cost, but come at a high computation cost. As we will show later, the computational cost of silent OT constructions can be alleviated by accelerating them on GPUs, resulting in very efficient implementations. Before detailing our implementation, we will provide a high level description of SOT [21] and Ferret [22] protocols.

B. Puncturable Pseudorandom Function

Pseudorandom functions (PRF) are keyed functions which are indistinguishable from truly random functions. Goldreich, Goldwasser, and Micali [7] proposed a method to construct a PRF F from a length doubling pseudorandom generator, i.e. $G : \{0,1\}^l \rightarrow \{0,1\}^{2l}$. The construction is based on generating a binary tree (namely, GGM tree) where the root node gets the input key k for evaluation, generating $G(k) = G_0(k)||G_1(k)$ where G_0 and G_1 are left and right half of the length doubling pseudorandom generator respectively. In the next level of the tree, the left child node gets $G_0(k)$ as input, and the right child node gets $G_1(k)$ as input. This process continues for depth n equal to input length of the PRF. The evaluation of PRF F_k at input $x = x_n \cdots x_1$ is defined as $F_k(x) = G_{x_n}(G_{x_{n-1}}(\cdots(G_{x_1}(k))))$. In other words, $F_k(x)$ is the leaf node with path $\{x_n \cdots x_1\}$, indicating which children to follow at each level to reach that leaf node from the root.

A punctured pseudorandom function (PPRF) is a PRF such that for an input x and PRF key k, a *punctured* key $k\{x\}$ exists which allows for evaluating the function at every point except for x. A PPRF can be constructed from a GGM tree as following. The PRF key is the root node of the tree which can evaluate the PRF on any input. Assume P_x represents the path from root to the leaf node x to be punctured, and N_x are the neighbors to nodes in P_x. The punctured key can be set as the values of all the nodes in N_x. In Figure 3, the binary tree on the right is punctured at point α, and P_α are represented as white squares. The neighboring nodes N_α representing the punctured key are shown as hashed (blue and red) squares. Having N_α, one can evaluate the tree at all points except α, by expanding the nodes shown as gray squares.

C. Learning Parity with Noise

Silent OT constructions rely on variants of the Learning Parity with Noise (LPN) assumption [29]. Here, we briefly discuss the (equivalent [21]) dual and primal variants of LPN which are used in SOT and Ferret respectively.

a) Dual-LPN: The dual-LPN assumption states that the dot product of a public matrix with a random sparse vector is computationally indistinguishable (expressed with $\overset{c}{\approx}$) from random. This assumption can be expressed as $(\mathbf{H}, \mathbf{H}\cdot\mathbf{e}) \overset{c}{\approx} (\mathbf{H}, \mathbf{r})$ where $\mathbf{H} \in \mathbb{F}_2^{n,m}$ is a random matrix ($m > n$), $\mathbf{e} \in \mathbb{F}_2^m$ is a random sparse vector with fixed hamming weight t, and $\mathbf{r} \in \mathbb{F}_2^n$ is a uniform vector.

b) Primal-LPN: The primal-LPN assumption states that the noisy dot product of a public matrix with a dense vector is computationally indistinguishable from random. This assumption can be expressed as $(\mathbf{A}, \mathbf{A}\cdot\mathbf{s}+\mathbf{e}) \overset{c}{\approx} (\mathbf{A}, \mathbf{r})$ where $\mathbf{A} \in \mathbb{F}_2^{n,k}$ is a random matrix, $\mathbf{s} \in \mathbb{F}_2^k$ is a dense short random vector, and $\mathbf{e} \in \mathbb{F}_2^n$ is random sparse vector with fixed hamming weight t, and $\mathbf{r} \in \mathbb{F}_2^n$ is a uniform vector.

D. Silent OT Extension

In this section, we briefly introduce SOT and Ferret protocols based on the building blocks described.

1) SOT Protocol: The construction of SOT [21] is based on three main steps: base OT, tree expansion, and dual-LPN. We provide details of these steps for sender and receiver parties below for generating n number of RCOTs.

a) Base OT: As depicted in Figure 1, the sender generates two vectors of random values $(\mathbf{r_0}, \mathbf{r_1})$ of length $\log n$, where each element is the size of an AES block. The receiver also prepares a random vector of choice bits $\mathbf{c}$ of the same length. The receiver then obtains a vector $\mathbf{r_c}$ from the sender using $\log n$ base OTs where the i^{th} element is picked as the i^{th} element of either $\mathbf{r_0}$ or $\mathbf{r_1}$ according to the choice bit at index i. SOT uses the construction of Chou et al. [30] as their base OT.

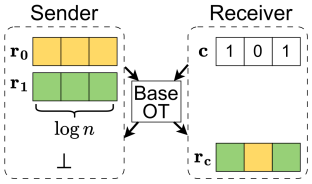


Fig. 1: Base OT consists of the sender picking two random vectors, and receiver obtaining elements according to their random choice bits.

b) Tree expansion: After performing the base OTs, the sender expands a GGM tree with m leaves using AES block cipher to implement the length doubling pseudorandom generator [31], with two keys representing left and right child nodes. The receiver obtains the tree punctured at input α according to her choice bits in the base OT step. The punctured tree can be acquired by receiving the set of neighboring nodes N_α, as described before. To do so, the sender computes two values (L_i, R_i) at i^{th} tree level where L_i is the XOR of all left child nodes, and R_i is the XOR of all right child nodes in that level, as depicted in Figure 3. The sender then computes $L_i \oplus \mathbf{r_0}[i]$ and $R_i \oplus \mathbf{r_1}[i]$ and sends them to the receiver ($\mathbf{r_0}$ and $\mathbf{r_1}$ were generated in base OT). The receiver can unmask one of the two values with $\mathbf{r}[i]$ which they obtained in the base OT step. As Figure 3 shows, the receiver can use a newly obtained node in N_α to expand sub-trees (shown as gray squares), and use the values in the sub-tree to learn the value of the next node in N_α. At the end of this step, the sender samples a random offset $\Delta \in \mathbb{F}_{2^l}$ (where l is the AES input length) and computes the XOR of all leaf values (represented as $\mathbf{v}[i]$) with Δ, namely $\bigoplus_{i\in[m]} \mathbf{v}[i] \oplus \Delta$. The sender then sends this value to the receiver who can compute $\bigoplus_{i\in[m]\setminus\{\alpha\}} \mathbf{v}[i]$ and obtain $\mathbf{v}[\alpha] \oplus \Delta$. The receiver can compute vector $\mathbf{w}$ where $\mathbf{w}[i] = \mathbf{v}[i]\ \forall i \in [m]\setminus\{\alpha\}$, and $\mathbf{w}[\alpha] = \mathbf{v}[\alpha] \oplus \Delta$. At the end of this step, the sender has $\mathbf{v}$ and Δ and the receiver has $\mathbf{w}$ and $\mathbf{e}$ such that $\mathbf{v} \oplus \mathbf{w} = \mathbf{e}\cdot\Delta$, where e is a one-hot vector with a 1 at index α. For brevity, in the description we assumed one punctured point in the tree, equivalent to hamming distance of $t = 1$ for the noise vector in the LPN assumption. For larger values of t, the steps in creating a punctured vector of leaves can be repeated and results added together.

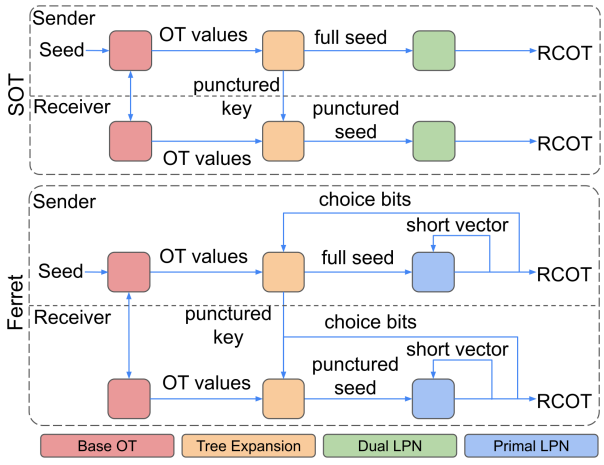


Fig. 2: High level flowchart for SOT (top), and Ferret (bottom) divided between sender and receiver operations. Base OT and tree expansion steps have the same structure in both protocols. SOT uses the dual variant of LPN, while Ferret uses the primal variant, and introduces an iterative approach.

c) LPN: In this step the sender and receiver locally multiply their vectors from the last step with public matrix $\mathbf{H}$. The sender computes $\hat{\mathbf{v}} = \mathbf{H}\cdot\mathbf{v}$, and the receiver computes $\hat{\mathbf{w}} = \mathbf{H}\cdot\mathbf{w}$ and $\hat{\mathbf{e}} = \mathbf{H}\cdot\mathbf{e}$. We have $\hat{\mathbf{v}} \oplus \hat{\mathbf{w}} = \hat{\mathbf{e}}\cdot\Delta$ and $\hat{\mathbf{e}}$ is pseudorandom under the dual-LPN assumption. SOT implements quasi-cyclic codes as a basis for dual LPN [32] to improve computational efficiency. The quasi-cyclic structure of the matrix $\mathbf{H}$ allows for matrix-vector multiplication in quasilinear time using fast Fourier transform as illustrated in Figure 6. Generating additive shares of $\mathbf{e}\cdot\Delta$ in SOT is very efficient, requiring a small communication. The LPN step is also performed locally, but is computationally expensive.

Putting everything together, SOT runs base OT, tree expansion, and LPN operations sequentially. The high level flowchart for main operations in SOT are depicted in Figure 2 (top). The computing blocks perform similar operations on the sender and receiver side to output COTs.

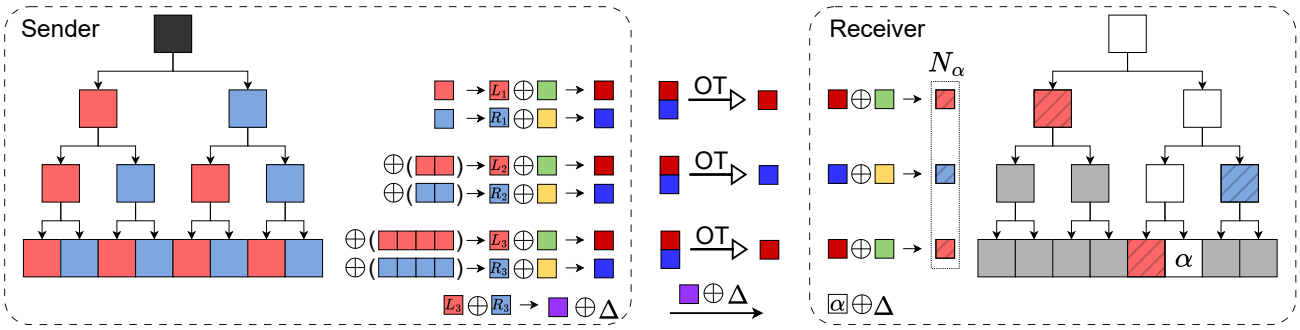


Fig. 3: Tree expansion: the sender expands a GGM tree, and computes the XOR of all the right and left child nodes in each layer i, i.e., L_i and R_i. The sender then XORs these values with $\mathbf{r_0}$ and $\mathbf{r_1}$ (from base OT), and sends the resulting values to the receiver. The receiver is able to unmask the correct values corresponding to the neighbors of path to punctured point α.

2) Ferret Protocol: Ferret [22] builds on the construction of Schoppmann et al. [33] which uses the primal variant of the LPN assumption. We'll describe the protocol of Schoppmann et al. at a high level, and refer the reader to their paper for more details. During initialisation, Ferret uses IKNP as a base protocol to produce a small number of COT. This is further extended via a small iteration of MPCOT tree expansion and primal LPN to obtain a vector of COT just large enough to start the first proper iteration of extension. This is needed as every standard iteration extracts, from the generated COT of the previous iteration, the short vector $\mathbf{s}$ and the hash values for the base OTs at every layer of MPCOT tree expansion. In the first step, sender and receiver follow the base OT and tree expansion steps (similar to SOT) to obtain $\mathbf{d}$ for the sender, and $\mathbf{e}$ for the receiver. In the LPN step, sender and receiver extract k blocks from the COT generated in the previous iteration as vector $\mathbf{r}$ and $\mathbf{s}$ respectively. Finally, sender computes $\mathbf{x} = \mathbf{A} \cdot \mathbf{r} + \mathbf{d}$, and the receiver computes $\mathbf{z} = \mathbf{A} \cdot \mathbf{s} + \mathbf{e}$ for public, sparse random matrix $\mathbf{A}$. $\mathbf{x}$ and $\mathbf{z}$ are pseudorandom according to the primal-LPN assumption. As $\mathbf{r}$ and $\mathbf{s}$ are extracted from previous COT, they are correlated. Given that $\mathbf{d}$ and $\mathbf{e}$ are the outcomes of GGM tree expansion, they are also correlated. With $\mathbf{A}$ being identical, a correlation exists between the resultant vectors $\mathbf{x}$ and $\mathbf{z}$.

The protocol of Schoppmann et al. allows for simpler codes and therefore cheaper computation. However, extra communication is needed to perform k OTs in the second step. Ferret implements optimizations to reduce the communication cost of Schoppmann et al. by introducing an iterative version of the protocol. Specifically, Ferret extends a small number of base OTs (based on IKNP [20]). Part of the generated COTs are then consumed by the application, and the remaining COTs are stored to be used for a next iteration of the protocol during the first and second steps of the Schoppmann protocol. Figure 2 (bottom) depicts the flow of execution in Ferret. After the base OT step, Ferret iterates between tree expansion and LPN operations until the desired number of OTs are generated.

E. Threat Model

The protocols of SOT and Ferret as we described above are secure against a semi-honest adversary, i.e., one who follows the protocol faithfully but tries to extract more information throughout the protocol execution. Accordingly, cuOT provides security against semi-honest attackers. The implementation can be easily extended to protect against malicious adversaries. We note that in the silent OT protocols described, the receiver has no cheating space as altering selection bits corresponds to different punctured coordinates. A malicious sender however, can use values inconsistent with any valid GGM tree or across t executions (for t punctured points) [21]. Correct seed expansion can be verified by validating the final layer of the tree between sender and receiver, and at a high level, is based on having the sender share a hash of its leaf values. We skip description of the malicious security check, and refer the reader to prior work for details [22], [28].

F. Overview on Graphics Processing Units (GPU)

We provide a brief summary of the main GPU features, and refer the readers to other references for more detail. [34]

Single Instruction Multi Thread (SIMT) is an execution model implemented in GPUs to run the same set of instructions on many threads across multiple processors over a wide range of data. In later parts of the paper, we frequently refer to GPU *kernel calls*, which is a scope or execution unit called from the CPU to be executed to the GPU. The GPU task is launched asynchronously from the perspective of the host machine, and explicit host-device synchronization is needed if the algorithm requires GPU kernel execution to complete before moving on with the rest of the program.

Thread blocks are a collection of at most 1024 GPU threads which get assigned to a *stream multiprocessor (SM)*, containing several floating point, integer and tensor cores. Several blocks can be assigned to an SM for execution. Typically, many such blocks are organised to a specified dimension and laid out as a grid, operating on incoming data. The smallest execution unit on a GPU is a *warp*, a collection of 32 GPU threads executing every instruction without needing explicit synchronization. A warp can be assigned to one of the cores contained in an SM, and any smaller number of threads assigned would yield the same runtime as a full warp. Warps should access coalesced memory to reduce memory overhead, and keep useful data cached for optimal performance.

Memory can be classified based on several metrics such as access time and pattern, read or write access, data lifetime,

and capacity. *Register files* are local to each thread and are the quickest to access. *Shared memory* is typically limited to 48kB depending on the configuration and are accessible to threads within the same block. Shared memory lasts for the duration of the block and is important for intra-block inter-thread communication. *Constant memory* is useful for read-only operations from the same memory region, whereas *texture memory* is useful for coalesced memory reads and writes. *Global memory* is the largest and slowest memory, storing universal data for the program lifetime.

III. cuOT Implementation

In this section, we describe cuOT, an efficient GPU implementation of silent OT extensions, targeting SOT and Ferret protocols. We will focus on the main computing blocks, namely, GGM tree expansion, dual-LPN, and primal-LPN.

Several challenges exist when implementing SOT and Ferret on GPU. For one, GPUs do not natively support bit operations such as matrix-vector multiplication in Galois field 2, or binary FFT. Additionally, while several processors provide specialized instructions for popular cryptographic components (e.g., Intel AES-NI [35] for efficient AES encryption and decryption), such operations are not officially supported on the GPU. We propose new GPU implementations for several computation blocks which are not natively supported, and introduce novel interfacing components which allows us to benefit from existing libraries such as NVIDIA cuFFT [36].

A. GGM Tree Expansion

The GGM tree implementation uses two AES blocks with keys corresponding to left and right child nodes. Our AES implementation on the GPU is based on Lee et al. [37], which focuses on optimizing AES through T-tables. In this method, the main computing steps of AES (i.e., sub-byte, shift-row, and mix-column) are pre-computed into four lookup tables. Each round of AES encryption consists of 16 T-table lookups for each AES block, and the lookup index is dependent on the current state. Therefore, many concurrent lookups may conflict and cause slowdown. The implementation of Lee et al. mitigates this problem by storing multiple copies of the T-table so that each concurrent thread gets a dedicated T-table, removing read conflicts in GPUs. We constructed our GGM tree based on this AES implementation, and introduced two optimizations for tree expansion and summation of left and right nodes in each level of the tree as described below.

Coalesced Tree Expansion: Our first optimization is based on the key observation that expanding multiple trees using the same two AES keys is equivalent to combining the root nodes into a single layer and expanding the GGM tree afterwards. For example in Figure 4, two trees (represented as red and yellow nodes) can be expanded together. We combine the root nodes into a coalesced memory block on the GPU as shown in Figure 4 and expand the tree in parallel. The main benefit of this approach is that the GPU threads in a warp access neighboring AES blocks, which reduces striding and memory transaction cost. After expanding each layer in the tree, the left and right child nodes are summed together. To optimize this operation, we keep two separate buffers for the expanded result at each layer (i.e., buffers 1 and 2), where one is used for performing the summations and the other is used for expanding the next layer as depicted in Figure 4. We discuss our receiver implementation of the tree expansion next, followed by our optimized implementation for summation of child nodes.

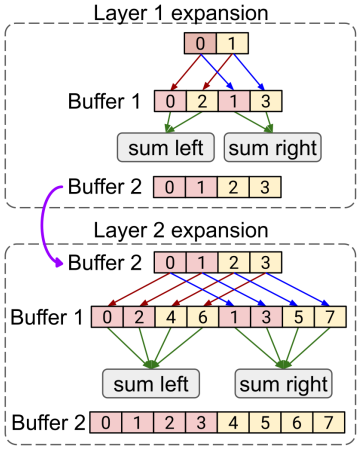


Fig. 4: Coalesced tree expansion.

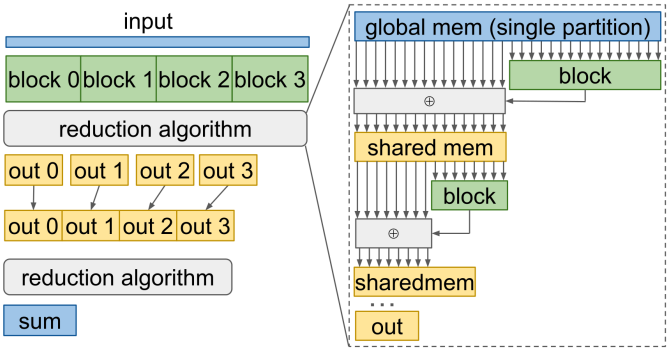
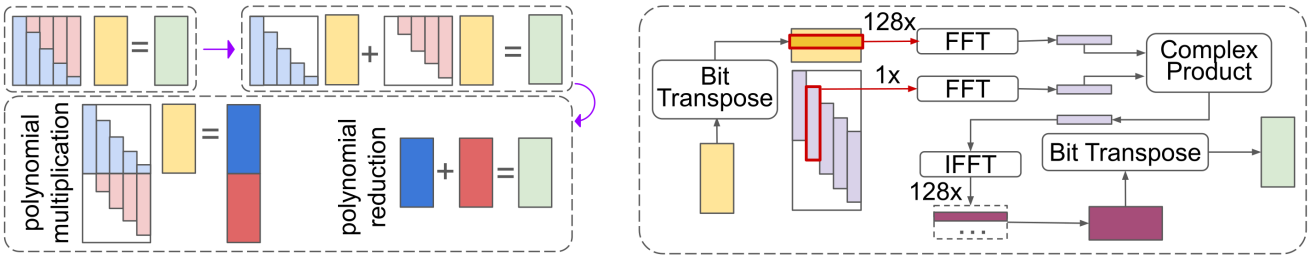


Fig. 5: Iterations of block reduction kernel calls to sum up subtotals (left) and warp synchronized block reduction algorithm within each kernel call (right).

Receiver Punctured Node Correction: Receiver seed expansion is almost identical to the sender, except that the receiver has to correct for the punctured node at every layer by inserting the retrieved node from the sender into the correct position. At every layer, a GPU thread is responsible for tracking the path of punctured nodes for each tree by calculating its index based on the index of the punctured node of the previous layer and the choice bit for the current layer. The receiver obtains the summation of all left and right nodes (L_i and R_i respectively) from the sender, and can unmask the node in N_α in the current layer by subtracting either L_i (or R_i depending on choice bit) from the partial sum of all left (or right) nodes that the receiver has pre-computed (gray squares in Figure 3).

Coalesced Summation of Left and Right Nodes: The sender uses OT to deliver the sum of either the left nodes or the right nodes for every layer of a tree to the receiver to achieve a punctured PRF. We implement an efficient summation through the use of a warp-synchronized, block reduction GPU kernel, which operates on buffer 1 in Figure 4. Block reduction is a common technique as explained in [38], however, to the best of our knowledge there exists no implementation



(a) (b)

Fig. 6: a) Decomposition of quasi-cyclic matrix into polynomial multiplication and reduction b) Extraction of polynomials from transposed vector (output of seed expansion) and quasi-cyclic matrix for multiplication using FFT.

for summation of blocks (e.g. AES blocks) in GF_2, which uses XOR as the operation. As illustrated on the left in Figure 5, each thread block depicted in green is responsible for reducing a share of the input into a 128-bit output. On the right side of Figure 5, we depict the reduction algorithm. As the figure shows, the threads within a block reduce the input data in blue by half in every step. The output of every step is then written to shared memory (similar to [38]) to reduce memory read time for the next halving step. When the remaining data size was greater than 64 AES blocks, the threads require explicit synchronization to prevent quicker scheduled threads from executing the next halving step before all threads are done with the current halving step. Once the data size is reduced to 64 AES blocks, only 32 threads need to do additional work. The 32 threads belong to a single warp and execute every instruction in lockstep, hence requiring no explicit synchronization, which speeds up the last few steps of halving. This process repeats with a decreasing number of threads until a single element remains (out in Figure 5). This value is the subtotal of this thread block, which is packed with the subtotals of other thread blocks as shown on the left of Figure 5 (out_0 to out_3). The packed result blocks, each of which are 128-bit blocks, are then input to the next reduction algorithm call until a single 128-bit block is produced at the end of the entire reduction process as the summation result.

B. Dual LPN

The dual LPN step is based on a *structured* random matrix $\mathbf{H}$, which the sender multiplies with vector $\mathbf{v}$ and the receiver multiplies with their punctured and noise vectors $\mathbf{w}$ and $\mathbf{e}$.

Generation of Quasi Cylic Matrix H: The construction of matrix $\mathbf{H}$ is based on quasi-cyclic codes, where each column is a shifted version of the previous column as depicted in Figure 6(a) above. This means there is only one unique string of n bits used to populate the quasi-cyclic matrix as column values. Therefore, the generation of a quasi-cyclic matrix can be done at very low cost by using AES as the basis of a cipher block algorithm. By AES encrypting an ascending sequence of integers using a common public key known to both parties, they may generate n/128 AES cipher blocks, assuming an AES block is 128-bits in size. The cipher blocks can then be stretched from being AES blocks to a stream of bits, which represents the coefficients of a binary polynomial for the next step, matrix vector multiplication in the form of repeated FFTs.

This matrix can be further decomposed into sums of two triangular matrices as Figure 6(a) on the top shows. We can further transform the matrix by concatenating the two triangular matrices vertically (Figure 6(a) on the bottom). The values in each column and the multiplier vector (shown as yellow) can be interpreted as the coefficients of two polynomials, which allows us to use FFT to accelerate polynomial multiplication. This trick is a well known practice that decreases runtime complexity to $\mathcal{O}(n \log n)$ [39]. The resultant polynomial of length $2n$ can then be reduced to length n via polynomial reduction. To do so, we reapply the block reduction algorithm used for seed expansion to achieve the result depicted in Figure 6(a) on the bottom, whereby the top half of the polynomial is XORed against the bottom half to get the result vector shown in green.

FFT for fast binary polynomial multiplication: Figure 6(b) depicts the detailed operations for the equivalent polynomial multiplication. The multiplier vector (shown as yellow) is transposed into a bit matrix of 128 rows first, where each row is used in the polynomial multiplication using FFT. The second polynomial is obtained from the columns of the concatenated matrix. CUDA offers a built-in library, cuFFT [36], for FFT operations. However, cuFFT only receives floating points as inputs. This requires a translation from bit to float representation, resulting in increased memory usage and strided memory access. On the other hand, cuFFT offers a batching feature, allowing 128 FFTs to be executed in groups which significantly reduces the amortised cost. To benefit from cuFFT, we convert the two input polynomials into float vectors of length n. Both vectors are then input to FFT, the outputs of which are two n length complex type vectors (Figure 6(b)). CUDA implements complex numbers as a struct of two floats, representing real and imaginary. A pair of complex type vectors are the FFT representations of the two initial polynomials and are multiplied point-wise to obtain the result. While cuFFT provides a complex product intrinsic, our evaluations show that it is considerably slow due to other built-in operations after the complex product is done. Therefore, we implement an optimized custom complex multiplier as follows. For two input polynomials in FFT representation, $\mathbf{a}$ and $\mathbf{b}$, the complex

product is implemented as $\mathbf{x}_{real} = \mathbf{a}_{real} \times \mathbf{b}_{real} - \mathbf{a}_{img} \times \mathbf{b}_{img}$ and $\mathbf{x}_{img} = \mathbf{a}_{real} \times \mathbf{b}_{img} + \mathbf{a}_{img} \times \mathbf{b}_{real}$. The resulting complex type product $\mathbf{x}$ is then passed through inverse FFT (based on cuFFT), generating a vector of real values of length $2n$. As a design choice for cuFFT, these floats are the coefficients of the product polynomial multiplied by FFT size, which is $2n$ [36]. Therefore, the result has to be divided by the FFT size $\mod 2$ to obtain the actual coefficients. During these float operations, inaccuracies will occur since floats cannot represent large integers with full precision. To avoid such inaccuracies, we first round the undivided raw output to the nearest multiple of the FFT size (before dividing by the FFT size), and obtain a whole number representing the coefficient. The coefficients of the product polynomial are always whole integers because the coefficients of the input polynomials are binary. We use a quasi-cyclic code dimension of $n \times 2n$ where n is the input size of the polynomial. This setting allows us to make use of cuFFT optimizations for power of 2 input sizes. We implement a custom kernel which casts the first half of the IFFT result (i.e., the coefficients of the product polynomial) from float to integer before reducing by modulo 2 and writing it in bit form into the result. This kernel then takes the second half of the coefficients of the product polynomial and performs the same type casting and modulo reduction before XORing the contents into the values previously stored in the result. Finally, we note that SOT requires larger device memory usage due to the buffers used for batched FFTs. We place CUDA memory allocations in the initialization phase of the protocol to avoid allocation overhead during time critical steps.

Fast Bit Transpose of a 128-bit Block Matrix: The smallest accessible data type on the GPU is a byte, which after bit-transpose, contributes to the same positioned bits scattered across 8 rows in the transposed matrix. As visualized in Figure 7(a), we define the atomic workload of this operation as the transposing of an 8×8 bit tile in the original matrix and writing the result into the appropriate tile in the output. Our implementation is inspired by the algorithm proposed by Warren [40]. We assign an 8×8 bit tile in the matrix to a thread for local bit transpose, before writing the result into the transposed matrix. The tiling ensures memory coalescing, as neighboring threads in a warp will iterate through neighboring columns of the same row across eight rows.

The transpose algorithm is depicted in Figure 7b, where an extracted tile is laid out as a 64 bit value. The protocol proceeds in two phases. In phase one, several iterations are performed where each iteration ANDs a formatted hexadecimal against the 64 bit tile. The first iteration extracts bit 0 of byte 0, bit 1 of byte 1, and so on. The extracted bits are written to the 64-bit tile with no bit shifting as the extracted bits reside along the diagonal of the tile (Figure 7b), making their positions fixed post transpose. The next iteration extracts a different set of bits and left shifts the values by 7 bits before placing it into the result tile (using an OR). This process iterates 8 times, each time shifting the bits by a multiple of 7. At the end of this phase, the top right triangle in the output matrix is populated. To complete the matrix, we perform similar steps

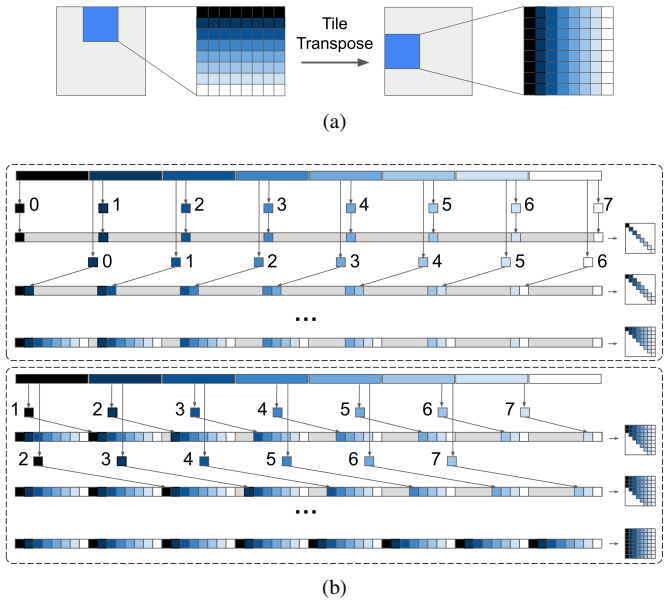


Fig. 7: a) Tiling the input matrix for local bit transpose and writing the result into correct tile in output matrix b) Extraction of bits from the input tile and bit shift OR into output tile.

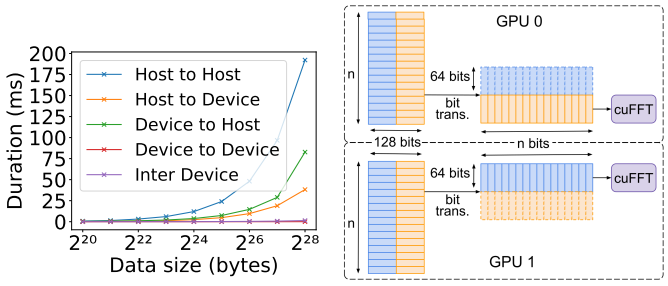
in the next phase, but instead shift the extracted bits to the right by an increasing multiple of 7. The final 64-bit output holds the transposed tile, which is written into the destination tile in the output matrix. The GPU kernel grid dimension is set to the dimension of the input matrix so that the threads index into corresponding tiles.

C. Primal LPN

As described in Section II-C, the primal LPN variant requires multiplying a random matrix $\mathbf{A}$ with a short dense vector $\mathbf{s}$. As opposed to the dual LPN matrix which is based on quasi-cyclic codes (resulting in repeated columns which can be efficiently generated), the matrix in primal LPN is populated by uniform random numbers. Here, we describe our optimizations for primal LPN implementation.

Random Indices Generation: The uniform generation of random numbers that populate the random matrix $\mathbf{A}$ is implemented through AES. This random matrix is of size $n \times k$, where each row has 10 non-zero values (parameter set by Ferret [22]). Due to the sparsity of this matrix, we instead represent it as an $n \times 10$ matrix containing the positions of the aforementioned non-zero values to save memory. Unlike the quasi-cyclic matrix, the matrix in primal-LPN has the effect of expanding the input vector $\mathbf{s}$ of length k into a longer vector of size n. While the CPU implementation uses the AES-NI instruction to generate random numbers as needed, we implement a more efficient function on the GPU that can generate all required random numbers at once. In our function, the first GPU operation populates the empty matrix with an incrementing counter value, and the next operation performs batched AES encryption on the matrix.

Row Vector Multiplication: We implement the multiplication of random matrix $\mathbf{A}$ with short vector $\mathbf{s}$ through a kernel call.



(a) (b)

Fig. 8: (a) CUDA API memory copy runtime, (b) Transpose of redundant expanded vectors and matrix slicing to execute a share of FFT on each GPU.

Each thread is assigned a row, and computes the LPN output vector element corresponding to the row assigned. Each thread iterates through the randomly generated 128-bit values and fetches elements from the short vector corresponding to the position indicated by the 128-bit values. We optimize the final step by integrating the addition (using XOR) of noise vector with the row vector multiplication instead of performing them as two separate steps. Afterwards, we obtain the output value for the primal LPN.

D. Generating COTs from RCOTs

The choice bits in the RCOT variant are random. However, many applications require one of the parties to pick the choice bits (COT variant). The result of the entire workflow up until LPN is the vector $\mathbf{z} = \mathbf{A} \cdot \mathbf{s} + \mathbf{e}$, which is randomly correlated with the vector $\mathbf{z}$ which the other party holds. To obtain COT from RCOT, the receiver retrieves the least significant bit of every block residing in vector $\mathbf{z}$ (i.e., the random bits). The receiver then XORs the random bits with the choice bits. The result is a difference vector which is sent to the sender. Note that this difference vector does not reveal any information about the receiver's choice or random bits. The sender makes changes to its vector $\mathbf{z}$ as following. If the difference bit is zero, the sender does not need to do anything and the sender and receiver's block are identical. If the difference bit is one, the sender XORs the corresponding block with Δ [22].

Our contribution to this procedure consists of two parts. First, we implement the GPU operation of the XOR between the choice bits and random bits at the receiver side. Second, we implement the XOR operation between the difference vector and the expanded vector $\mathbf{z}$ at the sender side. While the implementations are somewhat trivial, we incorporate several optimizations. Our implementation includes spawning the appropriate number of thread blocks, with each thread performing XOR for four 32-bit values to ensure balance between efficiency and overhead of spawning new GPU threads.

E. Multi-GPU Implementation

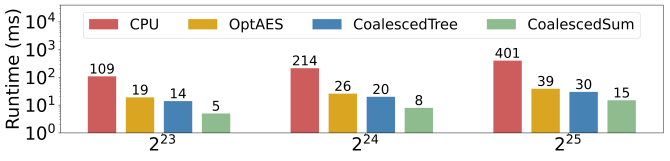
a) GGM Tree Division: For Ferret, our multi-GPU implementation of the tree expansion divides the trees evenly among available GPUs. The tree count for seed expansion may not always be a multiple of the number of GPUs available, in which case there may be minor load imbalance between GPUs. Intuitively, in SOT, tree expansion could be done similarly. However, LPN compression requires 128 FFTs to be executed on the transposed expanded tree. This means that after tree expansion, all GPUs hold a portion of each polynomial. However, the FFT operation works best when executed on the entirety of a polynomial at once, instead of executing on several parts of the polynomial and reconciling the results, which requires data exchanges between GPUs. Our measurements show that inter-GPU memory copy takes 4-5$\times$ longer to complete than memory copy within the same GPU, as shown in Figure 8(a). With data transfers occurring between every GPU pair, this will cause an inter-GPU memory copy time complexity of $O(n^2)$. Instead, cuOT redundantly expands the seed across all GPUs and selects the data needed to execute a share of FFTs on each GPU. Additionally, tree expansion in SOT is a small part of the total runtime as we will show in our evaluations later, which leads us to focus on implementing a multi-GPU LPN operation, as described next.

b) Parallel FFT Execution: Our multi-GPU implementation of LPN in SOT consists of distributing the 128 FFTs across the GPUs. Each GPU transposes the full expanded vector and extracts a share of the rows in the transposed matrix to process as depicted in Figure 8(b). The vector shown in blue and orange are the expanded vectors of AES blocks (output of tree expansion). After transpose, GPU 0 extracts the blue region corresponding to 64 FFTs while GPU 1 extracts the orange region corresponding to the other half. Due to all GPUs having all 128 polynomials , we circumvent any need for inter-GPU memory copy.

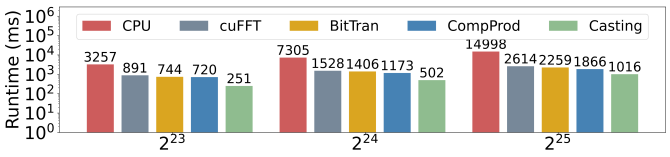
c) Primal-LPN Workload Division: The sum of products between the elements in a row of the random matrix $\mathbf{A}$ in primal LPN and the elements of the short vector $\mathbf{s}$ are done sequentially when iterating through the index of the non-zero values in each row of the random matrix. Therefore, with each row having an equal number of non-zero values, the rows can be partitioned evenly across available GPUs, with each GPU also containing their own copy of the short vector $\mathbf{s}$ and a share of the long expanded noise vector $\mathbf{e}$. In primal LPN, the scope of the expanded seed held by a GPU corresponds to a range of rows in the random matrix. This means no memory exchanges between GPUs are needed throughout the seed expansion and primal LPN, as each GPU locally computes the product of a share of the random matrix and the short vector and summation with a share of the expanded vector.

IV. Evaluation

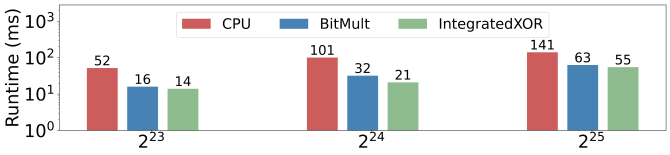
In this section, we evaluate the performance of cuOT, and compare it to existing implementations of silent OT extensions on the CPU. All GPU experiments were performed on (1 to 8) NVIDIA Tesla V100 GPU cards with CUDA version 12.4. Each GPU has 32GB of memory, SM count of 84, and 5376 INT32 cores. CPU experiments were performed on an Intel Xeon Gold 6248 @2.5GHz with 791GB of memory and up to 32 threads. SOT evaluations were performed based on



(a) Tree Exp.



(b) Dual LPN



(c) Primal LPN

Fig. 9: Microbenchmarks for tree expansion, dual LPN, and primal LPN when generating $\{2^{23}, 2^{24}, 2^{25}\}$ number of OTs with successive optimizations applied to the GPU implementation and compared against a baseline CPU.

LibOTe [41] library, and Ferret evaluations were performed based on EMP-OT [42] library. We present evaluation results for accelerating main computational components, i.e., GGM tree expansion and LPN steps.

a) Parameter Selection: For both CPU and GPU, an SOT run contains 8 parallel GGM trees for tree expansion. The quasi-cyclic code matrix $\mathbf{H}$ size is set to be n, the requested number of OTs. In Ferret, the length of the short vector used in primal LPN was based on the cache size on the platform to fully cache the private vector for quick memory access. We use the same parameters for tree count, tree depth and short vector length as emp-ot to provide the same security guarantees. Specifically, we use $\texttt{ferret}_{\texttt{b13}}$ parameters for benchmarking, which is a set of predefined parameters for tree expansion and primal LPN. In this set, n is the size of the expanded vector, t is the number of trees, d is the depth per tree, and k is the size of the short vector for primal LPN. n and k are parameters measured in units of AES blocks. During Ferret iterative extensions, n = 10485760, t = 1280, k = 452000, and d = 13 [22].

A. cuOT *Microbenchmarks*

In this section, we provide microbenchmarks on tree expansion, dual LPN, and primal LPN steps, and compare the single GPU runtime with successive optimizations against a baseline CPU implementation. Figure 9a shows the runtime of GPU tree expansion with successive optimizations on AES implementation, coalesced tree, and coalesced summation as described in Section III-A. Compared to the CPU baseline, cuOT optimizations cumulatively achieve between $21\times - 26\times$ speedup for the tree expansion step. In Figure 9b, we demonstrate the effects of our proposed optimizations for the dual LPN step, successively evaluating cuOT's implementation based on cuFFT, optimized bit transpose, complex product, and type casting as described in Section III-B. As observed from the figure, cuOT cumulative optimizations result in between $12.9\times - 14.7\times$ speedup over the CPU baseline for the dual LPN step. Finally, we present microbenchmarks for the primal LPN step in Figure 9c, where we compare the baseline CPU runtime with our GPU implementation using bit shifts for sparse matrix and dense vector multiplication, as well as integrated XOR optimization as described in Section III-C. cuOT optimizations result in up to $2.5\times - 4.8\times$ speedup over baseline CPU for the primal LPN step.

TABLE I: Comparing cuOT and naive GPU runtime in ms.

Steps	CPU	Naive GPU	cuOT
Tree Exp.	215	37	8
Dual LPN	7306	89636	502
Primal LPN	101	437	22

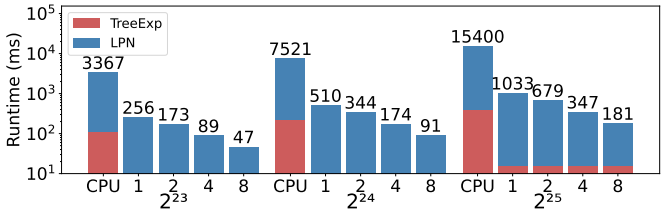
We also compare cuOT against a naive GPU version following the CPU implementation and using off-the-shelf CUDA libraries. The naive implementation for the tree expansion follows the CPU steps closely, including mirroring the AES implementation on the CPU. For dual LPN, the naive GPU implementation uses a specialized library for polynomial multiplication [43] and complex number product based on cuBLAS [44]. Finally, the naive GPU implementation for primal LPN relies on cuSparse [45] to represent the public random matrix in compressed sparse row (CSR) format for matrix vector multiplication and noise addition. We present the results in Table I. As the table shows, the naive GPU implementation is not always faster than even the CPU implementation, which emphasizes the role of GPU specific design and optimization to achieve speedups. Compared to the naive GPU implementations, cuOT achieves $3.8\times$, $182\times$, and $81\times$ runtime speedup for tree expansion, dual LPN, and primal LPN steps respectively.

B. cuOT *End-to-End Performance*

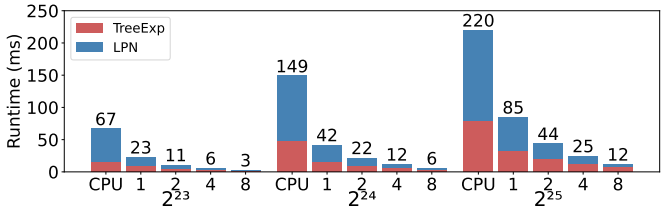
Figure 10a presents our evaluation results for cuOT SOT implementation for generating $n = \{2^{23}, 2^{24}, 2^{25}\}$ number of OTs. We present results for both single and multi-GPU implementations, and compare results to CPU runtime. GGM tree expansion and LPN operations on 1 GPU obtain up to $25\times$ and $15\times$ speedup respectively compared to the CPU implementation. The speedups get more significant for larger number of OTs generated. The end-to-end runtime of the 1 GPU implementation is up to $14\times$ faster than CPU.

We also present the multi-GPU runtime of cuOT SOT over $\{2, 4, 8\}$ GPUs. Our parallelized implementation scales efficiently as the number of GPUs increase, gaining close to $2\times$ improvement when the number of GPUs double. These results demonstrate the scalability of our FFT component of the LPN operation. The end-to-end runtime achieves $11\times$, $22\times$, $41\times$ and $75\times$ speedup over 8 threaded CPU implementation for 1, 2, 4, and 8 GPUs.

Figure 10b presents our evaluation results for cuOT Ferret implementation for generating the same n number of OTs. GGM tree expansion and LPN operations on 1 GPU obtain up to $3\times$ and $4\times$ speedup respectively compared to the CPU implementation on 32 threads. Similar to SOT, the speedups get more significant as n grows. The end-to-end runtime of the 1 GPU implementation is up to $4\times$ faster than CPU.



(a) SOT



(b) Ferret

Fig. 10: SOT and Ferret runtimes for generating $\{2^{23}, 2^{24}, 2^{25}\}$ number of OTs, scaling with respect to number of GPUs, indicated by x-ticks (1,2,4,8).

We also evaluate our framework when integrated into Cheetah [10] for private inference. cuOT results in 42% reduction in non-linear layer runtime for the DenseNet121 benchmark. As previously discussed, non-linear components take up 60% of total runtime for private inference and the underlying protocols have not been targeted for acceleration in prior work.

While Ferret enjoys a lower computation cost compared to SOT, we note that the overall communication cost of Ferret is higher than SOT. The communication cost over the entire execution of SOT and Ferret are reported in Table II for $n = \{2^{23}, 2^{24}, 2^{25}\}$ number of OTs. As these experiments show, Ferret's communication cost is up to $200\times$ higher than SOT. Therefore, for higher bandwidth scenarios when a lot of computing resources are not available, Ferret is the preferred option. Otherwise, under for lower bandwidth settings or when a lot of computing power is available, SOT outperforms Ferret.

n	SOT	Ferret
2^{23}	12512	1504521
2^{24}	13056	2057497
2^{25}	13600	3163449

TABLE II: Total communication cost in Bytes.

C. Discussion

Our evaluations demonstrate the superior performance of cuOT over software-based implementations. However, the downside of hardware acceleration is that the library could potentially leak sensitive information through *side-channel attacks* (SCA) more easily which measure physical quantities such as consumed time or power. In recent work, Berti et al. [46] proposed a formalized notion for leakage-resilient OT, focusing on protecting the sender's state. Berti et al. argue that by using a leakage-resistant block cipher (in the final step of OT for exchanging messages), an adversary exploiting SCAs on OT can only obtain the encryption of two messages which reveal no information about the messages themselves. Therefore, the OT protocol does not need additional protection against SCAs. Other threat models beyond semi-honest parties which consider corrupted sender or receiver are more difficult to address, and are interesting directions for future work.

V. Related Work

OT extension advancements: The importance of efficiently generating many OTs has resulted in a line of research on OT extensions, beginning with the seminal work of [20], which generate near-arbitrary number of OTs from a small number of base OTs. The latest generation of OT extension protocols are *silent OT* constructions [22], [28], which after a short interaction can extend OTs with only local computation. Other work in this area, e.g., Expand-Accumulate [47] and Expand-Convolute [48], focus on creating LPN-friendly codes to improve performance.

OT accelerators: We are unaware of any work that accelerates silent OT constructions. Frederiksen [49] propose a two-party secure computation framework based on GCs and accelerate it on the GPU. This work accelerates the older OT construction of Nielsen et al. [50]. While we were not able to directly compare cuOT to this work (source code unavailable and OT benchmarks are not presented in the paper), we note that SOT and Ferret significantly improve over prior OT constructions. For example, compared to Keller et al. [51] (which improves over the OT protocol of Nielsen), SOT and Ferret reduce communication by 1000x and 170x and gain runtime speedup of up to 60x and 140x respectively (Table 2 in [22]). In this work, we present the first GPU acceleration of silent OT constructions based on SOT and Ferret.

Other cryptographic primitive accelerators: Prior work has explored accelerating other cryptographic primitives, such as AES [37], [52]–[54], hash functions [55]–[57], public-key primitives [55], [57], GCs [4], and HE [5], [6], [15]–[18], [58] on hardware, FPGAs, and GPUs. Popularity of PPML applications in particular has contributed to the recent interest in accelerating homomorphic encryption (used for evaluating linear layers) on the GPU [6], [16], [18]. However, acceleration of the OT protocol (used for evaluating non-linear layers) on the GPU has not been studied in prior work.

VI. Conclusion

This work presented the first GPU library, cuOT, for implementing oblivious transfer extensions. We presented custom built modules for operations such as bit transpose, block reduction, GGM tree expansion, and LPN. Our results showed that cuOT targeted components and end-to-end optimizations improve OT generation runtime by up to $75\times$ (for SOT)

and $15\times$ (for Ferret). Our results demonstrate the potential of widely available hardware infrastructure such as GPUs in reducing the cost of privacy-preserving computing for practical deployments. We believe that there is growing demand for native support of several cryptographic primitives and computing blocks on CUDA, and such architectural support can open a new range of possibilities for the future of secure computation in emerging applications.

REFERENCES

[1] A. A. Ouallane, A. Bahnasse, A. Bakali, and M. Talea, "Overview of road traffic management solutions based on iot and ai," *Procedia Computer Science*, vol. 198, pp. 518–523, 2022.

[2] [Online]. Available: https://www.govinfo.gov/content/pkg/BILLS-104s1028is/pdf/BILLS104s1028is.pdf

[3] [Online]. Available: https://cppa.ca.gov/regulations/pdf/cppa_act.pdf

[4] J. Mo, J. Gopinath, and B. Reagen, "Haac: A hardware-software co-design to accelerate garbled circuits," in *Proceedings of thehttps://github.com/starry91/parallel-FFT 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–13.

[5] M.-S. Chen, C.-M. Cheng, P.-C. Kuo, W.-D. Li, and B.-Y. Yang, "Faster multiplication for long binary polynomials," *arXiv preprint arXiv:1708.09746*, 2017.

[6] W. Dai and B. Sunar, "cuhe: A homomorphic encryption accelerator library," in *Cryptography and Ihttps://github.com/starry91/parallel-FFTnformation Security in the Balkans: Second International Conference, BalkanCryptSec 2015, Koper, Slovenia, September 3-4, 2015, Revised Selected Papers 2*. Springer, 2016, pp. 169–186.

[7] O. Goldreich, S. Goldwasser, and S. Micali, "How to construct random functions," *Journal of the ACM (JACM)*, vol. 33, no. 4, pp. 792–807, 1986.

[8] D. Rathee, M. Rathee, N. Kumar, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma, "Cryptflow2: Practical 2-party secure inference," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 325–342.

[9] Z. Ghodsi, N. K. Jha, B. Reagen, and S. Garg, "Circa: Stochastic relus for private deep learning," *Advances in Neural Information Processing Systems*, vol. 34, pp. 2241–2252, 2021.

[10] Z. Huang, W.-j. Lu, C. Hong, and J. Ding, "Cheetah: Lean and fast secure {Two-Party} deep neural network inference," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 809–826.

[11] K. Garimella, Z. Ghodsi, N. K. Jha, S. Garg, and B. Reagen, "Characterizing and optimizing end-to-end systems for private inference," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2023, pp. 89–104.

[12] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

[13] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4700–4708.

[14] Y. Doröz, E. Öztürk, and B. Sunar, "Accelerating fully homomorphic encryption in hardware," *IEEE Transactions on Computers*, vol. 64, no. 6, pp. 1509–1521, 2014.

[15] B. Reagen, W.-S. Choi, Y. Ko, V. T. Lee, H.-H. S. Lee, G.-Y. Wei, and D. Brooks, "Cheetah: Optimizing and accelerating homomorphic encryption for private inference," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 26–39.

[16] W. Wang, Y. Hu, L. Chen, X. Huang, and B. Sunar, "Accelerating fully homomorphic encryption using gpu," in *2012 IEEE conference on high performance extreme computing*. IEEE, 2012, pp. 1–5.

[17] A. Al Badawi, B. Veeravalli, C. F. Mun, and K. M. M. Aung, "High-performance fv somewhat homomorphic encryption on gpus: An implementation using cuda," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 70–95, 2018.

[18] W. Jung, S. Kim, J. H. Ahn, J. H. Cheon, and Y. Lee, "Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with gpus," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 114–148, 2021.

[19] C. Gouert, V. Joseph, S. Dalton, C. Augonnet, M. Garland, and N. G. Tsoutsos, "Arctyrex: Accelerated encrypted execution of general-purpose applications," *arXiv preprint arXiv:2306.11006*, 2023.

[20] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank, "Extending oblivious transfers efficiently," in *Annual International Cryptology Conference*. Springer, 2003, pp. 145–161.

[21] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, P. Rindal, and P. Scholl, "Efficient two-round ot extension and silent non-interactive secure computation," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 291–308.

[22] K. Yang, C. Weng, X. Lan, J. Zhang, and X. Wang, "Ferret: Fast extension for correlated ot with small communication," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1607–1626.

[23] A. C.-C. Yao, "How to generate and exchange secrets," in *27th annual symposium on foundations of computer science (Sfcs 1986)*. IEEE, 1986, pp. 162–167.

[24] O. Goldreich, S. Micali, and A. Wigderson, "How to play any mental game, or a completeness theorem for protocols with honest majority," in *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*, 2019, pp. 307–328.

[25] M. Naor and B. Pinkas, "Oblivious transfer and polynomial evaluation," in *Proceedings of the thirty-first annual ACM symposium on Theory of computing*, 1999, pp. 245–254.

[26] R. Impagliazzo and S. Rudich, "Limits on the provable consequences of one-way permutations," in *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, 1989, pp. 44–61.

[27] D. Beaver, "Correlated pseudorandomness and the complexity of private computations," in *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, 1996, pp. 479–488.

[28] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, and P. Scholl, "Efficient pseudorandom correlation generators: Silent ot extension and more," in *Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part III 39*. Springer, 2019, pp. 489–518.

[29] A. Blum, M. Furst, M. Kearns, and R. J. Lipton, "Cryptographic primitives based on hard learning problems," in *Annual International Cryptology Conference*. Springer, 1993, pp. 278–291.

[30] T. Chou and C. Orlandi, "The simplest protocol for oblivious transfer," in *Progress in Cryptology–LATINCRYPT 2015: 4th International Conference on Cryptology and Information Security in Latin America, Guadalajara, Mexico, August 23-26, 2015, Proceedings 4*. Springer, 2015, pp. 40–58.

[31] C. Guo, J. Katz, X. Wang, and Y. Yu, "Efficient and secure multiparty computation from fixed-key block ciphers," Cryptology ePrint Archive, Paper 2019/074, 2019. [Online]. Available: https://eprint.iacr.org/2019/074

[32] C. Aguilar, O. Blazy, J.-C. Deneuville, P. Gaborit, and G. Zémor, "Efficient encryption from random quasi-cyclic codes," Cryptology ePrint Archive, Paper 2016/1194, 2016. [Online]. Available: https://eprint.iacr.org/2016/1194

[33] P. Schoppmann, A. Gascón, L. Reichert, and M. Raykova, "Distributed vector-ole: Improved constructions and implementation," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1055–1072.

[34] "Cuda c++ programming guide." [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[35] K. Akdemir, M. Dixon, W. Feghali, P. Fay, V. Gopal, J. Guilford, E. Ozturk, G. Wolrich, and R. Zohar, "Breakthrough aes performance with intel aes new instructions," *White paper, June*, vol. 12, p. 217, 2010.

[36] NVIDIA, "cuFFT library." [Online]. Available: https://docs.nvidia.com/cuda/cufft/contents.html

[37] W.-K. Lee, H. J. Seo, S. C. Seo, and S. O. Hwang, "Efficient Implementation of AES-CTR and AES-ECB on GPUs With Applications for High-Speed FrodoKEM and Exhaustive Key Search," *IEEE Transactions on Circuits and Systems II: Express Briefs*, pp. 2962–2966, Jun. 2022.

[38] M. Harris, "Optimizing parallel reduction in cuda." [Online]. Available: https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

[39] A. Townsend, "Matrix-vector multiplication using the fft." [Online]. Available: https://math.mit.edu/icg/resources/teaching/18.085-spring2015/toeplitz.pdf

[40] H. Warren, "Hacker's delight." Upper Saddle River: Addison-Wesley, 2013.

[41] P. Rindal, "libote." [Online]. Available: https://github.com/osu-crypto/libOTe
[42] "Emp-ot." [Online]. Available: https://github.com/emp-toolkit/emp-ot
[43] starry91, "Parallel-fft." [Online]. Available: https://github.com/starry91/parallel-FFT
[44] Nvidia, "cublas library." [Online]. Available: https://docs.nvidia.com/cuda/cublas/index.html
[45] ——, "cusparse library." [Online]. Available: https://docs.nvidia.com/cuda/cusparse/
[46] F. Berti, C. Hazay, and I. Levi, "Lr-ot: leakage-resilient oblivious transfer," in *International Conference on Security and Cryptography for Networks*. Springer, 2024, pp. 182–204.
[47] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, N. Resch, and P. Scholl, "Correlated pseudorandomness from expand-accumulate codes," Cryptology ePrint Archive, Paper 2022/1014, 2022. [Online]. Available: https://eprint.iacr.org/2022/1014
[48] S. Raghuraman, P. Rindal, and T. Tanguy, "Expand-convolute codes for pseudorandom correlation generators from lpn," Cryptology ePrint Archive, Paper 2023/882, 2023. [Online]. Available: https://eprint.iacr.org/2023/882
[49] T. K. Frederiksen and J. B. Nielsen, "Fast and maliciously secure two-party computation using the gpu," *International Association for Cryptologic Research*, 2013.
[50] J. B. Nielsen, P. S. Nordholt, C. Orlandi, and S. S. Burra, "A new approach to practical active-secure two-party computation," in *Annual Cryptology Conference*. Springer, 2012, pp. 681–700.
[51] M. Keller, E. Orsini, and P. Scholl, "Actively secure ot extension with optimal overhead," in *Annual Cryptology Conference*. Springer, 2015, pp. 724–741.
[52] S. A. Manavski, "Cuda compatible gpu as an efficient hardware accelerator for aes cryptography," in *2007 IEEE International Conference on Signal Processing and Communications*. IEEE, 2007, pp. 65–68.
[53] O. Harrison and J. Waldron, "Efficient acceleration of asymmetric cryptography on graphics hardware," in *Progress in Cryptology–AFRICACRYPT 2009: Second International Conference on Cryptology in Africa, Gammarth, Tunisia, June 21-25, 2009. Proceedings 2*. Springer, 2009, pp. 350–367.
[54] A. Gielata, P. Russek, and K. Wiatr, "Aes hardware implementation in fpga for algorithm acceleration purpose," in *2008 International Conference on Signals and Electronic Systems*. IEEE, 2008, pp. 137–140.
[55] M. Khalil-Hani, V. P. Nambiar, and M. Marsono, "Hardware acceleration of openssl cryptographic functions for high-performance internet security," in *2010 International Conference on Intelligent Systems, Modelling and Simulation*. IEEE, 2010, pp. 374–379.
[56] A. A. Fairouz, M. Abusultan, V. V. Fedorov, and S. P. Khatri, "Hardware acceleration of hash operations in modern microprocessors," *IEEE Transactions on Computers*, vol. 70, no. 9, pp. 1412–1426, 2020.
[57] H. Bensalem, Y. Blaquière, and Y. Savaria, "Acceleration of the secure hash algorithm-256 (sha-256) on an fpga-cpu cluster using opencl," in *2021 IEEE international symposium on circuits and systems (ISCAS)*. IEEE, 2021, pp. 1–5.
[58] K. Shivdikar, G. Jonatan, E. Mora, N. Livesay, R. Agrawal, A. Joshi, J. L. Abellán, J. Kim, and D. Kaeli, "Accelerating polynomial multiplication for homomorphic encryption on gpus," in *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*. IEEE, 2022, pp. 61–72.