cuXCMP: CUDA-Accelerated Private Comparison Based on Homomorphic Encryption

Hao Yang[1], Shiyu Shen[2,4], Zhe Liu[3,1], and Yunlei Zhao[2,4]

[1] College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China
[2] School of Computer Science, Fudan University, Shanghai, China
[3] Research Institute of Basic Theories, Zhejiang lab, Hangzhou, China
[4] State Key Laboratory of Cryptology, Beijing

Abstract. Private comparison schemes constructed on homomorphic encryption offer the noninteractive, output expressive and parallelizable features, and have advantages in communication bandwidth and performance. In this paper, we propose cuXCMP, which allows negative and float inputs, offers fully output expressive feature, and is more extensible and practical compared to XCMP (AsiaCCS 2018). Meanwhile, we introduce several memory-centric optimizations of the constant term extraction kernel tailored for CUDA-enabled GPUs. Firstly, we fully utilize the shared memory and present compact GPU implementations of NTT and INTT using a single block; Secondly, we fuse multiple kernels into one AKS kernel, which conducts the automorphism and key switching operation, and reduce the grid dimension for better resource usage, data access rate and synchronization. Thirdly, we precisely measure the IO latency and choose an appropriate number of CUDA streams to enable concurrent execution of independent operations, yielding a constant term extraction kernel with perfect latency hide, i.e., CTX. Combining these approaches, we boost the overall execution time to optimum level and the speedup ratio increases with the comparison scales. For one comparison, we speedup the AKS by 23.71×, CTX by 15.58×, and scheme by 1.83× (resp., 18.29×, 11.75×, and 1.42×) compared to C (resp., AVX512) baselines, respectively. For 32 comparisons, our CTX and scheme implementations outperform the C (resp., AVX512) baselines by 112.00× and 1.99× (resp., 81.53× and 1.51×).

Keywords: Private comparison · Homomorphic encryption · GPU optimization · Number theoretic transform · Key switching.

1 Introduction

Decision tree is one of the most popular classification models in machine learning that makes decisions by comparing the input values and the thresholds. To reduce the computational burden, it is often deployed on a cloud server to evaluate the input data, but this raises security issues when the data is privacy sensitive. Privacy preserving decision trees (PPDT) is such a technique that proposed to provide prediction results without compromising the client inputs as well as the tree model except the number of thresholds, thus ensuring the privacy for both parties.

The main building block of PPDT is private comparison, which takes two encrypted numbers as input and outputs a ciphertext that contains the comparison result. Recently, various works concentrate on constructing private comparison schemes, based on techniques such as Yao's garbled circuit [20, 27], secret sharing [7], oblivious transfer [10], and homomorphic encryption (HE). Among these schemes, the first three types feature multi-party computing scenario that the interactions between parties is in nature cannot be omitted, making communication latency the main bottleneck to efficiency. The vulnerability becomes more obvious in settings where network communication is poor and the computation capability of clients is limited. Constructing noninteractive private comparison based on homomorphic encryption, which allows computation over ciphertexts, is more desirable, for it offers parallel evaluation within single-round interaction and can greatly boost the overall execution time.

The first noninteractive private comparison scheme is XCMP proposed by Lu et al. [21], which is built from fully homomorphic encryption (FHE) and offers output expressive and parametrizable features. This scheme allows comparison between two encrypted non-negative integers, and the result can be obtained after decryption. They also embedded it in the framework proposed by Tai et al. [25] to conduct PPDT evaluation. However, the scale of the input domain and the reuse of the outputs is limited, for it does not support multiplying two comparison results. Since then, studies such as [17] and [26] have tried to exploit more practical designs to obtain fully output expressive schemes, which is, however, at the cost of reducing the performance or increasing the communication bandwidth. Although several efforts have been made, these HE-based schemes are difficult to apply due to the high computational overhead of HE and limited input domain, i.e., only integers.

The improvement in computing capacity of devices has alleviated this problem, especially on devices equipped with graphic processing units (GPUs), which can fully exploit the parallelism of GPUs to accelerate the computing process. Previous works demonstrated that using GPU to accelerate computation can improve efficiency greatly by orders of magnitude, which can be divided into two categories, one focusing on accelerating functions of HE such as homomorphic multiplication and bootstrapping, and the other focusing on accelerating applications constructed on HE such as PrivFT. Nevertheless, the memory overhead of the scheme is still high, and optimizing the kernel for a specific function requires many factors to be taken into account, such as trade-offs between efficiency and resource, conflict caused by stride data access, allocation of pipelines, etc.

Table 1: Comparison with existing private comparison protocols.

Methods	Input	Input domain	Plaintext space	Multiplicative depth	Expressiveness
XCMP [21]	$[\![A]\!], [\![B]\!]$	$A, B \in [0, N-1]$ $A, B \in [N, N^2-1]$	$\mathcal{R}_q$	1 2	ADD, MUL
[17]	$[\![A]\!], [\![B]\!]$	$A, B \in [0, N-1]$ $A, B \geq N$	$\mathcal{R}_q$	1 $\left\lceil \log \left\lceil \frac{\mu}{\log N} \right\rceil \right\rceil + 1$	ADD, MUL
cuXCMP	$[\![A_{\mathcal{I}}]\!], [\![A_{\mathcal{D}}]\!], [\![B_{\mathcal{I}}]\!], [\![B_{\mathcal{D}}]\!]$	$A, B \in \mathbb{Q}$	$\mathcal{R}_q$	$\left\lceil \log \left\lceil \frac{\mu}{\log N} \right\rceil \right\rceil + 1$	ADD, MUL

Contributions. In this paper, we propose a more generic noninteractive private comparison scheme cuXCMP based on HE as well as an optimized GPU implementation to accelerate the execution. Our cuXCMP has good expansibility and performance, and offers easy integration of existing PDTE protocols. A comparison of similar schemes is shown in Table 1. In details, the main contributions of this paper can be summarized as follows:

– We change the element encoding method and exploit a constant term extraction approach, resulting in a more extensible and practical scheme that is fully output expressive and allows comparison over negative and float numbers.
– We improve the performance of the scheme by optimizing the implementation using CUDA-enabled GPUs. We focus on the automorphism and key-switching kernel (AKS) and constant term extraction kernel (CTX), which are in nature memory-bound, and propose several memory-centric optimizations. In detail, our optimizations include the follows:
    - We present compact GPU implementations of NTT and INTT, which fully utilize the shared memory and can process 8192-point NTT within a single block;
    - For the AKS kernel, we apply kernel fusing to reuse the data stored in the shared memory and registers, and reduce the grid dimension decrease the resource usage. Meanwhile, we eliminate some synchronization instructions during execution to reducing the runtime.

  - For the CTX kernel, we improve the parallelism by allocating multiple CUDA streams that allow kernels to run concurrently and hide the IO latency. We fine tune the number of streams through precisely latency measurement;

- Our optimizations boost the overall execution time. The AKS kernel, CTX kernel and scheme outperform the C (resp., AVX512) baselines by 23.71×, 15.58×, and 1.83× (resp., 18.29×, 11.75×, and 1.42×) for one number comparison, respectively. Additionally, we obtain the best speedup ratio by setting the scale of comparisons to be multiples of the stream numbers, e.g., we speed up the scheme by 1.99× when comparing 32 numbers.

The rest of the paper is organized as follows: Sec 2 covers preliminaries, including introductions to the HE scheme, private comparison, and GPU implementation basics. We present our scheme design in Sec 3 and implementation optimizations in Sec 4. Sec 5 provides the performance results of our implementation on NVIDIA Tesla V100S PCIe.

2 Preliminaries

2.1 Notation

We define the group of rational numbers by $\mathbb{Q}_N = \{i \in \mathbb{Q}, -\frac{N}{2} \leq i < \frac{N}{2}\}$ and $\mathbb{Q}_N^+ = \{i \in \mathbb{Q}, 0 \leq i < N\}$. For a positive integer q, denote the finite filed of all congruence classes modulo q by $\mathbb{A}_q = [0, q) \cap \mathbb{Z}$ and $\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z} = [-\frac{q}{2}, \frac{q}{2}) \cap \mathbb{Z}$. Let $\mathcal{R} = \mathbb{Z}[X]/(\Phi_{2N}(X))$ be the ring of polynomials of which the coefficients come from $\mathbb{Z}$, and $\mathcal{R}_q = \mathcal{R}/q\mathcal{R}$ be the residue ring of $\mathcal{R}$ modulo q, where N is a power of 2 and $\Phi_{2N}(X) = X^N + 1$ is the $2N$-th cyclotomic polynomial of degree N. A polynomial $f \in \mathcal{R}_q$ is represented in the form $f = \sum_{i=0}^{N-1} f_i X^i$.

We use $[\![\cdot]\!]$ to denote an encrypted element. The notation $2^k \mid\mid i$ refers to $2^k \mid i$ and $2^{k+1} \nmid i$. Denote $\mathbf{1}\{\mathcal{P}\}$ a Boolean expression that returns 1 when $\mathcal{P}$ is true and 0 otherwise. The homomorphism addition, subtraction and multiplication are written as $\oplus$, $\ominus$, and $\otimes$ respectively. The symbol $\leftarrow$ represents variable assignment or uniformly random sampling from a distribution.

2.2 RLWE-based homomorphic encryption

Homomorphic encryption allows secure arithmetic computations over encrypted data without the knowledge of a secret key. An HE scheme consists of the following algorithms:

- Setup. Takes the security parameter λ as input and outputs the public parameters $\mathsf{Prm} \leftarrow \mathrm{HE.Setup}(1^\lambda)$.
- Key generation. This process takes the parameters Prm as input and generates two types of keys. The first is the public and secret key pair $(pk, sk) \leftarrow \mathrm{HE.KeyGen}_{enc}(\mathsf{Prm})$ used in encryption and decryption, and the other is the evaluation key $ek \leftarrow \mathrm{HE.KeyGen}_{eva}(\mathsf{Prm})$.
- Encryption. Given a public key pk and a message m, output $c \leftarrow \mathrm{HE.Enc}(pk, m)$, which is the encryption of m using pk.
- Decryption. Decrypt the ciphertext c using the secret key sk and output the result $m \leftarrow \mathrm{HE.Dec}(sk, c)$.
- Evaluation. Take as inputs an evaluation key ek, a function f, and two ciphertext c_1, c_2, output the encrypted evaluation result of f over c_1 and c_2. The following properties hold during evaluation:

$$\mathrm{HE.Dec}(sk, [\![c_1]\!] \oplus [\![c_2]\!]) = \mathrm{HE.Dec}(sk, [\![c_1 + c_2]\!]), \mathrm{HE.Dec}(sk, [\![c_1]\!] \oplus c_2) = \mathrm{HE.Dec}(sk, [\![c_1 + c_2]\!])$$
$$\mathrm{HE.Dec}(sk, [\![c_1]\!] \otimes [\![c_2]\!]) = \mathrm{HE.Dec}(sk, [\![c_1 \times c_2]\!]), \mathrm{HE.Dec}(sk, [\![c_1]\!] \otimes c_2) = \mathrm{HE.Dec}(sk, [\![c_1 \times c_2]\!])$$

In this work, we focus on HE schemes based on the ring-learning with errors (R-LWE) problem [22] due to its efficient algebraic structure. Many efficient HE schemes are constructed on the R-LWE problem, such as BGV [6] and CKKS [9], where the computation is performed on polynomials.

Polynomial multiplication and reduction. Polynomial multiplication and reduction are two performance bottlenecks of HE schemes. The Number Theoretic Transform (NTT) is often used to accelerate multiplication, for it reduces the computation complexity from $O(N^2)$ to $O(N \log N)$. Denote ω and γ as the N-th and $2N$-th root of unity respectively. The forward NTT of a polynomial $f = \sum_{i=0}^{N-1} f_i X^i$ can be represented as $\mathrm{NTT}(f) = \hat{f} = \sum_{i=0}^{N-1} \hat{f}_i X^i$ where $\hat{f}_i = \sum_{j=0}^{n-1} \gamma_n^j \cdot \omega_n^{ij} \cdot f_j$, and for the inverse NTT is $f = \mathrm{INTT}(\hat{f}) = \sum_{i=0}^{N-1} f_i X^i$, where $f_i = \frac{1}{n} \cdot \gamma_n^{-i} \sum_{j=0}^{n-1} \omega_n^{-ij} \cdot \hat{f}_j$. The forward NTT transforms the polynomials into the NTT domain, and then two polynomials are multiplied in a point-wise manner. Since this transformation is an isomorphism, we have $f \cdot g = \mathrm{NTT}^{-1}(\mathrm{NTT}(f) * \mathrm{NTT}(g))$, here $*$ denote to the point-wise multiplication. Barrett reduction is commonly applied to reduce coefficients. This method replaces the time-consuming division operation with a multiplication and a bit shift operation, which can be implemented efficiently due to the hardware structure of a processor.

RNS representation. Let $q = \Pi_{i=0}^{r-1} q_i$ donate the decomposition of the modulus q to a set of co-prime moduli. The Chinese Remainder Theorem (CRT) define a map from $\mathbb{Z}_q$ to $\Pi_{i=0}^{r-1} \mathbb{Z}_{q_i}$ so that $a \bmod q \mapsto (a \bmod q_0, a \bmod q_1, \cdots, a \bmod q_{r-1})$. This can be extended to a ring isomorphism $\mathcal{R}_q \cong \mathcal{R}_{q_0} \times \mathcal{R}_{q_1} \times \cdots \times \mathcal{R}_{q_{r-1}}$. Through this way, a polynomial with large coefficients can be decomposed into r polynomials with smaller coefficients, which is called the residue number system (RNS) representation. The CRT/RNS technique can be used to reduce computational overhead, for the scale of the coefficients decreased.

Modulus switching. Modulus switching is used for converting a ciphertext c under the modulus q into another ciphertext c' under the modulus q' correspond to the same secret key s. This method is used by scaling c with factor $\frac{q'}{q}$ and then rounding it to an integer, and is often used in controlling noise.

Key switching. For RLWE ciphertexts, the key switching procedure, given a ciphertext $\mathbf{ct}_0 = (c_0, c_1) \in \mathcal{R}^2$ and secret keys $s_0, s_1 \in \mathcal{R}$, outputs a ciphertext $\mathbf{ct}_1 \in \mathcal{R}^2$ with subgaussian error, aiming at converting a ciphertext into another under a different key while approximately preserving its phase [8, 14]. It consists of two procedures: first, compute the switching key $\mathbf{K} = [\mathbf{k}_0 \mid \mathbf{k}_1] \in \mathcal{R}_q^{d \times 2}$, where $\mathbf{k}_0 = -s_1 \cdot \mathbf{k}_1 + s_0 \cdot \mathbf{g} + \mathbf{e} (\bmod q)$, $\mathbf{k}_1 \leftarrow U(\mathcal{R}_q^d)$, and $\mathbf{e} \leftarrow \chi^d$; second, compute and return the ciphertext $\mathbf{ct}_1 = (c_0, 0) + \mathbf{g}^{-1}(c_1) \cdot \mathbf{K} \pmod q$, where the map $\mathbf{g}^{-1} : \mathcal{R}_q \to \mathcal{R}^d$ is the gadget decomposition [8]. The output satisfies that $\langle \mathbf{ct}_1, (1, s_1) \rangle = \langle \mathbf{ct}_0, (1, s_0) \rangle + \left\langle \mathbf{g}^{-1}(c_1), \mathbf{e} \right\rangle$.

Galois automorphism. For $i \in \mathbb{Z}_{2N}^*$, the Galois automorphism over $\mathcal{R}$ is defined as a map $\varrho_i : \mathcal{R} \to \mathcal{R}$, $f(X) \mapsto f(X^i) \bmod \Phi_{2N}(X)$. Specifically, there are N automorphisms over $\mathcal{R}$ in total that form N different permutations on plaintext slots, yielding new ciphertexts corresponding to other secret keys. As this operation changes the secret key, we need to switch it back through the key switching operation each time after applying an automorphism.

2.3 Private Comparison and XCMP

We recall the homomorphic encryption based private comparison scheme XCMP proposed by Lu et al. [21], which allows to evaluate the comparison $a > b$ noninteractively and homomorphically. The inputs are two encrypted $2 \log N$-bit integers, and the output holds the expressive feature that the arithmetic operations can act directly on it without extra interactions with other entities.

The basic idea of XCMP is to encode the two integers into two monomials through the map $\pi : \mathbb{Z} \to \mathcal{R}_q$, $\pi(a) = X^a$, and then construct a polynomial containing a and b of which the constant term reveals the comparison result. In details, the proposed polynomial is $C = T \times \pi(a) \times \pi(b) \bmod (X^N + 1)$, where $T = 1 + X + \cdots + X^{N-1}$. When $a \le b$, $C[0]$ comes from the $(b-a)$-th coefficient of T, so that we have $C[0] = X^{b-a} \cdot X^{a-b} = X^0 = 1 \bmod (X^N + 1)$. Otherwise, $C[0]$ comes from the $(N + b - a)$-th coefficient of T and $C[0] = X^{N+b-a} \cdot X^{a-b} = X^N = -1 \bmod (X^N + 1)$. Meanwhile, XCMP offers a way to parametrize the output. Here, the coefficients of T is set as μ and a noise polynomial $R \leftarrow \mathcal{R}_q$ with $R[0]$ set to be $\bar{\mu}$ should be added to C, where $\bar{\mu} = (\mu_0 + \mu_1) \cdot 2^{-1}$ and $\mu = \mu_0 - \bar{\mu}$. So that $C[0] = \mu_0$ if $a \le b$ and otherwise $C[0] = \mu_1$.

The output of XCMP is an encryption of the polynomial $C = \mathbf{1}\{a > b\} + \sum_{i=1}^{N} r_i X^i$, which preserves additive and multiplicative homomorphism properties. Specifically, denote $[\![C_1]\!] = \mathrm{XCMP}([\![X^a]\!], [\![X^{b_1}]\!])$ and $[\![C_2]\!] = \mathrm{XCMP}([\![X^a]\!], [\![X^{b_2}]\!])$, summing $[\![C_1]\!]$ and $[\![C_2]\!]$ homomorphically gives the encryption of $\mathbf{1}\{a > b_1\} + \mathbf{1}\{a > b_2\}$, and $[\![e \cdot C_1]\!]$ contains the result of $e \cdot \mathbf{1}\{a > b\}$, where e is a number.

2.4 CUDA Programming Model

Compute Unified Device Architecture (CUDA) [23] is a parallel computing platform and programming model that allows software to access and accelerate processing with graphics processing units (GPUs), which provides more powerful compute capabilities. The programming method follows the host-to-device model. In this setting, a kernel is called by the by the host (CPU) and executed on the device (GPU), and then a set of CUDA threads is launched and executes the same functions defined by the kernel in parallel. The structure of GPU can be abstract as three hierarchies: grid, block and thread. A grid consists of several blocks, and each block can launch up to 1024 threads. All these can be formed into 1- to 3-dimension arrays to be compatible with the operands' structure. During execution, the threads are bundled per 32 in a warp, and then the streaming multiprocessor (SM) schedules the warps and executes the operations concurrently.

Memory in GPU includes global memory, constant memory, shared memory and registers. The global memory is accessible by all threads, which has the largest storage capacity and access latency. The threads within a block can access the shared memory, and the communication speed is much more faster. The registers are private and are the local memory for a thread.

Cooperative groups. Synchronization among threads ensures the safety and maintainability of programs. The cooperative group is an extension of the CUDA programming model that allows the kernel to organize groups of threads dynamically and synchronize them within and across blocks. For example, synchronization across the grid within a kernel can be realized by invoking the function `grid.sync()`.

Kernel fusing. There are two alternative methods to utilize kernels: single-kernel approach and multi-kernel approach, denoting the number of kernels invoked to accomplish a task. Both approaches have their relative strengths and weaknesses in relation to different application scenarios, and a technique named kernel fusing can be introduced to fuses multiple kernels into one. Combining multiple kernels allows threads to share the data in the shared memory and reuse the data in the registers, so that the time consumed in data transmission is reduced.

Cuda stream. Using multiple streams and letting independent operations run concurrently lead to a more flexible and fast execution. CUDA streams and events are such techniques though which we can prescribe execution order of operations within and across streams. This technique is of great benefits as it overlaps the data communication and kernel execution. Specifically, we can launch multiple streams to pipeline the data transmission between different memory field and the computation of operations, which are not necessarily sequential. Through this, we can reduce the latency and improve the performance.

3 Private Decision Tree Evaluation

The cuXCMP scheme inherits the noninteractive feature of XCMP, while offers rational number comparison and is fully output expressive. In this section, we present a modular description of cuXCMP, as well as a method to integrate it into a PDTE protocol.

```
$\mathrm{CTX}([\![C]\!], \mathbf{swk})$ :
  1: for $i = 0$ to $\log N$ do
  2:     $[\![C]\!] \leftarrow [\![C]\!] \oplus \mathrm{AKS}([\![C]\!], \mathbf{swk}[i])$
  3: return $[\![c]\!] \leftarrow [\![C]\!]$

$\mathrm{cuXCMP}_l^{neq}([\![X^{a_\mathcal{D}}]\!], [\![X^{b_\mathcal{D}}]\!])$ :
  1: $[\![X^{-b_\mathcal{D}}]\!] \leftarrow [\![X^{b_\mathcal{D}}]\!]$
  2: $[\![E]\!] \leftarrow 1 \ominus [\![X^{a_\mathcal{I}}]\!] \otimes [\![X^{-b_\mathcal{I}}]\!]$
  3: return $[\![e]\!] \leftarrow \mathrm{CTX}([\![E]\!])$

$\mathrm{cuXCMP}_l^{cmp}([\![X^{a_\mathcal{I}}]\!], [\![X^{b_\mathcal{I}}]\!])$ :
  1: $\bar{\mu} \leftarrow 2^{-1} \bmod q$, $\mu \leftarrow -\bar{\mu} \bmod q$
  2: $T \leftarrow \sum_{i=0}^{N-1} \mu X^i$
  3: $R \leftarrow \mathcal{R}_q$, $R[0] \leftarrow \mu$
  4: $[\![X^{-b_\mathcal{I}}]\!] \leftarrow [\![X^{b_\mathcal{I}}]\!]$
  5: $[\![C]\!] \leftarrow [\![X^{a_\mathcal{I}}]\!] \otimes [\![X^{-b_\mathcal{I}}]\!] \otimes T \oplus R$
  6: return $[\![c]\!] \leftarrow \mathrm{CTX}([\![C]\!])$
```

```
Client :
$([\![A_\mathcal{I}]\!], [\![A_\mathcal{D}]\!]) \leftarrow \mathsf{Enc}(a, pk)$, $a \in \mathbb{Q}_N$
1: $a' \leftarrow \sigma(a)$, $a_\mathcal{I} \leftarrow \lfloor a' \rfloor$, $a_\mathcal{D} \leftarrow \delta(a' - a_\mathcal{I})$
2: $A_\mathcal{I} \leftarrow \pi(a_\mathcal{I})$, $A_\mathcal{D} \leftarrow \pi(a_\mathcal{D})$
3: $[\![A_\mathcal{I}]\!] \leftarrow \mathrm{HE.Enc}(A_\mathcal{I}, pk)$,
   $[\![A_\mathcal{D}]\!] \leftarrow \mathrm{HE.Enc}(A_\mathcal{D}, pk)$
Server :
$[\![c]\!] \leftarrow \mathrm{cuXCMP}([\![A_\mathcal{I}]\!], [\![A_\mathcal{D}]\!], [\![B_\mathcal{I}]\!], [\![B_\mathcal{D}]\!])$
1: $[\![c_\mathcal{I}]\!] \leftarrow \mathrm{cuXCMP}^{cmp}([\![A_\mathcal{I}]\!], [\![B_\mathcal{I}]\!])$,
   $[\![c_\mathcal{D}]\!] \leftarrow \mathrm{cuXCMP}^{cmp}([\![A_\mathcal{D}]\!], [\![B_\mathcal{D}]\!])$
2: $[\![e_\mathcal{I}]\!] \leftarrow \mathrm{cuXCMP}^{neq}([\![A_\mathcal{I}]\!], [\![B_\mathcal{I}]\!])$
3: $[\![c]\!] \leftarrow [\![e_\mathcal{I}]\!] \otimes ([\![c_\mathcal{I}]\!] \ominus [\![c_\mathcal{D}]\!]) \oplus [\![c_\mathcal{D}]\!]$
Client :
$c \leftarrow \mathrm{HE.Dec}([\![c]\!], sk)$
```

Fig. 1: The specifications of the proposed cuXCMP scheme (right), and the private comparison and inequality test algorithm (left).

3.1 Our proposal: cuXCMP

The private comparison scheme XCMP proposed in [21] (see Sec 2.3) allows comparison between two ℓ-bit integers $a, b \in \mathbb{A}_N$, $\ell = \log N$. Although it offers a way to extend the domain to $\mathbb{A}_{N^2}$ to compare two 2ℓ-bit integers, the use in practice is limited for it only supports comparison between nonnegative integers. In this section, we present a scheme cuXCMP, aiming at evaluating comparison between rational numbers privately to meet practical requirements. The proposed cuXCMP consists of two algorithms: $\mathrm{cuXCMP}_\ell^{cmp}$ and $\mathrm{cuXCMP}_\ell^{neq}$, one for comparing two $(\ell + 1)$-bit signed elements (one bit for the sign) in $\mathbb{Q}_N$ and the other is to test whether the two numbers are unequal.

cuXCMP offers similar noninteractive and output expressive features as XCMP [21] as it derived from it, while allowing a larger input domain and multiplication of ciphertexts. The scheme is shown in Fig. 1. We consider the client-server setting, which is a simplification of the three kinds of stakeholders in XCMP by letting the encryptor and decryptor be identical. Specifically, the client encodes and encrypts the data through the function Enc, and then sends it to the server. The evaluation is completely delegated to the server, for the inputs and output are all ciphertexts. The server executes the private comparison as cuXCMP defines and sends the result to the client for decryption. Different to XCMP that outputs an encrypted polynomial, the output of cuXCMP is a Boolean of the expression $\mathbf{1}\{a > b\}$.

Elements encoding. The compared elements need to be encoded to the plaintext space of the encryption scheme. Define map $\sigma : \mathbb{Q}_N \to \mathbb{Q}_N^+, a \mapsto a + \frac{N}{2}$, which maps elements in $\mathbb{Q}_N$ to $\mathbb{Q}_N^+$ by adding $\frac{N}{2}$ to obtain nonnegative numbers. For $a \in \mathbb{Q}_N$, the encoding procedure first transforms the domain from $\mathbb{Q}_N$ to $\mathbb{Q}_N^+$ through mapping a to $a' = \sigma(a)$, then split a' and get the integral part $a_\mathcal{I}$ and the rescaled decimal part $a_\mathcal{D}$. One thing to note is that we rescale the decimal part by multiplying it with δ and get $a_\mathcal{D}$, which is an integer, and the system accuracy determines δ. Afterwards, we follow the XCMP encoding procedure and map $a_\mathcal{I}$ and $a_\mathcal{D}$ to monomials through $\pi(\cdot)$ and get $A_\mathcal{I} = X^{a_\mathcal{I}}$ and $A_\mathcal{D} = X^{a_\mathcal{D}}$.

Constant term extraction. XCMP [21] takes two integers as input and outputs the ciphertext of a polynomial that contains the comparison result in its constant term, and the other terms are random and have no actual meaning. This feature makes the output not fully expressive, for multiplying two ciphertexts only yields an encryption of a random polynomial. A constant term extraction technique over the ciphertext can remove the influence caused by random coefficients and make the output fully expressive. Evaluating the field trace through summing the automorphisms τ_γ can homomorphically extract the constant term [8,17,18]. Define sets $I_j = \{i \in [N] : 2^j \mid\mid i\}$ for $j \in [0, logN)$ and $I_{\log N} = 0$. Then, the disjoint union of I_j forms the polynomial index set, i.e., $\bigcup_{0\le j\le \log N} I_j = [0, N)$. For an integer $\gamma = 2^\ell + 1$, $1 \le \ell \le \log N$, the automorphism τ_γ, which maps i to $i \cdot \gamma$, is a signed permutation on I_j [8]. Specifically, if $2^{\log N - \ell + 1} \mid i$, $\tau_\gamma(X^i) = X^i$, while $\tau_\gamma(X^i) = -X^i$ for $2^{\log N - \ell} \mid\mid i$, and we can obtain this through the Galois automorphism. Namely, adding the automorphism to polynomial doubles the coefficients of the even terms while zeroizes the odd terms. Summing up the automorphisms recursively yields a way to isolate the polynomial coefficients [4,18], which equals to the field trace function $\mathrm{Tr}_{\mathcal{R}/\mathbb{Z}} : \mathcal{R} \to \mathbb{Z}$. This function can be evaluated homomorphically, with one time-consuming key switching procedure needed after each automorphism to get the encryption with respect to the original key. The $\mathrm{Tr}_{\mathcal{R}/\mathbb{Z}}$ is often decomposed into individual trace functions $\mathrm{Tr}_{\mathcal{R}^{(i)}/\mathcal{R}^{(i-1)}} = \mathcal{R}^{(i)} \to \mathcal{R}^{(i-1)}$, where $\mathbb{Z} = \mathcal{R}^{(0)} \subseteq \mathcal{R}^{(1)} \subseteq \cdots \subseteq \mathcal{R}^{(r)} = \mathcal{R}$ [4,8]. Thus, we can evaluate it in quasilinear time, and the complexity is reduced.

Inequality test. Denote cuXCMP_l^{neq} as the inequality test procedure of two ℓ-bit integers in the cuXCMP scheme. As is specified in Fig. 1, cuXCMP_l^{neq} follows the same method as XCMP [21] in inequality test. Specifically, we can obtain the test result through computing $[\![E]\!] = 1 \ominus [\![X^a]\!] \otimes [\![X^{-b}]\!]$. The constant term of polynomial E maintains 1 when $a > b$ and downgrades to 0 otherwise. Then, we can extract the result through the constant term extraction method.

Domain extension. We apply the domain extension method proposed in [17], of which the main idea is splitting the numbers into several chunks and comparing them respectively. Denote m and n as the number of chunks, the outputs of inequality test and comparison can be expressed as the following:

- $[\![e]\!] \leftarrow [\![e_0]\!] \otimes [\![e_1]\!] \otimes \cdots \otimes [\![e_{m-1}]\!]$
- $[\![c]\!] \leftarrow ([\![e_{n-1}]\!] \otimes [\![c_{n-1}]\!]) \oplus \left([\![c_0]\!] \otimes_{i=1}^{n-1} [\![\overline{e_i}]\!]\right) \oplus_{i=1}^{n-2} \left([\![e_i]\!] \otimes [\![c_i]\!] \otimes_{j=i+1}^{n-1} [\![\overline{e_j}]\!]\right)$

Private comparison. The input to cuXCMP is four encrypted polynomials $[\![A_\mathcal{I}]\!]$, $[\![A_\mathcal{D}]\!]$, $[\![B_\mathcal{I}]\!]$, and $[\![B_\mathcal{D}]\!]$, which refer to the ciphertexts of integral and decimal parts of a and b, $a, b \in \mathbb{Q}_N^+$. cuXCMP functions by calling the subprocedure cuXCMP_l^{cmp} several times and manipulating the output results. Here, the cuXCMP_l^{cmp} takes two ℓ-bit nonnegative integers as input and homomorphically evaluates the comparison as XCMP [21] does, while differs in the output form, which is the extracted encrypted constant term. During the evaluation, we first compare the integral and decimal parts respectively and get $[\![c_\mathcal{I}]\!] \leftarrow \mathrm{cuXCMP}_{l_1}^{cmp}([\![A_\mathcal{I}]\!], [\![B_\mathcal{I}]\!])$ and $[\![c_\mathcal{D}]\!] \leftarrow \mathrm{cuXCMP}_{l_2}^{cmp}([\![A_\mathcal{D}]\!], [\![B_\mathcal{D}]\!])$. Then, by testing the inequality of the integral part, we get $[\![e_\mathcal{I}]\!] = \mathrm{cuXCMP}_{l_1}^{neq}([\![A_\mathcal{I}]\!], [\![B_\mathcal{I}]\!])$. The comparison of a and b can be expressed as the following:

$$\begin{aligned}
&\mathbf{1}\{a > b\} = \mathbf{1}\{a' > b'\} \\
&= \mathbf{1}\{a_\mathcal{I} \neq b_\mathcal{I}\} \cdot \mathbf{1}\{a_\mathcal{I} > b_\mathcal{I}\} + \mathbf{1}\{a_\mathcal{I} = b_\mathcal{I}\} \cdot \mathbf{1}\{a_\mathcal{D} > b_\mathcal{D}\} \\
&= \mathbf{1}\{a_\mathcal{I} \neq b_\mathcal{I}\} \cdot (\mathbf{1}\{a_\mathcal{I} > b_\mathcal{I}\} - \mathbf{1}\{a_\mathcal{D} > b_\mathcal{D}\}) + \mathbf{1}\{a_\mathcal{D} > b_\mathcal{D}\}
\end{aligned}$$

So the result can be obtained by computing $[\![c]\!] \leftarrow [\![e_\mathcal{I}]\!] \otimes ([\![c_\mathcal{I}]\!] \ominus [\![c_\mathcal{D}]\!]) \oplus [\![c_\mathcal{D}]\!]$.

3.2 Noninteractive privacy-preserving decision tree

In the noninteractive HE-based private decision tree evaluation setting, the server preserves an encrypted decision tree. A feature vector is encrypted by the client and sent to the server for evaluation. Then, the server conducts the comparison of every tree nodes and evaluates the path homomorphically. The computation is completely dedicated to the server with no information leakage, for the evaluation is processed over encrypted data.

<u>Client</u>		<u>Server</u>
Initiation		**Deserialization**
$\mathsf{Prm} \leftarrow \mathrm{HE.Setup}(1^\lambda)$	$\xrightarrow{\mathsf{Prm}}$	
$(pk, sk) \leftarrow \mathrm{HE.KeyGen}_{\mathrm{enc}}(\mathsf{Prm})$		Get Prm
$ek \leftarrow \mathrm{HE.KeyGen}_{\mathrm{eva}}(\mathsf{Prm})$	$\xrightarrow{pk, ek}$	
		Get pk, ek
Encoding and encryption		
$\mathbf{a} = \{a_0, \cdots, a_{n-1}\} \in \mathbb{Q}^n$		
For $i \in \{0, \cdots, n-1\}$		
$\quad C_{A_i} \leftarrow \mathsf{Enc}(a_i)$	$\xrightarrow{n, \{C_{A_i}\}}$	**Evaluation**
		For $i \in \{0, \cdots, n-1\}$
		$\quad [\![c_i]\!] \leftarrow \mathrm{cuXCMP}(C_{A_i}, C_{B_i})$
Decryption	$\xleftarrow{[\![v]\!]}$	$[\![v]\!] \leftarrow \mathcal{T}(\{[\![c_i]\!]\})$
$v \leftarrow \mathrm{HE.Dec}([\![v]\!], sk)$		

Fig. 2: The privacy-preserving decision tree evaluation protocol embedded with cuXCMP.

We follow the definitions in [25] with a modification to the data structure and denote the decision tree evaluation function as $\mathcal{T} : \mathbb{Q}_n \mapsto \mathbb{Z}^m$, which maps a rational attribute vector $\mathbf{a} = (a_0, \cdots, a_{n-1}) \in \mathbb{Q}^n$ to a classification labels set $\{v_0, \cdots, v_{m-1}\}$. This evaluation function can be instantiated according to the application, e.g., $\mathcal{T} := \sum_i^{n-1} w_i \cdot c_i$, where w_i denotes to the weight. Let (pk, sk) be the keypair used in data encryption, where the client holds the sk. The privacy-preserving decision tree evaluation protocol is shown in Fig. 2 and illustrated as follows.

1. **Client:** Sets the system parameters according to the security level , and then generate the public and private keys (pk, sk) for encryption and the ek for evaluation. Afterwards, the client sends the public parameters, pk and ek to the server.
2. **Server:** Deserializes the received packets to get the public parameters and keys.
3. **Client:** Encodes and encrypts each element of the feature vector $\mathbf{a} \in \mathbb{Q}^n$ through the function Enc and gets a ciphertexts set $\{C_{A_i} := ([\![A_{\mathcal{I},i}]\!], [\![A_{\mathcal{D},i}]\!])\}$, $i \in [0, n)$, and then sends it to the server.
4. **Server:** A decision tree is dedicated to the server, of which each decision node b_i has an encrypted label $C_{l_i} := [\![l_i]\!]$ and threshold $C_{B_i} := ([\![B_{\mathcal{I},i}]\!], [\![B_{\mathcal{D},i}]\!])$, $i \in [0, n)$. After receiving the set, the server applies cuXCMP on each element and gets the comparison results of every decision node. The classification result is obtained through the function EvalPath, which is defined as $v := \sum_i^{n-1} l_i \cdot c_i$ and can be evaluated homomorphically with encrypted inputs. The output that sent to the client is a ciphertext of the classification result.
5. **Client:** Decrypt the ciphertext using sk and get the classification result.

Security analysis. Our protocol is secure under the semi-honest assumption, as it preserves the privacy of the feature vector and the decision tree. In details, all the feature vectors sent to the server are encrypted. Thus, the server can learn nothing about the components of the vectors. The decision tree is preserved in the form of ciphertexts, and the evaluation process is conducted through homomorphic operation, thus leaking no information about the threshold and label of every decision nodes. The only thing they can get through the evaluation is the number of attributes, which is the system parameter and has no effect on the entire security. Thus, there is no additional information that can help to take apart the real world execution from the deal world execution, of which the protocol players can access a trusted third party that evaluates the functionality [21].

4 GPU Implementation and Optimizations

In this section, we introduce our compact and efficient GPU implementation of cuXCMP. We focus on optimizing the constant term extraction function based on a deep analysis of the performance bottlenecks, i.e., the CTX kernel, which contains the NTT, INTT, and AKS. The memory-bound nature of HE makes us strive for better memory management. We propose a compact design of NTT and INTT which fully utilized the shared memory. Based on this, we present several memory-centric optimizations, such as kernel fusing and grid dimension reduction. We also allocate multiple CUDA streams to hide the IO latency. Through these approaches, we reduce the wait time of data interaction and resource usage, thus improve the overall performance.

4.1 Baseline implementation and bottleneck analysis

To meet the requirements of practical applications and security, we instantiate our cuXCMP with the polynomial degree $N = 8192$, and the detailed parameters are shown in Table 2. In this configuration, we provide comparison of two integers of which the sizes do not exceed 13 bits. For cuXCMP, this requirement is equivalent to having the integer part of the input number (except for the sign bit) not exceed 12 bits, and the precision not exceed 13 bits. According to the LWE estimator [3], this parameter set offers 128-bit security.

Because XCMP instantiated on SEAL shows better performance [21], we follow their method and implement the C baseline of cuXCMP based on this library. To provide the property of fully output expressive, our scheme requires additional constant term extraction compared to XCMP, which consists of $\log N$ automorphism and key switching operations. This cannot be eliminated, but the total execution time can be reduced by optimizing the implementation or by parallel acceleration, etc.

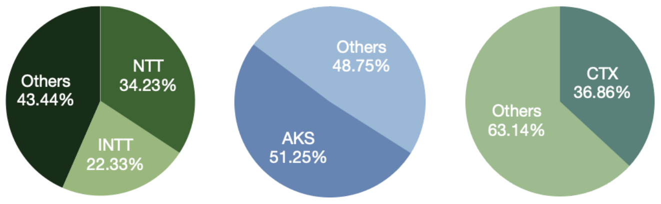

Fig. 3: Execution time analysis of baseline C implementation.

Table 2: Parameter specifications of cuXCMP

Params.	Polynomial modulus degree	Modulus		Security level
		Plaintext	Ciphertext	
Value	8192	20 bits	174 bits	128

To obtain the major performance bottlenecks, we conducted an execution time analysis of our baseline C implementation of cuXCMP. We measured the total runtime of the functions contained in this scheme, and the results are shown in Fig. 3. For the arithmetic operations, NTT accounts for 34.23% of the total runtime of the entire program, while for INTT is 22.33%. Meanwhile, the upper-layer functions, such as AKS and CTX, account for 51.25% and 36.86% respectively.

The CTX function plays an important role in cuXCMP since the claimed output expressive property can only be achieved by extracting the constant term of the encrypted polynomials. Considering that the CTX function occupies more than one-third of the running time of the program, we concentrate on optimizing these functions throughout this paper. It is worth noting that this function has a highly parallel processing structure, which makes our optimizations practical. Firstly, the element-wise arithmetic operations can be executed separately; secondly, the NTT and INTT of different residual families under the RNS representation are mutually independent; thirdly, multiple AKS functions in the CTX can be executed simultaneously.

The HE scheme is inherently memory-bound because it has many data interactions while the arithmetic operations are simple and non-intensive. Moreover, the latency of memory access can be tens of times higher than that of arithmetic operations. This fact makes it a priority to reduce data accesses when optimizing the GPU implementation of our HE-based scheme.

4.2 Compact design of NTT

According to the instantiation and security requirement of cuXCMP, we implement both forward and inverse NTT supporting $N = 8192$, represented as NTT and INTT in the following respectively, for short.

Tradeoffs and optimizations. The process of NTT and INTT contain simple arithmetic operations like addition, subtraction and multiplication, but involve quite time-consuming data access for polynomial coefficients and twiddle factors. This low arithmetic intensity nature indicates that the implementation of NTT and INTT on the GPUs should be memory-bound, of which the most important performance bottleneck is the IO latency. Thus, the straightforward method that relaunching a kernel after each NTT level will introduce $O(N \log N)$ fetching and storing data from global memory and cause a significant drop in performance. To reach peak bandwidth, we should minimize the interaction with global memory and store data in shared memory as much as possible, which has a faster data access speed.

For this consideration, we utilize one block fully to perform a 4 per-thread 8192-point NTT, i.e., each thread handles 4 butterfly operations that involve 8 elements during the processing. The process and memory access pattern of our technique is shown in Fig. 4. Different from prior works such as [19] that employed two kernels and more blocks, our compact design reduces the complexity of interaction with global memory to $O(\log N)$. To make full use of the block resource, we store the polynomial in shared memory and the roots in registers. This approach requires 64KB in shared memory allocation to store N 64-bit elements.

One thing to notice is that high block launch overhead will lead to low SM occupancy. In details, all SMs cannot execute blocks with too much memory consumption separately at the same time because the GPU onboard memory they share is fixed. This leads to a part of the SMs being idle, as well as low GPU occupancy, which is a fallacy that low occupancy leads to low efficiency. The truth is that for memory-bound GPU implementation, letting each thread perform more operations provides a better way to hide latency. Meanwhile, by storing all the elements in shared memory, we can omit the time-consuming data synchronization operations among different blocks and global memory.

PTX butterfly. For the butterfly operation we apply the method proposed by Harvey [16] in a reverse manner. Specifically, we perform Cooley-Tukey (CT) butterflies in the NTT and Gentleman-Sande (GS) butterfly in the INTT and implement them in assembly, as shown in Algorithm 1 and 2. This approach applies a redundant representation for elements and loose the range of input to $[0, 4q]$ for NTT and $[0, 2q]$ for INTT, and $q < \frac{\beta}{4}$, where $\beta = 2^{64}$ denotes the machine word size. In the CT butterfly we first conduct a conditional subtraction for $X \geq 2q$, and then compute $T \leftarrow (WY - \lfloor W'Y/\beta \rfloor q) \bmod \beta$. In the end it output $X' \leftarrow X + T$ and $Y' \leftarrow X - T + 2q$, where $0 \leqslant X', Y' < 4q$. For the GS butterfly it conditionally subtract $2q$ from X to get X', compute $T \leftarrow X - Y + 2q$ and output X' and $Y \leftarrow (WT - \lfloor W'T/\beta \rfloor q)$. The division and rounding operations in both algorithms can be realized by taking the high bits through the instruction `mul.hi.u64`.

4.3 Memory-efficient design of the AKS kernel

The constant term extraction kernel contains $\log N$ automorphism and key switching, which denoted as the AKS kernel. We adopt the RNS-friendly key switching proposed in [15]. The main idea behind this method

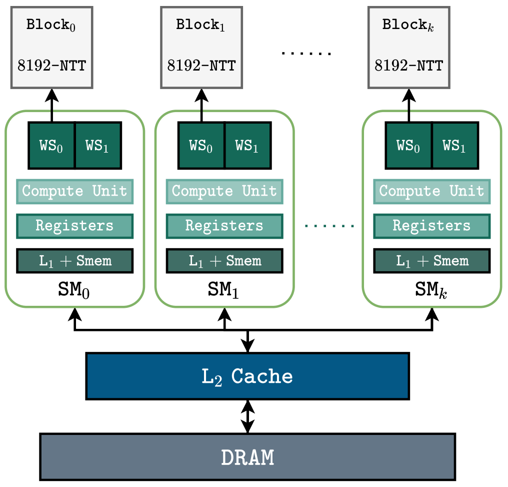

Fig. 4: The process and memory access pattern of NTT.

Algorithm 1 CT Butterfly in CUDA PTX Assembly

```
Input: $0 \leqslant X, Y < 4p$, $\beta = 2^{64}$, $q < \beta/4$, $0 < W < q$, $W' = \lfloor W\beta/q \rfloor$
Output: $X' = X + WY$, $Y' = X - WY$, $0 \leqslant X', Y' < 4q$
 1: setp.hs.u64   $d, X, 2p$                ▷ $d = X \geqslant 2p$
 2: @d  sub.u64   $X, X, 2p$                ▷ if $d$, then $X = X - 2p$
 3:     mov.u64   $U, X$                    ▷ $U = X$
 4: mul.hi.u64    $Q, W', Y$                ▷ $Q = \lfloor W'Y/\beta \rfloor$
 5: mul.lo.u64    $T, W, Y$                 ▷ $T = WY \bmod \beta$
 6: mul.lo.u64    $Q, Q, p$                 ▷ $Q = Qp \bmod \beta$
 7:     sub.u64   $T, T, Q$                 ▷ $T = T - Q$
 8:     add.u64   $X', X, T$                ▷ $X' = X + T$
 9:     add.u64   $X, X, 2p$                ▷ $X = X + 2p$
10:     sub.u64   $Y', X, T$                ▷ $Y' = X - T$
```

Algorithm 2 GS Butterfly in CUDA PTX Assembly

```
Input: $0 \leqslant X, Y < 2q$, $\beta = 2^{64}$, $q < \beta/4$, $0 < W < q$, $W' = \lfloor W\beta/q \rfloor$
Output: $X' = X + Y$, $Y' = W(X - Y)$, $0 \leqslant X', Y' < 2q$
 1:     add.u64   $X', X, Y$                ▷ $X' = X + Y$
 2: setp.hs.u64   $d, X', 2p$               ▷ $d = X' \geqslant 2p$
 3: @d  sub.u64   $X', X', 2p$              ▷ if $d$, then $X' = X' - 2p$
 4:     add.u64   $X, X, 2p$                ▷ $X = X + 2p$
 5:     sub.u64   $T, X, Y$                 ▷ $T = X - Y$
 6: mul.hi.u64    $Q, W', T$                ▷ $Q = \lfloor W'T/\beta \rfloor$
 7: mul.lo.u64    $T, W, T$                 ▷ $T = WT \bmod \beta$
 8: mul.lo.u64    $Q, Q, p$                 ▷ $Q = Qp \bmod \beta$
 9:     sub.u64   $Y', T, Q$                ▷ $Y' = T - Q$
```

Algorithm 3 Constant term extraction

```
1: function AKS($c = (c_0, c_1) \in \mathcal{R}^2_{Q'}, \mathbf{K} \in \mathcal{R}^{d\times 2}_{pQ}$)
2:     $(c'_0, c'_1) \leftarrow (\tau_\gamma(c_0), \tau_\gamma(c_1))$                ▷ Apply $\tau_\gamma$
3:     $\mathbf{C}_1 \leftarrow$ ModUp(RNSDecompose($c'_1$))
4:     $(c''_0, c''_1) \leftarrow (\langle \mathbf{C}_1, \mathbf{K}[0]\rangle, \langle \mathbf{C}_1, \mathbf{K}[1]\rangle)$
5:     $(\hat{c}_0, \hat{c}_1) \leftarrow$ (ModDown($c''_0$), ModDown($c''_1$))
6:     $(c_0, c_1) \leftarrow (c'_0 + \hat{c}_0, \hat{c}_1)$                ▷ Switch key
7:     return $(c_0, c_1)$
8: function CTX($c \in \mathcal{R}^2_{Q'}, \mathbf{swk}$)
9:     $c' \leftarrow c$
10:    for $i \in (0, 1, \cdots, \log N - 1)$ do
11:        $c' \leftarrow c' \oplus$ AKS($c', \mathbf{swk}[i]$)
12:    return $c'$
```

is to decompose a ciphertext that consists of polynomials with large coefficients into a set of smaller ones to accelerate the computation. Since the inner production requires the two operands under the same modulus, the RNS modulus switching technique is introduced to change the modulus and at the same time reduce the noise. The procedure is illustrated in Algorithm 3.

We apply many approaches for performance optimizations and memory management. Generally, we fuse multiple GPU kernels into a single kernel and reduce the dimension of the grid. The combination of the techniques offers a reduction in the time elapsed in waiting for the IO and synchronizations between threads or blocks. The number of blocks in the idle state during the execution is minimized, leading to the optimal resource utilization.

Kernel fusing. We fuse all the subprocedures, including automorphism τ_γ, ModUp, ModDown, RNSDecompose, NTT, and INTT, which is described in Algorithm 3, to obtain an AKS kernel, which first applies automorphism to a ciphertext that contains two polynomials, and then converts its corresponding key to the correct secret key through key switching. Our implementation adopts a compact design by copying a set of residues with the same modulus of a polynomial under the RNS representation into the shared memory of a block, which then performs all the operations on the residues. Thereby, the data interaction between global memory and shared memory for transferring the residues is reduced to once in the AKS kernel, instead of the case before fusing that requires multiple transfers of intermediate results. Another benefit is that when computing the matrix multiplication, the partially accumulated sum can be stored in the registers. Through this way, the data can be preserved in the shared memory and registers that have lower IO latency, of which the data access rate is faster, eliminating all redundant data access.

Grid dimension reduction. The size of the operands changes after applying the ModUp, ModDown, and RNSDecompose. Let k denote the number of current moduli, and r denote the RNS-decomposition number. In detail, the NTT procedure takes $k \cdot r$ polynomials as an input, while the number is $2 \cdot r$ for INTT and $2 \cdot k$ for automorphism τ_γ and ModDown. This makes it necessary to maintain the compatibility of each subprocedure during the processing of key switching, for which we have two solutions.

The basic idea is to instantiate a grid with $N/2^{11} \cdot k \cdot r$ blocks and each block has 1024 threads to execute the operations. The downside to this method is that resources may be underused and wasted, for the key switching and modulus switching manipulate different size of inputs. Meanwhile, the resource occupation grows exponentially with the polynomial size, as r is often set as $k + 1$.

These considerations lead us to focus on downgrading the dimensions of the grid. In our implementation, we form the grid with $1 \cdot 2 \cdot r$ blocks, which is compatible with all procedures (as $2 < k < r$) and can complete all tasks by recursively calling. This method significantly increases resource utilization because there are at most 2 blocks staying in the idle state during the lifetime of the kernel. Additionally, the resource occupation grows linearly with r, regardless of the polynomial degree size. This also allows us to reserve more memory

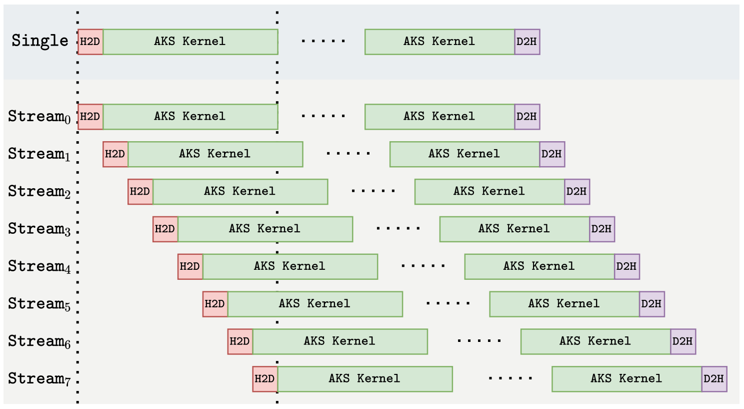

Fig. 5: Comparison of single-stream and multi-stream configurations.

resource to hold the intermediate values, e.g., the partial sum, which we store in the registers of each thread. One thing to note is that the dimensional reduction will not decrease the parallelism, as the number of threads executed by a SM concurrently is limited.

4.4 Management of multiple CUDA streams

During the execution, all kernels will be invoked sequentially on a single stream if unspecified, i.e., the $\log N$ AKS kernels in the CTX. However, the wait time is unnecessary in some cases when the operations are not data dependent, such as data transmission and computation.

To hide the memory latency of `h2d` and `d2h`, we define 8 streams and let the AKS kernels run on different streams simultaneously and interleaved. Fig. 5 gives a visualization of how our approach makes a difference to the performance. We precisely measure the IO latency and the execution time of AKS and set the number of streams to 8. Through this approach, we minimize the time interval between the `H2D` of the last stream and the first stream. In the single-stream setting, all operations follow the issue-order, and the next kernel will wait until the execution of the previous ends and data transmission completes. Sequential execution limits flexibility and introduces a lot of unnecessary wait time. The multi-stream configuration allows operations to run concurrently and through interleaving the time of data access is well hidden, leading to a great boost of the overall execution time and better resource utilization.

5 Results and Comparison

5.1 Testing environment

All experiments were performed on an NVIDIA Tesla V100S PCIe with 5120 cores and an Intel Xeon Silver 4210R@ 2.40 GHz CPU with 10 cores. Table 3 shows the hardware specifications of the platform. This type of GPU has 80 SMs and each SM is equipped with 2 WSs. The on-board global memory is 32 GB. The shared memory and L1 cache assigned to each block is 128 KB in total, and the maximum number of 32-bit registers that a thread can use is 255.

We use g++ 7.5 as the C++ compiler and NVIDIA Cuda 11.6 as the GPU compiler driver. The operating system is Ubuntu 20.04.3 LTS (GNU/Linux 5.4.0-97-generic x86_64). We integrated our GPU implementation into Microsoft SEAL library v3.7 [24] and profiled both CPU and GPU implementations with the same build and execution environment.

We conduct the benchmark in two levels: low-level arithmetic operations and high-level kernels. Comparisons of implementations on different platforms and with different parameters are also provided. The reported running time is an average of 1000 runs and is measured in microseconds.

Table 3: Testing environment specifications

CPU	Intel Xeon Silver 4210R @ 2.40 GHz						
GPU	NVIDIA Tesla V100S PCIe						
Info.	SM	WSs	Freq.	Memory			
	count	per SM	(GHz)	Global	Registers	Shared + L1	L2
	80	2	1.6	32GB	255	128KB	6MB

Table 4: Benchmark and Comparison of our NTT implementation and related works

Work	$(N, \|q_i\|)$	NTT (ms)	INTT (ms)	Type	Platform
[1]	$(1024, 62)$	1.16	-	AVX2	Macbook Air
[5]	$(4096, 50)$	5.81	5.72	AVX512	3rd Gen Intel Xeon
[11]	$(8192, 24)$	0.84	0.98	GPU	NVIDIA GeForce GTX690
	$(16384, 24)$	1.78	2.09		
	$(32768, 24)$	6.24	6.86		
[2]	$(8192, 62)$	0.031	0.033	GPU	NVIDIA Tesla K80
	$(16384, 62)$	0.039	0.041		
	$(32768, 62)$	0.048	0.050		
CPU baseline	$(8192, 64)$	0.0984	0.1098	C	Intel Xeon 4210R
HEXL	$(8192, 64)$	0.0527	0.0596	AVX512	Intel Xeon 4210R
Our work	$(8192, 64)$	0.0354	0.0373	GPU	NVIDIA Tesla V100S

5.2 cuXCMP

Table 4 summarizes the performance of our optimized NTT and INTT and other related works. As our cuXCMP is instantiated with $N = 8192$, we first compare it with the baseline implementation in C and AVX512 with the fixed parameter set $(N, |q_i|) = (8192, 64)$. By fully utilizing the shared memory, we beat the C and AVX512 implementations by 2.78× and 1.49× for NTT, and 2.94× and 1.60× for INTT, respectively. Compared to other related works, our implementation also gains a speedup, and the result we obtain is similar to [2], while our design is compact with just one block usage.

We also evaluate the impact of our memory-centric optimizations and stream managements. Table 5 presents the execution times of both the optimized kernels and the scheme. With kernel fusing and grid dimension reduction, our AKS kernel outperforms the C and AVX512 baselines by 23.71× and 18.29× respectively. To measure the effectiveness of our optimization, we execute the original XCMP scheme on the same platform and instantiate the fully homomorphic scheme with the BFV and keep the parameters consistent. Compared to the origin XCMP scheme with one compared number, our optimizations result in a 1.83× and 1.42× speedup for C and AVX512. Meanwhile, by setting the scale of comparisons to be multiples of 8, which is the number of launched CUDA streams, we achieve an increasing speedup ratio. Our cuXCMP outperforms the baseline by 1.99×, which nearly halves the running time of the original implementation. This demonstrates the effectiveness of our optimizations.

5.3 PPDT evaluation

To demonstrate our optimizations, we follow the work [21] and integrate cuXCMP into the PPDT proposed in [25], which use the DGK's private comparison protocol [12]. The original protocol suffers from two drawbacks. First, both parties must be online throughout the evaluation process and need to interact to get the final

Table 5: Benchmark and Comparison of our implementation and baselines

Benchmark		XCMP		cuXCMP
(ms)		CPU baseline	AVX512	GPU
AKS		3.32 (**23.71**×)	2.56 (**18.29**×)	0.14
1 Cmp.	CTX	40.81 (**15.58**×)	30.79 (**11.75**×)	2.62
	Scheme	116.56 (**1.83**×)	90.88 (**1.42**×)	63.86
8 Cmps.	CTX	327.91 (**85.39**×)	242.09 (**63.04**×)	3.84
	Scheme	915.89 (**1.91**×)	727.31 (**1.52**×)	479.18
32 Cmps.	CTX	1319.38 (**112.00**×)	960.39 (**81.53**×)	11.78
	Scheme	3674.07 (**1.99**×)	2796.10 (**1.51**×)	1850.20

result. Second, the decision tree delegated to the server cannot be encrypted, which leaks the evaluation model. Through instantiating the private comparison with cuXCMP, we alleviate this problem and achieve a new noninteractive protocol that allow extended input domain, offline execution, and privacy of the decision tree.

We trained decision tree models through the scikit-learn library on three real data sets, heart-disease (HD), housing (HG), and spambase (SP) from the UCI repository

we evaluate our scheme in the semi-honest model on the three real datasets from the UCI repository [13], i.e., Heart-disease (HD), Housing (HS), Spambase (SPA), as well as the artificial data used in [21].

6 Conclusions

In this work, we present a private comparison scheme cuXCMP, as well as optimization techniques tailored for CUDA-enabled GPUs. Our scheme allows encrypted negative and float inputs and offers noninteractive and fully output expressive features. Meanwhile, we propose a set of optimizations to boost up the performance, including memory-efficient low-level arithmetic, kernel dimension and IO latency reduction, and finely tuned stream management. To demonstrate the efficiency, we provide performance benchmarks on NVIDIA Tesla V100S. The results shows that our solution can make a difference to the scheme performance.

References

1. Aguilar-Melchor, C., Barrier, J., Guelton, S., Guinet, A., Killijian, M.O., Lepoint, T.: Nfllib: Ntt-based fast lattice library. In: Topics in Cryptology - CT-RSA 2016. Lecture Notes in Computer Science, vol. 9610, pp. 341–356 (2016). https://doi.org/10.1007/978-3-319-29485-8_20
2. Al Badawi, A., Veeravalli, B., Mun, C.F., Aung, K.M.M.: High-performance fv somewhat homomorphic encryption on gpus: An implementation using cuda. IACR Transactions on Cryptographic Hardware and Embedded Systems **2018**(2), 70–95 (2018). https://doi.org/10.13154/tches.v2018.i2.70-95
3. Albrecht, M.R., Player, R., Scott, S.: On the concrete hardness of learning with errors. J. Math. Cryptol. **9**(3), 169–203 (2015), `http://www.degruyter.com/view/j/jmc.2015.9.issue-3/jmc-2015-0016/jmc-2015-0016.xml`
4. Alperin-Sheriff, J., Peikert, C.: Practical bootstrapping in quasilinear time. In: Advances in Cryptology - CRYPTO 2013. Lecture Notes in Computer Science, vol. 8042, pp. 1–20 (2013). https://doi.org/10.1007/978-3-642-40041-4_1
5. Boemer, F., Kim, S., Seifu, G., Souza, F.D.M.d., Gopal, V.: Intel hexl: Accelerating homomorphic encryption with intel avx512-ifma52. In: WAHC '21: Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography. pp. 57–62 (2021). https://doi.org/10.1145/3474366.3486926
6. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (leveled) fully homomorphic encryption without bootstrapping. In: Innovations in Theoretical Computer Science 2012. pp. 309–325 (2012). https://doi.org/10.1145/2090236.2090262

7. Catrina, O., de Hoogh, S.: Improved primitives for secure multiparty integer computation. In: Security and Cryptography for Networks. pp. 182–199. Lecture Notes in Computer Science (2010). https://doi.org/10.1007/978-3-642-15317-4_13
8. Chen, H., Dai, W., Kim, M., Song, Y.: Efficient homomorphic conversion between (ring) lwe ciphertexts. In: Applied Cryptography and Network Security. pp. 460–479. Lecture Notes in Computer Science (2021). https://doi.org/10.1007/978-3-030-78372-3_18
9. Cheon, J.H., Kim, A., Kim, M., Song, Y.: Homomorphic encryption for arithmetic of approximate numbers. In: Advances in Cryptology - ASIACRYPT 2017. Lecture Notes in Computer Science, vol. 10624, pp. 409–437 (2017). https://doi.org/10.1007/978-3-319-70694-8_15
10. Couteau, G.: Efficient secure comparison protocols. IACR Cryptol. ePrint Arch. p. 544 (2016), http://eprint.iacr.org/2016/544
11. Dai, W., Sunar, B.: cuhe: A homomorphic encryption accelerator library. In: Cryptography and Information Security in the Balkans - Second International Conference, BalkanCryptSec 2015. Lecture Notes in Computer Science, vol. 9540, pp. 169–186 (2015). https://doi.org/10.1007/978-3-319-29172-7_11
12. Damgård, I., Geisler, M., Krøigaard, M.: A correction to 'efficient and secure comparison for on-line auctions'. Int. J. Appl. Cryptogr. **1**(4), 323–324 (2009). https://doi.org/10.1504/IJACT.2009.028031, https://doi.org/10.1504/IJACT.2009.028031
13. Dua, D., Graff, C.: UCI machine learning repository (2017), http://archive.ics.uci.edu/ml
14. Ducas, L., Micciancio, D.: Fhew: Bootstrapping homomorphic encryption in less than a second. In: Advances in Cryptology - EUROCRYPT 2015. Lecture Notes in Computer Science, vol. 9056, pp. 617–640 (2015). https://doi.org/10.1007/978-3-662-46800-5_24
15. Han, K., Ki, D.: Better bootstrapping for approximate homomorphic encryption. In: Topics in Cryptology - CT-RSA 2020. pp. 364–390. Lecture Notes in Computer Science (2020). https://doi.org/10.1007/978-3-030-40186-3_16
16. Harvey, D.: Faster arithmetic for number-theoretic transforms. Journal of Symbolic Computation **60**, 113–119 (2014). https://doi.org/10.1016/j.jsc.2013.09.002
17. Ishimaki, Y., Yamana, H.: Non-interactive and fully output expressive private comparison. In: Progress in Cryptology - INDOCRYPT 2018. pp. 355–374. Lecture Notes in Computer Science (2018). https://doi.org/10.1007/978-3-030-05378-9_19
18. Ishimaki, Y., Yamana, H.: Faster homomorphic trace-type function evaluation. IEEE Access **9**, 53061–53077 (2021). https://doi.org/10.1109/access.2021.3071264
19. Jung, W., Kim, S., Ahn, J.H., Cheon, J.H., Lee, Y.: Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with gpus. IACR Transactions on Cryptographic Hardware and Embedded Systems **2021**(4), 114–148 (2021). https://doi.org/10.46586/tches.v2021.i4.114-148
20. Kolesnikov, V., Sadeghi, A., Schneider, T.: Improved garbled circuit building blocks and applications to auctions and computing minima. In: Garay, J.A., Miyaji, A., Otsuka, A. (eds.) Cryptology and Network Security, 8th International Conference, CANS 2009, Kanazawa, Japan, December 12-14, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5888, pp. 1–20. Springer (2009). https://doi.org/10.1007/978-3-642-10433-6_1, https://doi.org/10.1007/978-3-642-10433-6_1
21. Lu, W.j., Zhou, J.j., Sakuma, J.: Non-interactive and output expressive private comparison from homomorphic encryption. In: Proceedings of the 2018 on Asia Conference on Computer and Communications Security, AsiaCCS 2018. pp. 67–74 (2018). https://doi.org/10.1145/3196494.3196503
22. Lyubashevsky, V., Peikert, C., Regev, O.: On ideal lattices and learning with errors over rings. In: Advances in Cryptology - EUROCRYPT 2010. Lecture Notes in Computer Science, vol. 6110, pp. 1–23 (2010). https://doi.org/10.1007/978-3-642-13190-5_1
23. NVIDIA: Cuda c++ programming guide (v11.6.0) (2022), https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
24. Microsoft SEAL (release 3.7). https://github.com/Microsoft/SEAL (Apr 2022), microsoft Research, Redmond, WA.
25. Tai, R.K.H., Ma, J.P.K., Zhao, Y., Chow, S.S.M.: Privacy-preserving decision trees evaluation via linear functions. In: Computer Security - ESORICS 2017. pp. 494–512. Lecture Notes in Computer Science (2017). https://doi.org/10.1007/978-3-319-66399-9_27
26. Tueno, A., Boev, Y., Kerschbaum, F.: Non-interactive private decision tree evaluation. In: Data and Applications Security and Privacy XXXIV. pp. 174–194. Lecture Notes in Computer Science (2020). https://doi.org/10.1007/978-3-030-49669-2_10
27. Yao, A.C.: How to generate and exchange secrets. In: 27th Annual Symposium on Foundations of Computer Science (sfcs 1986). pp. 162–167 (1986). https://doi.org/10.1109/sfcs.1986.25