

CAT : A GPU-Accelerated FHE Framework with Its Application to High-Precision Private Dataset Query

Qirui Li
IDEA Institute

Rui Zong
IDEA Institute

Abstract—We introduce an open-source¹ GPU-accelerated fully homomorphic encryption (FHE) framework *CAT*, which surpasses existing solutions in functionality and efficiency. *CAT* features a three-layer architecture: a foundation of core math, a bridge of pre-computed elements and combined operations, and an API-accessible layer of FHE operators. It utilizes techniques such as parallel executed operations, well-defined layout patterns of cipher data, kernel fusion/segmentation, and dual GPU pools to enhance the overall execution efficiency. In addition, a memory management mechanism ensures server-side suitability and prevents data leakage.

Based on our framework, we implement three widely used FHE schemes: CKKS, BFV, and BGV. The results show that our implementation on Nvidia 4090 can achieve up to $2173\times$ speedup over CPU implementation and $1.25\times$ over state-of-the-art GPU acceleration work for specific operations. What's more, we offer a scenario validation with CKKS-based Privacy Database Queries, achieving a $33\times$ speedup over its CPU counterpart. All query tasks can handle datasets up to 10^3 rows on a single GPU within 1 second, using 2-5 GB storage.

Our implementation has undergone extensive stability testing and can be easily deployed on commercial GPUs. We hope that our work will significantly advance the integration of state-of-the-art FHE algorithms into diverse real-world systems by providing a robust, industry-ready, and open-source tool.

I. INTRODUCTION

Fully homomorphic encryption (FHE), first proposed in 1978 [39], enables directly operating on encrypted data while yielding the same encrypted results as if the operations were run on plaintexts. FHE stands as a powerful tool in privacy computing, offering secure outsourced computation without revealing any data. In 2009, Gentry [19] gave the first FHE construction based on the concept of bootstrapping, which homomorphically evaluates the decryption process. Since then, many innovative works have been proposed, and the most distinguished FHE schemes can be classified into three categories: 1) CKKS[11] for float number arithmetic, 2) BFV [16] and BGV [9] for integer arithmetic, 3) FHEW [15] and TFHE [14] for Boolean operations.

While conceptually ingenious, FHE is still far from widespread adoption due to challenges in computational efficiency, data expansion, and functional limitations. Even with the most highly optimized CPU-based FHE libraries, computations on encrypted data still cost $10^3\times$ to $10^6\times$ more time and $10^2\times$ to $10^5\times$ more memory than equivalent

plaintext computations [17]. To address these issues, FHE SIMD (Single Instruction Multiple Data) operation [43], [11] has been proposed, which encrypts multiple plaintexts into a single ciphertext and enables simultaneous evaluation of subsequent operations. Another way to enhance FHE computational efficiency is through highly parallel hardware-based implementations. There have been numerous attempts and research efforts based on different types of hardware, e.g., FPGA [38], [40], [29], [37], [30], [34], GPU [17], [6], [7], [22], [2], [4], [27], [21], [31], [35], and ASIC [23], [24], [28], [18], [41], [42], [1]. Most FPGA-based works focus on accelerating specific underlying mathematical operations or HE operations within the algorithm. ASIC-based algorithm implementations currently offer the highest performance and support bootstrapping operations, but their high cost and limited flexibility hinder real-world adoption. GPU-based implementations face on-chip memory limitations that restrict kernel stacking for certain functions.

In this paper, we present an all-in-one GPU-accelerated FHE implementation spanning from fundamental hardware realization to concrete application scenarios. We refer to our framework as *CAT*, an acronym for Cipher-Acceleration-Textile. This nomenclature reflects its exquisite proficiency in overseeing the entire lifespan of accelerated FHE workloads. Within our framework, cipher data seamlessly flows akin to shuttles and threads in a textile machine. Our key contributions are threefold:

1) Three-Layer Framework

We propose a three-layer framework for GPU-based FHE implementation, where the foundational layer provides core mathematical operations; the intermediate layer bridges with pre-computed elements and combined operations; and the topmost layer exposes FHE operations, such as encryption and multiplication, as API interfaces to users. To enhance efficiency, scalability, and security across all layers, we equip them with innovative techniques: parallel executions, well-defined data layouts, kernel fusion/segmentation. And we introduce two GPU pools, the Stream Pool leverages these techniques to maximize performance, while the Memory Pool ensures memory usage efficiency for server-side suitability.

2) High Performance Implementation

Unveiling the power of our three-layer framework, we successfully implement three widely-used FHE schemes (CKKS, BFV, BGV) with remarkable performance improvements. To validate their practicality,

¹<https://github.com/Rayman96/CAT>

we further deploy these implementations on several Nvidia commercial GPUs, demonstrating their potential across different GPU architectures. On Nvidia 4090, our implementations achieve astounding speedups of $2173\times$ over the CPU-based SEAL library and $1.25\times$ over the state-of-the-art public GPU implementations for specific operations;

3) Scenario Validation

To demonstrate its real-world applicability, we implemented a private database query protocol based on our high-performance CKKS implementation. This protocol achieves functional completeness, high precision, and high efficiency:

- Supports both search and computation in a single query;
- Achieves 32-bit precision for non-linear operations;
- Exhibits a 1-second query latency on 10^3 rows with 2-5GB of storage.

In Section II, we discuss the background knowledge and our motivations. In Section III, we give a detailed introduction of our framework. In Section IV, we introduce our PDQ protocol. In Section V, we give the implementation results. In Section VI, we summarize our work and list future plans.

II. BACKGROUND AND MOTIVATION

A. Notations

This section introduces basic notations and HE-related terminologies. Raw data is called *clear text* and it is encoded to *plaintext*. The data is converted into *ciphertext* via encryption.

RLWE is an encryption methodology widely used in HE schemes, which is the ring version of LWE-based cryptosystems.

The plaintext space is determined by an integer parameter t , a plaintext is an element of $\mathcal{R}_t = \mathcal{R}/(t\mathcal{R})$, i.e. a polynomial of degree at most $n - 1$ with coefficients in \mathbb{Z}_t . Ciphertexts are elements of $\mathcal{R}_q = \mathcal{R}/(q\mathcal{R})$, i.e. polynomials of degree at most $n - 1$ with coefficients in \mathbb{Z}_q . The parameters q and n determines the security of the underlying RLWE cryptosystem. For more information, refer to [5].

B. RNS Representation

Usually, $q >> t$, the length of q can be hundreds of bits. Directly computing on these big numbers is inefficiency. q can be a product of small pairwise coprime numbers $q = \prod_{i=1}^k q_i$. Then, the HE operations can be implemented as a Residue Number System (RNS), using the Chinese Remainder Theorem (CRT) which essentially offers a ring isomorphism $\mathbb{Z}_q \rightarrow \prod_{i=1}^k \mathbb{Z}_{q_i}$, to manipulate the large coefficients of ciphertext polynomials, as the isomorphism can be extended to polynomials: $\mathcal{R}_q \simeq \mathcal{R}_{q_1} \times \mathcal{R}_{q_2} \times \cdots \times \mathcal{R}_{q_k}$.

The RNS concept is also well-suited for coefficient level manipulation in HE operations. The level concept is important for operations such as Rescale in CKKS and ModulusSwitch in BFV/BGV.

C. FHE Hardware Acceleration

Research on hardware acceleration for FHE primarily focuses on three approaches: FPGA, ASIC, and GPU.

Many FPGA-based FHE implementations focus on optimizing some specific operations, such as the underlying mathematical operation NTT [30], [34] or HE operators such as key-switch [38] and modular multiplication [29]. In [40], the authors give an Arm-FPGA co-processor framework to achieve $13\times$ speedup of BFV compared with the CPU implementation. In [37], an HE accelerator for high-performance matrix-vector product is given and validated in multiplication-centric scenarios.

Compared to software implementations, ASIC-based implementations can achieve up to $10000\times$ speedup and thus become the most notable acceleration results. However, whether these efforts can truly advance the practical deployment of FHE requires further validation and optimization. For example, due to the integrated-into-single-chip design, the chip area of [23], [24], [28], [18], [41], [42] is large, which can be up to more than 100 mm^2 . In [1], the authors propose the first chiplet-based FHE accelerator to solve the scalability problem. However the development cycle, market readiness, and costs need further clarification and optimization, especially considering the high development cost and extended time-to-market.

Many GPU-based studies also only focus on accelerating basic operations [17], [6], [7], [22], [2], [4], [27], and excluding bootstrapping acceleration. The work [21] firstly accelerates all HE operations, including CKKS bootstrapping, and overcomes the GPU's off-chip memory bottleneck through kernel fusion, which achieve a bootstrapping process that is 242 times faster than on a CPU. However, the GPU's on-chip memory is still the major limitation restricting kernel fusion for certain functions [27]. Also, there are efforts [31], [35] targeting boolean HE schemes.

D. FHE-based Private Database Query

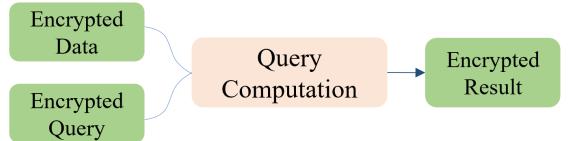


Fig. 1. Workflow for a PDQ Query

Private Database Query (PDQ) empowers secure encrypted database queries, safeguarding query confidentiality. It establishes a secure client-server protocol, enabling search and computation processes—including comparison, linear, and non-linear operations—directly on encrypted data, as shown in Fig 1. FHE offers an elegant approach for PDQ implementation. However, its noise budget faces significant depletion due to the inherently complex nature of encrypted comparison operations. This constraint has impeded advanced computations in previous works [44], [13], [20], [36], [25], [26], [12], as exemplified by Tan et al. [44], where a single comparison of 64-bit integers necessitates 10 seconds and exhausts the available noise budget, rendering bootstrapping indispensable for further operations.

Lee et al. [32] prioritized functional completeness and accuracy over efficiency, implementing PDQ using CKKS with bootstrapping. However, unlike BGV/BFV/TFHE, CKKS bootstrapping merely enables further computations without reducing ciphertext error. To compensate, they leveraged the NTL library for enhanced precision, albeit sacrificing performance compared to RNS-based methods [10]. Notably, a single bootstrapping operation in their implementation takes approximately 25 seconds.

Scheme-switching [33], [8] offers linear operations or computations up to a certain depth using SIMD-capable algorithms, but it necessitates ciphertext conversion to a different FHE algorithm with efficient bootstrapping when needed. However, these bootstrapping-efficient algorithms typically support narrow data bit-widths. In PDQ operations, ciphertext decomposition during intermediate stages is computationally expensive. Thus, fresh CKKS ciphertexts would also be constrained to limited width data, conflicting with PDQ requirements, rendering scheme-switching unsuitable for PDQ scenarios.

To date, no known work has achieved high efficiency for FHE-based PDQ while also satisfying functional completeness and accuracy requirements, e.g., completing a query under 1 second.

III. ARCHITECTURE DESIGN OVERVIEW

Our framework adopts a three-tier architecture, as shown in Figure 2, with two GPU pools providing full-process support.

The bedrock of the framework resides in the lower layer, which houses meticulously optimized fundamental mathematical computations. These include numerical operations such as addition, subtraction, and multiplication, as well as essential FHE operations like Number Theoretic Transform (NTT) and bit-reversal. This optimized foundation ensures robust performance across the entire framework.

In the middle layer, combined operations and pre-compute elements, meticulously orchestrate a symphony of efficiency for the upper layers. Two distinct types of combination strategies reside within this layer:

Theoretical Concoctions: Operations such as BFV’s multiplication using the BEHZ algorithm[3] emerge as natural amalgamations of lower-level operators, directly stemming from theoretical underpinnings.

Parallelism Prowess: Operations like ‘neg_multiply’ fuse multiple actions into a single optimized kernel. This ingenious fusion slashes latency by halving the number of GPU kernel launches, unlocking a seamless flow of computations.

The upper layer of the framework is the culmination of the framework’s design, providing a comprehensive set of APIs for users. Drawing primarily upon the middle layer’s operations and selectively tapping into lower-level functions. Not only focusing on the calculation stage, the accelerated APIs span the entire life cycle of FHE programs: GPU-powered fundamental FHE operations unleash a torrent of computational power; the encode/decode and encrypt/decrypt stages undergo meticulous optimization, ensuring a streamlined data flow; key generation periods of various types benefit from efficiency enhancements, paving the way for swifter cryptographic setups.

In addition to the three-tier structure, two crucial GPU pools support all layers: a memory pool that manages the distribution and retrieval of GPU memory (discussed in III-B), and a GPU stream pool containing a fixed number of CUDA streams constructed at the beginning of an FHE workload. We design the operators to leverage multi-stream pipelines to reduce running time, detailed in III-A5.

In the following subsections, we will delve into the details of the architecture of the proposed framework. The chapter is organized as follows: Subsection III-A introduces the details of the main body of the three-layer framework, Subsection III-B presents the design of the GPU memory pool, and Section IV briefly describes how a high-precision PDQ process runs in our framework.

A. A three leveled acceleration framework

We have provided a brief overview of the three-level acceleration framework, describing its theoretical design and composition. In this section, we will delve into key points that contribute to the efficient functionality of this layered structure, particularly in the context of a PDQ process.

1) Convert Operator to SIMD Paradigm: FHE, grounded in the Residue Number System (RNS) and the Chinese Remainder Theorem (CRT) introduced in II-B, simplifies the transformation of the calculations into the Single Instruction Multiple Data (SIMD) paradigm. The core concept involves allocating the computation of each integer to an individual GPU thread, thereby converting most operators into element-wise kernels. This approach enhances parallelism and accelerates the processing of cryptographic operations.

The detailed implementation of over 200 kernels, constituting the operators, can be explored in our open-source code. Here, we would like to spotlight several key innovations.

2) Multi Level Memory Usage on GPU: There are three widely-used levels of memory on GPU, namely global memory, shared memory, and register, ordered by their size in descending way and I/O speed in ascending order. Typically, the data we process is stored in the largest global memory most of the time, achieving remarkable acceleration already. To further enhance the performance, we optimize the utilization of shared memory and registers. Leveraging registers is facilitated by declaring const variables in a kernel when they are used multiple times. This approach allows the values to be extracted from global memory to registers, significantly speeding up the retrieval stage. Utilizing shared memory follows a similar principle to prefetching but enables the storage of a larger amount of data at one time. In the following sections, we will present two typical examples illustrating the use of different levels of memory: the calculation of modulo for large integers and the Number Theoretic Transform (NTT).

Mod for Big Integers on GPU

A significant portion of computations in FHE concludes with a mod operation, which can pose computational challenges, especially for large numbers. The % operator for large numbers in the CUDA toolkit comes with the Barrett algorithm. We have implemented a faster GPU version of the algorithm introduced in [45], demonstrated in Algorithm 1. The algorithm revolves around segmenting the dividend X based

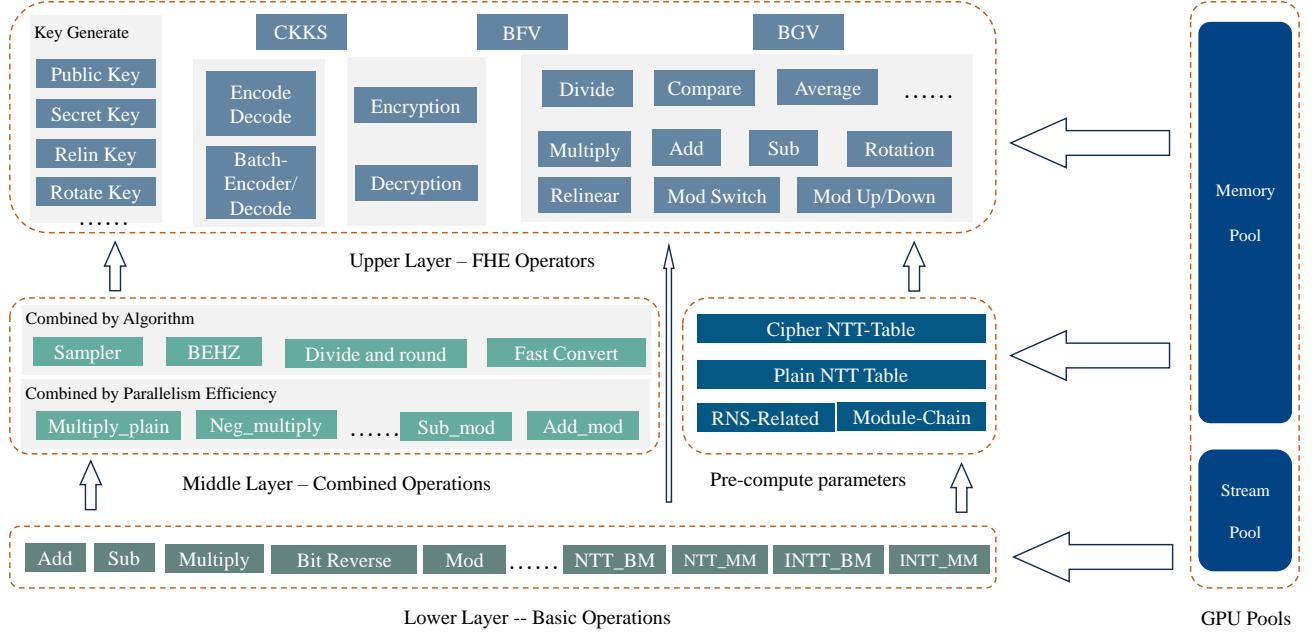


Fig. 2. Overall architecture of the GPU-accelerated FHE framework

on the length of the divisor Y and, in parallel, segmenting the divisor Y itself. Each segment is then processed by referencing a precomputed table and is prefetched into shared memory, with the final result derived through a series of addition, subtraction, and bit-shift operations. Key optimizations have been integrated into our GPU implementation:

- 1) Memory Transfer Optimization: The precomputed table contents are prefetched from global memory to shared memory, boosting access speed, with fully used registers.
- 2) Batch Modulus Calculations: Multiple modulus calculations are optimized by having each GPU thread compute several numbers, effectively reducing the GPU load for table lookup.
- 3) Arithmetic Operation Optimization: Specific arithmetic operations are optimized to minimize arithmetic instruction overhead.
- 4) Conditional Declaration: Certain cases are explicitly declared during table lookup, enhancing overall computational efficiency.

NTT&INTT

The NTT operation stands out as a pivotal component in FHE, and various endeavors aim to enhance its computational efficiency. A prominent approach involves the utilization of the butterfly method (BM) and the exploration of strategies to eliminate certain intermediate results or reduce the number of calculation steps. In contrast, approaches such as TensorFHE adhere to the original definition of NTT, emphasizing the maximization of matrix multiplication (MM) capabilities on GPUs. The BM version offers a more generalized solution, requiring computation in $O(\log n)$ steps, yet it might not fully exploit the GPU's inherent computational power. Conversely, the MM version has the advantage of leveraging the giant amount of computation cores of GPU's architecture, achieving considerable efficiency even with computational complexity of

$$O(n^2).$$

To create a framework that exhibits greater versatility across various GPU architectures, we have implemented both types of NTT methods, denoted as NTT_BM (butterfly method version) and NTT_MM (matrix multiplication version), along with their respective inverse counterparts. The implementation of NTT_MM on standard CUDA cores involves efficient matrix multiplication utilizing shared memory and registers. We employ a trick that when the degree of the polynomial is less than 1024, NTT_MM is used, and NTT_BM is chosen otherwise. This adaptive strategy ensures optimal performance based on the specific characteristics of the polynomial being processed.

3) Kernel Fusion and Segmentation: In the middle and upper levels, kernels or operators from lower levels are amalgamated to complete the desired function. Although a straightforward approach involves initiating kernels sequentially, there exist opportunities for enhancements to further elevate overall performance by strategically fusing or segmenting kernels in specific situations. Such optimizations contribute to a more streamlined and efficient execution of the computational tasks at higher levels of the framework.

Kernel Fusion

Kernel fusion merges multiple kernels into one, thus reducing intermediate results, kernel launches, and re-computation.

Take multiplication in CKKS as an example, the multiply of data X and Y involves five stages: four calculating combinations of (X_0, X_1) and (Y_0, Y_1) , and one adding X_0Y_1 and X_1Y_0 together. While executing these stages separately requires storage for intermediate results, kernel fusion can optimize this process. We can fuse the stages of generating X_0Y_1 , X_1Y_0 and their addition into one, directly storing $X_0Y_1 + X_1Y_0$ in its destination. Even more, we can fuse all five stages, reducing kernel launches to 1/5 and eliminating

Algorithm 1 Calculation of modulus for two big integers on GPU using method in [45]

```

1: procedure SET_HIGHER_BITS_0(T)
2:   ploy_mod_Y  $\leftarrow (2^{len_Y} \bmod Y)$ 
3:   while ( $T \geq (1 \ll len_Y)$ ) do
4:      $T \leftarrow T - (1 \ll len_Y) + ploy\_mod\_Y$ 
5:   end while
6: end procedure

7: procedure MODULUS
8:   shift1st  $\leftarrow 64 \div len_Y * len_Y$ 
9:   shift2ed  $\leftarrow ((len_Y - 1) \bmod len_{look\_up}) + 1$ 
10:  idx_shift1st  $\leftarrow len_Y - shift_{2st}$ 
11:  idx_shift2ed  $\leftarrow len_Y - len_{look\_up}$ 
12:  MAX_2ED_SHIFT  $\leftarrow \lceil len_Y \bmod len_{look\_up} \rceil$ 
13:  SHARED_SIZE ==  $2^{len_{look\_up}}$ 

14:  iinit  $\leftarrow (blockIdx.x \times blockDim.x + threadIdx.x) \times P$ 
15:  shared uint64_t shared_table[SHARED_SIZE]
16:  if threadIdx.x < table_size then
17:    shared_table[threadIdx.x]  $\leftarrow table[threadIdx.x]$ 
18:  end if
19:  sync threads

20:  for  $i \leftarrow i_{init}$  to  $i_{init} + P$  and  $i < size$  do
21:    numX  $\leftarrow X[i]$ 
22:    T  $\leftarrow numX \gg shift_{1st}$ 
23:    for shift  $\leftarrow shift_{1st} - len_Y$  down to 0 by lenY do
24:      next  $\leftarrow (numX \gg shift) \& digits$ 
25:      idx  $\leftarrow T \gg idx\_shift_{1st}$ 
26:      T  $\leftarrow T \ll shift_{2ed}$ 
27:      if idx  $\neq 0$  then
28:        T  $\leftarrow T + shared\_table[T]$ 
29:        Set_higher_bits_0(T)
30:      end if
31:      for j  $\leftarrow 1$  up to MAX_2ED_SHIFT by 1 do
32:        idx  $\leftarrow T \gg idx\_shift_{2ed}$ 
33:        T  $\ll len_{look\_up}$ 
34:        T  $\leftarrow T \& digits + shared\_table[idx]$ 
35:        Set_higher_bits_0(T)
36:      end for
37:      T  $\leftarrow T + next$ 
38:      Set_higher_bits_0(T)
39:    end for
40:    result[i]  $\leftarrow (T \geq Y) ? (T - Y) : T$ 
41:  end for
42: end procedure
```

extra memory needs.

Kernel Segmentation

While kernel fusion excels at streamlining short, intensive computations, kernel segmentation shines in scenarios with extended multi-stream workflows. In such cases, synchronous calculation across multiple streams using smaller kernels can outperform a single, massive kernel launch.

In operations like NTT and INTT, which require processing data on each modulus, segmentation often outperforms a single large kernel. While a monolithic kernel might reduce launch overhead, internal linear execution can hinder performance.

Instead, breaking the kernel into modulus-specific cases, assigning them to separate streams, and launching them concurrently often leads to remarkable efficiency gains. Section III-A5 delves further into multi-stream usage and stream pools.

Trade off Between Fusion and Segmentation

The optimal choice between kernel fusion and segmentation hinges on a delicate trade-off. Kernel fusion outperforms when multiple kernels share redundant calculations and intermediate results, effectively streamlining the process. Conversely, kernel segmentation excels when individual kernels exhibit low register requirements, lengthy execution times, and parallelizable tasks across distinct data segments, enabling concurrent execution for enhanced performance.

4) Data layout and Indexing: To address the performance costs associated with data transfers and non-continuous indexing, we've designed the *C_Data* structure (Fig.3). This structure optimizes memory usage and access patterns, mitigating these challenges. It features two key parameters—*Capacity* and *Size*—along with three supporting size-related parameters: *Size_{poly}*, *Size_{Modulus}*, and *Size_{Data}*. The *Size* parameter, calculated as the product of these three supporting sizes, determines the memory allocation for a single encrypted data item.

C_Data instances dynamically allocate and manage memory to optimize performance. When data of *Size_{init}* arrives, the instance requests a corresponding memory block from the memory pool, setting both *Capacity* and *Size* to *Size_{init}*. Upon expansion to *Size₂*, the instance returns the initial memory and requests a new block of *Size₂*, ensuring optimal memory usage. Conversely, when downsizing, the instance retains the existing memory to minimize data transfer overhead, adjusting *Size* to the smaller value while *Capacity* remains unchanged as *Size_{init}*. This approach prioritizes time reduction over memory conservation.

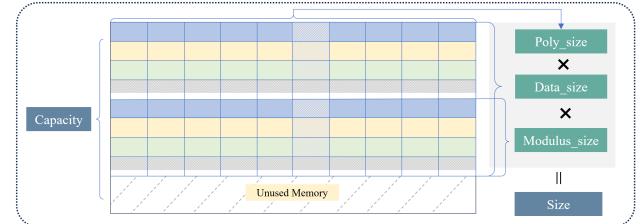


Fig. 3. Structure for storing underlying data

Within the memory, the data are logically stored in levels. Data bits belonging to the same modulus are stored continuously, one modulus after another. From a broader perspective, if a data number contains $n = Size_{Data}$ parts for longer bit storage, they will be stored separately for the convenience of calculation.

5) Utilizing Multi stream: CUDA streams offer a potent yet delicate mechanism for accelerating programs. They enable concurrent execution with explicit and carefully managed synchronization. To harness this capability responsibly, we've constructed a stream pool and meticulously integrated multi-stream operations into select operators. A common strategy involves assigning distinct streams to individual modulus, capitalizing on the contiguous data layout discussed in Section

III-A4. This modulus independence facilitates stream distribution and minimizes cross-modulus interference.

To prudently manage CUDA streams, the framework initializes a stream pool at the outset of each workload. The pool's size, denoted as S_{stream} , is determined by choosing the smaller one between the modulus chain length(*modulus_length*) and an empirically derived constant *MAX_LEN*, typically set between 3 to 5. When computations can be executed independently on each modulus, the framework judiciously launches a maximum of S_{stream} concurrent streams, ensuring optimal utilization of multi-stream capabilities. Subsequent synchronization is performed only when calculations hinge on the collective results of these streams. Section V showcases the tangible performance gains achieved through this optimized stream management approach.

B. Design of GPU-memory pool for CAT

While exploiting GPUs for FHE computation unlocks substantial speed gains, it demands caution due to potentially ballooning memory requirements. This trade-off, made for ironclad data security, becomes particularly acute in large-scale tasks. For example, a single encrypted *ciphertext*, with a default polynomial degree of 32768, occupies roughly 8 MB in its initial state. Unmanaged memory allocation in such scenarios can swiftly devour precious GPU DRAM, crippling performance and hindering large-scale FHE computations.

To address this issue, we have implemented a memory pool managing memory distribution and retrieval, allowing us to execute PDQ computing tasks composed of hundreds of operators using only 6 GB of GPU memory in serving mode.

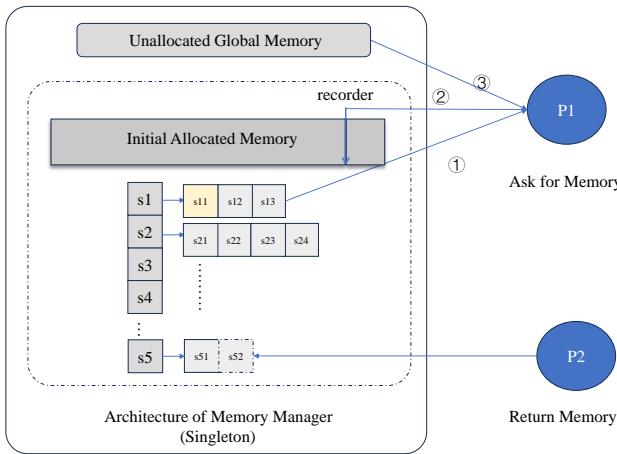


Fig. 4. Architecture of Memory pool

1) Architecture of Memory pool: Let's begin by scrutinizing the architecture of the memory pool (refer to Fig. 4). Our framework establishes an initial pool P by allocating a contiguous memory segment with a size of S , calculated by Equation 1. This allocation is accomplished directly by invoking CUDA API. Subsequently, we establish a hash map to oversee the utilization of this allocated memory segment. Within the map, the key s denotes the size of a slice of memory, and the corresponding value is a list of pointers indicating the starting points of continuous memory pieces with a size of s . Another pivotal element in the memory pool

is the one-way *recorder*, responsible for monitoring the extent of pool utilization. If we envision the initial memory as a linear railway, the *recorder* progresses along the railway in a unidirectional manner and cannot backtrack.

$$S = \text{Min}(2048, \text{Length}_{\text{modulus}} * 200) \text{ MB} \quad (1)$$

2) Details about memory pool usage: The memory pool primarily offers two essential functions: distributing and retrieving memory. We will elaborate on the processes involved in these two stages under three different conditions.

Ask for memory in normal condition

Upon receiving a request for memory of size s_{need} , the memory pool P follows a systematic process. It begins by inspecting the value list associated with the key equal to s_{need} in the hashmap. If the list is not empty, the last element d in the list is allocated, along with the memory of size s_{need} starting at position d . However, in scenarios where the list linked to the corresponding key is empty or the key is absent in the map, the position of the *recorder* is designated, and the recorder, subsequently, advances forward for a length of s_{need} . This method ensures efficient allocation and utilization of memory resources based on the specified size requirements.

Ask for memory when running out of the initial memory

If the initial memory with size S is depleted, new space for global GPU memory with size s_{need} must be requested, rendering the recorder unnecessary from this point forward.

Return memory

In cases of memory shortage, when certain data d completes its operation and will not be utilized in a subsequent operator, it is necessary to return the occupied memory s_{return} to the memory pool. The return process is straightforward: d only needs to link the starter of its memory to the value list of the key s_{return} . If the key does not exist, a new key and an empty list are generated to store it. It's noteworthy that the return process remains consistent, irrespective of the stage from which the request originates, as detailed earlier. This standardized approach ensures a systematic and efficient handling of memory resources throughout the framework.

3) Discussions about the design of memory pool: Several critical questions naturally arise regarding our memory pool design:

Is memory pool really needed?

A rudimentary solution to tackle memory shortages entails requesting and returning memory to the GPU using the CUDA API as needed. While this method is straightforward, it significantly hampers the overall computational efficiency, as illustrated in Fig 5. The memory allocation and return stages alone account for more than 60% of the time in the entire FHE program. Employing this method would essentially nullify the optimization efforts directed toward enhancing the computational operators. The adoption of a memory pool strategy becomes paramount in mitigating the computational overhead caused by frequent memory allocation and retrieval,

Why choose sizes as keys for the hash map?

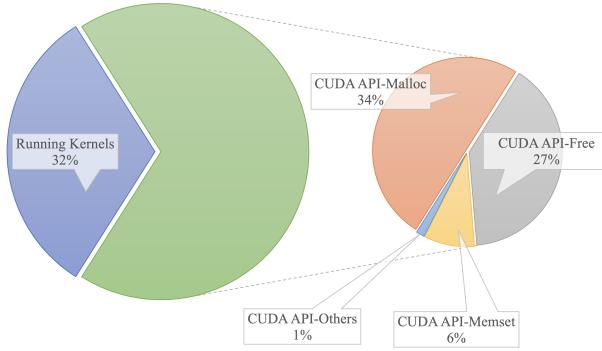


Fig. 5. Time consumption of each stages without any optimization

The CKKS/BFV/BGV schemes demonstrate that the size of encrypted and encoded data is determined by the multiplication of fixed parameters. This inherent property leads to the concentration of required memory within a limited range of size values. This trend extends beyond basic encrypted data to encompass intermediary parameters essential for operators. The structure of the memory pool can be likened to a retail dress shop exclusively stocked with fixed-size jackets. Customers simply select the size they need, and it fits seamlessly. In the realm of FHE, the necessity for a tailor to customize memory sizes for diverse needs becomes obsolete.

Will the returned memory be wasted?

Concerns may arise regarding potential imbalances in the hash map, particularly when certain keys result in much longer lists containing memory pieces used only once, because the mechanism we designed addresses this concern by ensuring that such memory remains unused until the entire program concludes. Our extensive experiments reveal a balanced distribution during both the ask and return stages for individual keys, as illustrated in Fig 6. This equilibrium implies that memory asked more often will also be returned more frequently. Additionally, we've implemented a strategy to address potential imbalance. In cases where the map becomes unbalanced, during the asking stage, one can acquire a piece of memory larger than s_{need} and closest to s_{need} to mitigate this situation.

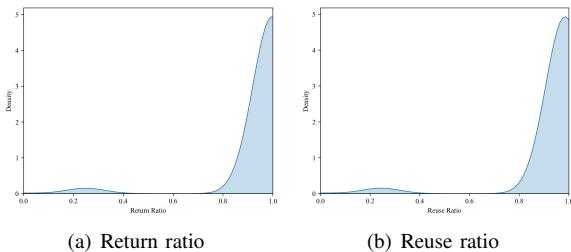


Fig. 6. Status for memory return and reuse, pic a shows the density of memory blocks return ratio, calculated by return times divided by use times. pic b checks the times for memory askers get the memory needed from the pool

IV. APPLICATION TO PDQ

We implemented the PDQ protocol based on CKKS, a scheme well-suited for multi-precision computations as dis-

cussed in [32], to effectively validate our framework's capabilities in this scenario. In this section, we first introduce our query execution framework in Section IV-A. Then, we present the specific query types implemented in Section IV-B. Finally, we give a detailed introduction of the search-and-average query based on our newly proposed two-party division protocol in Section IV-C.

A. Privacy Database Query Framework

The PDQ protocol aims to provide the client with a set of records that match its query while maintaining security. We adopt the same approach as in [46] for the security model and threat model, which guarantees both query privacy and database privacy under the semi-honest threat model. This model assumes that both the server and the user adhere to the protocol specifications, but they may attempt to learn additional information from the exchanged data.

The entire framework execution is divided into two stages: Data Upload and Query Execution, as depicted in Figure 7.

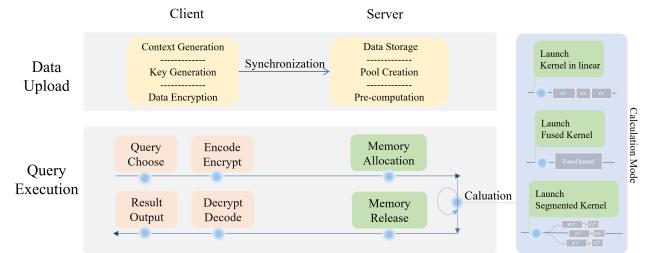


Fig. 7. Query Execution Process

In the first stage, the client generates the context that meets all query requirements. This includes selecting the appropriate RLWE parameters q and n , generating the keys, encrypting the data and uploading the ciphertext to the server, and synchronizing the context and public parameters with the server. After that, the server generates the memory pool and stream pool for all needed foundational operations based on the information uploaded by the client, as well as other parameters that can be pre-computed, such as the NTT table.

In the second stage, for each query process, the client encodes and encrypts the specific query condition and sends it to the server. The server stores the encrypted query in GPU memory allocated from the memory pool, and remains there for the query's duration. The framework optimizes calculation modes (linear, fused, segmented, or combinations) for efficient and accurate computations.

Upon completion, encrypted results will be transferred to CPU memory are sent back to the client, while preserving all the context including pre-computed parameters and the dual GPU pools. The client then gets the final result by decrypting and decoding the received ciphertext.

B. Instantiated Queries

Our implementation of the query function follows the methodology proposed in [46]. The search part is implemented by first decomposing the clear text by a small base p and then performing an FFT transformation to complete the pre-processing before encoding. The resulting values are then

used to execute the comparison with specific encrypted query condition values using the Lagrange Polynomial Interpolation technique. The computation part is implemented by performing subsequent linear and non-linear operations on the results of the search part.

In [46], they implement three queries: 1. The search query implementation extracts the data that meets specific comparison-like conditions, by multiplying the comparison results (1 or 0) with the data; 2. The search-and-sum query can be seen as performing an addition operation on the result of the search query; 3. The BMI query, which in clear text is $height/weight \cdot weight$, performs a square and division operation on the data that meets specific conditions.

We implement all these queries and give a detailed performance comparison with [46] in Section V.

C. Search-and-Average Query

Motivated by that the bootstrapping-based multiply-inverse ($1/x$) operation in [32] is inefficient, and the method adopted in [46], [11] requires the range of x to be within $p/2$ to $3p/2$ for some known p to ensure accuracy, we present a new multiply-inverse method based on multiplicative masking.

Our new method assumes that the constraint² in [32], which is not strictly necessary for our targeted single-data-holder outsourcing scenario. Therefore, the encrypted intermediate results can be returned to the data holder, while ensuring that the data holder does not perform any ciphertext evaluation operations, thus complying with the requirements of FHE-based PDQ. The two-party computation process is as shown in Algorithm 2.

Algorithm 2 The Two-Party Multiply-Inverse Method

```

1: procedure MULTIPLY-INVERSE
2:   Input:  $C_0 = Enc(x)$ 
3:   Output:  $C_2 = Enc(\frac{1}{x})$ 

4:   Server:
5:   Choose random number  $r$  and compute  $C_1 = C_0 \cdot r$ 
6:   Send  $C_1$  to Client

7:   Client:
8:   Decrypt and get  $x_1 = Dec(C_1)$ 
9:   Compute  $x_2 = \frac{1}{m_1}$ 
10:  Encrypt  $x_2$  and send  $C_1 = Enc(x_2)$  to Server

11:  Server:
12:  Compute  $C_2 = C_1 \cdot r = Enc(\frac{1}{x})$ 
13: end procedure
```

Privacy Security: The Server is only able to learn the ciphertext values, while the Client is only able to learn the plaintext value of $r \cdot x$, which is masked by a multiplication mask. Therefore, the plaintext value of x or $1/x$ is not leaked to either Client or Server.

Our method can hold significance when the cost of each interaction is substantially lower than that of performing a

²In [32], the author's constraints that *after sharing data, they must not participate in the analysis process*.

bootstrapping operation. And based on this new multiply-inverse method, we achieve a new query function: search-and-average. The experiment result shows that all query processes can be completed within one second. We will elaborate more in Section V.

V. EVALUATION AND RESULTS

We employ a series of benchmarks to demonstrate the efficacy of our framework. Including a designed script that contains almost all the basic operators, and a few functions in the scenario of PDQ to demonstrate realistic computational workloads. We run most of the experiments on an Nvidia 4090 GPU and an AMD EPYC 9654 CPU by default to show the best speed-up that we can achieve. Moreover, we have run experiments on different GPUs like Nvidia 3090, Nvidia V100, etc. to reveal the relationship between the running speed and the hardware capacity.

A. Basic FHE operators

Basic FHE operators mainly contain 5 stages: Encode, Encrypt, Calculation, Decrypt, Decode, along with Creating several keys. We have built a performance-testing script that contains all the operators that an FHE program may use. We set the degree of poly(N) to 32768, the length of modulus chain(L) to 16, and the total bit count(Q) to 881 by default and test the performance for CKKS, BFV, and BGV scheme respectively³. We ran each test 10 times and recorded the average time each operator used. The results are shown in Table I, Table II and Table III. As can be seen from the results, our framework can achieve up to $2173 \times$ when comparing the running time of common operators in FHE with Microsoft's SEAL. We achieve at least 10 times acceleration throughout all stages including encoding, encrypting, and computation, which makes the PDQ application built on FHE 20 times faster.

Besides, we have compared our framework with some early works focused on the acceleration of FHE with GPU. Most of the source codes are not publicly available, we can only compare a few of open sourced operators (Hmult, Cmult, Add, Rescale, Rotate) by [21] with our framework under default configuration, and then using the relative comparison result given by other works[17], [2] to estimate their performance.⁴. We summarize the result in Table IV, and the results shows in the table is how many times computation time will be consumed individually if we set the time cost of our framework as 1. It can be seen that we outperform other works with large gaps. The result shows that our framework has a promising acceleration ability.

B. CKKS-based PDQ Application

Benefiting from the GPU acceleration of our implementation and the newly proposed multiply-inverse method, our PDQ implementation addresses the challenges described in

³The implementation of BGV scheme is similar to CKKS's and so is the test result, we attach the result table in the appendix.

⁴In terms of achieving a balance between efficiency and capability in PDQ, we restrict the modulus length in CAT under 17. While [17], [2] only release their relative strength over [21] under 44, thereby rendering our comparison a reasonable estimation..

TABLE I. RUNNING TIME IN μ S AND SPEED UP TIMES FOR ALL CKKS OPERATORS

	Operator	SEAL(CPU)	CAT(3090)	SpeedUp	CAT(4090)	SpeedUp
CKKS N=32768, L=16, Q=881	encode	12387	2477	5x	1083	11x
	decode	59041	3691	16x	1718	34x
	encrypt	71886	1432	50x	1294	56x
	decrypt	4472	27	166x	20	223x
	add	3954	7	565x	2	1977x
	multiply	8187	32	256x	23	356x
	multiply_plain	3374	4	844x	2	1687x
	square	8269	26	318x	17	486x
	relinearize	175520	3607	49x	2617	67x
	rescale	20132	687	29x	440	46x
	rotate_vector_one_step	175741	3595	49x	2577	68x
	rotate_vector_random	876642	18440	48x	11837	74x
	complex_conjugate	174508	3495	50x	2556	68x

TABLE II. RUNNING TIME IN μ S AND SPEED UP TIMES FOR ALL BFV OPERATORS

	Operator	SEAL(CPU)	CAT(3090)	SpeedUp	CAT(4090)	SpeedUp
BFV N=32768, L=16, Q=881	encode_batch	680	366	2x	339	2x
	encrypt	57018	696	82x	427	136x
	decrypt	25688	662	39x	471	55x
	add	4130	12	344x	3	1378x
	multiply	306318	4280	72x	3675	83x
	multiply_plain	42808	762	56x	556	77x
	square	230590	7470	31x	4157	55x
	relinearize	181471	7800	23x	4607	39x
	rotate_vector_one_step	181883	3643	50x	2633	69x
	rotate_vector_random	808661	15882	51x	11299	72x
	rotate_columns	181205	3275	55x	1623	112x

TABLE III. RUNNING TIME IN μ S AND SPEED UP TIMES FOR ALL BGV OPERATORS

	Operator	SEAL(CPU)	CAT(3090)	Speed	CAT(4090)	Speed
BGV N=32768, L=16, Q=881	encode_batch	640	466	1x	371	2x
	encode_unbatch	773	527	1x	480	2x
	encrypt	98133	1432	50x	1294	76x
	decrypt	19376	762	25x	519	37
	add	4053	10	405x	6	676x
	multiply	8762	23	381x	17	515x
	multiply_plain	15433	420	37x	191	81x
	square	10865	6	1810x	5	2173x
	relinearize	193632	4979	39x	3169	61x
	rotate_rows_one_step	203200	4182	49x	2981	68x
	rotate_rows_random	872223	16847	52x	11837	74x
	rotate_columns	196284	4018	49x	2786	70x

TABLE IV. COMPARE WITH OTHER IMPLEMENTATIONS ON SEVERAL OPERATORS

	Hmult	Cmult	Add	Rescale	Rotate
TensorFHE	1.33	1.13	1.29	1.85	1.27
100x	1.39	3.25	5.54	19.5	1.30
PrivFT	4.8	1.755	2.77	42.9	-
CAT	1	1	1	1	1

Section II-D and fulfills the requirements of functional completeness, accuracy, and efficiency, without employing bootstrapping or scheme switching. Our query can support data range up to 2^{32} , where the data can be integers or real numbers. We ensure integer part accuracy for all query results, meaning multiply-inverse operations $\frac{1}{x}$ have a 32-bit binary accuracy beyond the decimal point with no range limitation of x .

As introduced in Section IV, we test our framework on four different types of PDQ queries, recording its running time and memory usage shown in Table V. Since there is no open-source industrial ready GPU framework for FHE to the best of our knowledge, we can only compare the result with the implementation using Microsoft SEAL on CPU. Besides, we have also compared with [46] as mentioned in IV-B on PDQ

with CPU on certain cases which [46] have provided their results. The query we evaluated are as listed and the number of data rows is 1024, the Col_x represents the x column in the dataset, Sum() and Avg() are the operations for summation and average, and the Index() simply fetches the ids of the rows that meet the filters.

- Index() where ($Col_a \leq Col_b$ and $Col_c \neq Col_d$)
- Sum(Col_a) where ($Col_b \leq Col_c$ and $Col_d \neq Col_e$)
- Col_a / Col_b^2 where ($Col_b \leq Col_c$)
- Avg(Col_a) where ($Col_b \leq Col_c$ and $Col_d == Col_e$)

Table V presents the results, the "Cal" in the first two columns indicates the pure calculation stage, and "Full-Task" in the third column consists of the time for Encryption, Decryption, Calculation, and Serialization to simulate the query application in real world. "1st Cal" shows the time needed for running the first time, it would be slower than later ones because of the process of building the memory pool and segmentation memory into pieces. It can be seen from the Table that our framework achieves up to 33x acceleration compared

to the implementation on CPU, and the time needed for the entire task is no more than 1 second, making it possible to be deployed into real-world applications. Moreover, Table VI records the memory occupation on GPU when running each PDQ task, as shown in the result that the queries process about 2~5 GB to finish the entire task.

TABLE V. RUNNING TIME IN mS AND SPEED UP TIMES FOR RUNNING PDQ TASKS

	1st Cal	Cal	Full-Task ¹
PDQ-1	CPU	3173	2971
	[46]	5200 ²	5200
	CAT	179	167
	SpeedUp	18x	18x
PDQ-2	CPU	9430	9198
	[46]	3220 ²	3220
	CAT	533	357
	SpeedUp	18x	26x
PDQ-3	CPU	8.7s	8.4s
	[46]	14.58s ²	14.58s
	CAT	291	254
	SpeedUp	30x	33x
PDQ-4	CPU	16.8s	16.1s
	CAT	977	887
	SpeedUp	18x	18x
			7x

¹ The Full-Task does not take the time of transferring data through into consideration.

² The 1st Cal for [46] is set equal to Cal.

³ The Full-Task of [46] is estimated.

TABLE VI. GPU MEMORY COST FOR RUNNING PDQ TASK IN MB

	PDQ-1	PDQ-2	PDQ-3	PDQ-4
GPU Mem Cost	2316	2521	2191	5429

C. Ablation Analysis

Since we have tested the framework we proposed for its efficiency on the operator level and high-precision PDQ application as a whole, we would like to dig deeper into how the key variants would influence the running performance.

TABLE VII. HARDWARE PARAMETERS OF DIFFERENTGPUS

Type	Memory (GB)	Computation Power(TFLOPS)		Architecture
		Single	Half	
1080Ti	11	11.34	11.34	Pascal
3090Ti	24	35.58	71	Ampere
A100	40	19.5	312	Ampere
V100	26	15.7	125	Volta
4090	24	82.58	165.2	Ada Lovelace

Different GPUs

The GPU card one uses would intuitively affect the running time consumed. We have tested all the operators and the PDQ-1 workload on 5 different types of GPU cards, including 4 major hardware architectures shown in Table VII. As shown in Fig 8, the program runs fastest on Nvidia 4090, which reaches the highest single precision computation capacity. We use the $\frac{1}{t_{compute}}$ to represent the power of performing FHE computation, the higher the value, the more capable the card is. We can notice that under most situations, none of the other types of card can outperform 4090 because of

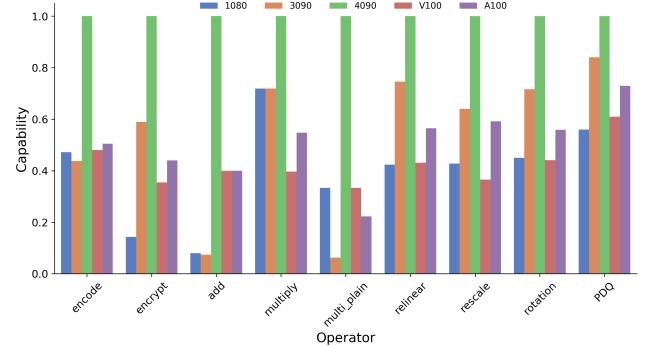


Fig. 8. Computation capability of different type of GPUs

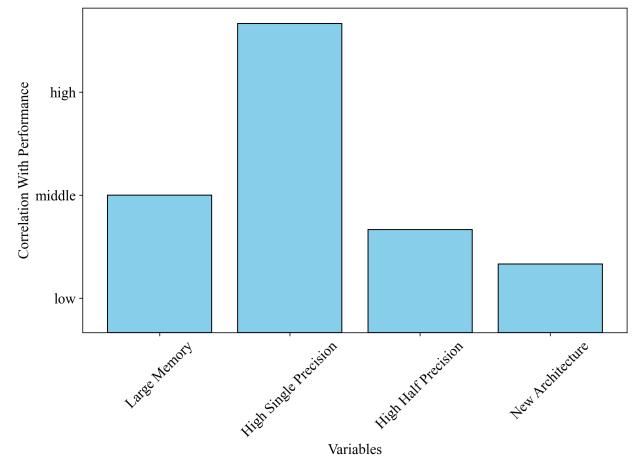


Fig. 9. Correlation between computation capability on FHE with GPUs' hardware Variables

its overwhelming superiority of computation ability on single precision. And it shows a high correlation between single precision computation power and its capacity on FHE. We can draw some other conclusions shown in Fig 9 that the computation power of half-precision and architecture types have relatively less relationship with the computation ability of FHE. While the memory has some impact because of that the threshold of a certain amount of memory is needed for running the whole workload, once the threshold has been met, the excess memory does not bring more advantages for running FHE workloads with our framework.

It is delightful to observe that even a budget card like the 1080Ti can finish all the tests including the PDQ task, and even reach about 60% of the fastest Nvidia 4090 in the PDQ scenario. It shows the great generality of our framework and makes acceleration for industrial FHE applications more reachable just with a card having obsolete architecture and computation power.

Memory pool

Bringing and developing the GPU memory pool is the key to our framework can run the workload costing less memory and maintain the speed of calculating. There are two ways to manage the memory without the pool, by ignoring the returning stage and returning it back to the device every time. The first way will not add any additional time to the running

workload, but the speed of consuming memory is significantly fast, while the second way will not waste any part of memory, but the extra time cost for returning the memory will slow the whole workload down. The numerical result in Fig 10 shows that the first way will consume 3 times more memory to finish the whole computation, and the second way costs 30% more time for the entire task.

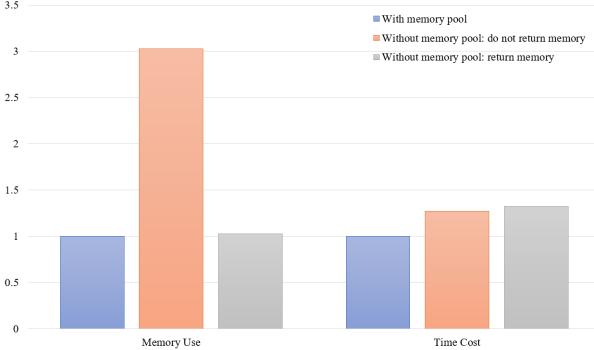


Fig. 10. Memory usage and time cost under different running mode

Stream Pool

We have theoretically shown the advantage of using multiple streams brought by overlapping several independent workloads while consuming more hardware resources at the same time. We test the time latency by running several complicated operators with and without stream pool(using only one stream), additional with different stream numbers. The result in Fig 11 shows the introducing of the stream pool will reduce at most 40% latency to the whole computation, and a pool with more than 3 streams will achieve the best result.

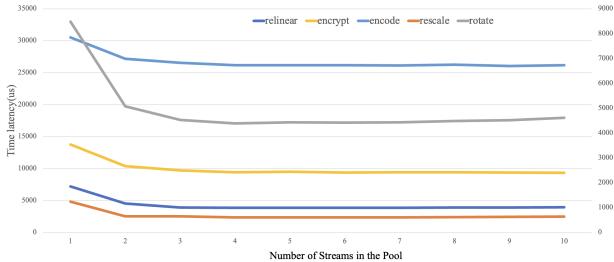


Fig. 11. Time latency of running operators without and with stream pool containing different number of streams

Mod for Big Integers

The implementation of $\%$ for big integers we presented in Algorithm 1 contributes to the workflow. We run a comparison between our implementation and the embedded operator $\%$ in CUDA for big integers. We run 10^8 times $X\%Y$ with two implementations individually, and the numerator and denominator are both set larger than 32-bit and smaller than 64-bit to simulate the situation met in the FHE task. The result in Fig 12 shows that ours outperforms the embedded mod in the CUDA toolkit by 8%.

VI. SUMMARY AND FUTURE PLANS

We present *CAT*(Cipher-Acceleration-Textile), a GPU accelerated FHE framework that achieves significant perfor-

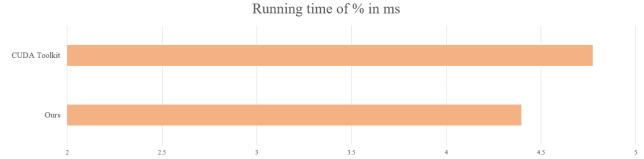


Fig. 12. Running time in ms for $\%$ with our implementation and CUDA toolkit's

mance gains over existing solutions. Its three-layer architecture, parallel execution strategies, and innovative resource management enable a $2173\times$ speedup on single operators and considerable acceleration of CKKS, BFV, and BGV schemes.

We also demonstrate its effectiveness in FHE-based Privacy Database Queries, executing complex SQL queries and computations for 10^3 rows within one second while maintaining memory consumption below 6GB. Extensive testing has been conducted, ensuring its reliability and commercial deployability.

In the future, we plan to optimize bootstrapping procedures, incorporate GPU-accelerated scheme switching mechanisms, and extend the framework's capabilities by integrating additional data query operators. These efforts will further enhance the effectiveness of GPU-accelerated FHE and pave the way for its wider adoption in real-world systems.

ACKNOWLEDGEMENT

Projects and people supporting this work will be added later in compliance with the double blind policy.

REFERENCES

- [1] Aikata, Ahmet Can Mert, Sunmin Kwon, Maxim Deryabin, and Sujoy Sinha Roy. REED: chiplet-based scalable hardware accelerator for fully homomorphic encryption. *CoRR*, abs/2308.02885, 2023.
- [2] Ahmad Al Badawi, Louie Hoang, Chan Fook Mun, Kim Laine, and Khin Mi Mi Aung. Privft: Private and fast text classification with homomorphic encryption. *IEEE Access*, 8:226544–226556, 2020.
- [3] Ahmad Al Badawi, Yuriy Polyakov, Khin Mi Mi Aung, Bharadwaj Veeravalli, and Kurt Rohloff. Implementation and performance evaluation of rns variants of the bfv homomorphic encryption scheme. *IEEE Transactions on Emerging Topics in Computing*, 9(2):941–956, 2019.
- [4] Ahmad Al Badawi, Bharadwaj Veeravalli, Chan Fook Mun, and Khin Mi Mi Aung. High-performance fv somewhat homomorphic encryption on gpus: An implementation using cuda. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 70–95, 2018.
- [5] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, et al. Homomorphic encryption standard. *Protecting privacy through homomorphic encryption*, pages 31–62, 2021.
- [6] Pedro Geraldo MR Alves, Jheyne N Ortiz, and Diego F Aranha. Faster homomorphic encryption over gpgpus via hierarchical dgt. In *International Conference on Financial Cryptography and Data Security*, pages 520–540. Springer, 2021.
- [7] Ahmad Al Badawi, Yuriy Polyakov, Khin Mi Mi Aung, Bharadwaj Veeravalli, and Kurt Rohloff. Implementation and performance evaluation of RNS variants of the BGV homomorphic encryption scheme. *IEEE Trans. Emerg. Top. Comput.*, 9(2):941–956, 2021.
- [8] Christina Boura, Nicolas Gama, Mariya Georgieva, and Dimitar Jetchev. CHIMERA: combining ring-lwe-based fully homomorphic encryption schemes. *J. Math. Cryptol.*, 14(1):316–338, 2020.
- [9] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Trans. Comput. Theory*, 6(3):13:1–13:36, 2014.
- [10] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. A full RNS variant of approximate homomorphic encryption. In Carlos Cid and Michael J. Jacobson Jr., editors, *Selected Areas in Cryptography - SAC 2018 - 25th International Conference, Calgary, AB, Canada, August 15-17, 2018, Revised Selected Papers*, volume 11349 of *Lecture Notes in Computer Science*, pages 347–368. Springer, 2018.
- [11] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 409–437. Springer, 2017.
- [12] Jung Hee Cheon, Miran Kim, and Myungsun Kim. Search-and-compute on encrypted data. In Michael Brenner, Nicolas Christin, Benjamin Johnson, and Kurt Rohloff, editors, *Financial Cryptography and Data Security - FC 2015 International Workshops, BITCOIN, WAHC, and Wearable, San Juan, Puerto Rico, January 30, 2015, Revised Selected Papers*, volume 8976 of *Lecture Notes in Computer Science*, pages 142–159. Springer, 2015.
- [13] Jung Hee Cheon, Miran Kim, and Myungsun Kim. Optimized search-and-compute circuits and their application to query evaluation on encrypted data. *IEEE Trans. Inf. Forensics Secur.*, 11(1):188–199, 2016.
- [14] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: fast fully homomorphic encryption over the torus. *J. Cryptol.*, 33(1):34–91, 2020.
- [15] Léo Ducas and Daniele Micciancio. FHEW: bootstrapping homomorphic encryption in less than a second. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 617–640. Springer, 2015.
- [16] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch.*, page 144, 2012.
- [17] Shengyu Fan, Zhiwei Wang, Weizhi Xu, Rui Hou, Dan Meng, and Mingze Zhang. Tensorfhe: Achieving practical computation on encrypted data using GPGPU. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2023, Montreal, QC, Canada, February 25 - March 1, 2023*, pages 922–934. IEEE, 2023.
- [18] Robin Geelen, Michiel Van Beirendonck, Hilder V. L. Pereira, Brian Huffman, Tynan McAuley, Ben Selfridge, Daniel Wagner, Georgios D. Dimou, Ingrid Verbauwhede, Frederik Vercauteren, and David W. Archer. BASALISC: programmable hardware accelerator for BGV fully homomorphic encryption. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(4):32–57, 2023.
- [19] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 169–178, 2009.
- [20] Ilia Iliashenko and Vincent Zucca. Faster homomorphic comparison operations for BGV and BFV. *Proc. Priv. Enhancing Technol.*, 2021(3):246–264, 2021.
- [21] Wonkyung Jung, Sangpyo Kim, Jung Ho Ahn, Jung Hee Cheon, and Younho Lee. Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with gpus. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 114–148, 2021.
- [22] Wonkyung Jung, Eojin Lee, Sangpyo Kim, Jongmin Kim, Namhoon Kim, Keewoo Lee, Chohong Min, Jung Hee Cheon, and Jung Ho Ahn. Accelerating fully homomorphic encryption through architecture-centric analysis and optimization. *IEEE Access*, 9:98772–98789, 2021.
- [23] Jongmin Kim, Sangpyo Kim, Jaewan Choi, Jaiyoung Park, Donghwan Kim, and Jung Ho Ahn. SHARP: A short-word hierarchical accelerator for robust and practical fully homomorphic encryption. In Yan Solihin and Mark A. Heinrich, editors, *Proceedings of the 50th Annual International Symposium on Computer Architecture, ISCA 2023, Orlando, FL, USA, June 17-21, 2023*, pages 18:1–18:15. ACM, 2023.
- [24] Jongmin Kim, Gwangho Lee, Sangpyo Kim, Gina Sohn, Minsoo Rhu, John Kim, and Jung Ho Ahn. ARK: fully homomorphic encryption accelerator with runtime data generation and inter-operation key reuse. In *55th IEEE/ACM International Symposium on Microarchitecture, MICRO 2022, Chicago, IL, USA, October 1-5, 2022*, pages 1237–1254. IEEE, 2022.
- [25] Myungsun Kim, Hyung Tae Lee, San Ling, Shu Qin Ren, Benjamin Hong Meng Tan, and Huaxiong Wang. Search condition-hiding query evaluation on encrypted databases. *IEEE Access*, 7:161283–161295, 2019.
- [26] Myungsun Kim, Hyung Tae Lee, San Ling, and Huaxiong Wang. On the efficiency of fhe-based private queries. *IEEE Trans. Dependable Secur. Comput.*, 15(2):357–363, 2018.
- [27] Sangpyo Kim, Wonkyung Jung, Jaiyoung Park, and Jung Ho Ahn. Accelerating number theoretic transformations for bootstrappable homomorphic encryption on gpus. In *2020 IEEE International Symposium on Workload Characterization (ISWC)*, pages 264–275. IEEE, 2020.
- [28] Sangpyo Kim, Jongmin Kim, Michael Jaemin Kim, Wonkyung Jung, John Kim, Minsoo Rhu, and Jung Ho Ahn. BTS: an accelerator for bootstrappable fully homomorphic encryption. In Valentina Salapura, Mohamed Zahran, Fred Chong, and Lingjia Tang, editors, *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, pages 711–725. ACM, 2022.
- [29] Sunwoong Kim, Keewoo Lee, Wonhee Cho, Jung Hee Cheon, and Rob A. Rutenbar. Fpga-based accelerators of fully pipelined modular multipliers for homomorphic encryption. In David Andrews, René Cumplido, Claudia Feregrino, and Marco Platzner, editors, *2019 International Conference on ReConfigurable Computing and FPGAs, ReConFig 2019, Cancun, Mexico, December 9-11, 2019*, pages 1–8. IEEE, 2019.
- [30] Sunwoong Kim, Keewoo Lee, Wonhee Cho, Yujin Nam, Jung Hee Cheon, and Rob A Rutenbar. Hardware architecture of a number theoretic transform for a bootstrappable rns-based homomorphic encryption scheme. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 56–64. IEEE, 2020.
- [31] Moon Sung Lee, Yongje Lee, Jung Hee Cheon, and Yunheung Paek. Accelerating bootstrapping in FHEW using gpus. In *26th IEEE*

- International Conference on Application-specific Systems, Architectures and Processors, ASAP 2015, Toronto, ON, Canada, July 27-29, 2015*, pages 128–135. IEEE Computer Society, 2015.
- [32] Younho Lee, Jinyeong Seo, Yujin Name, Jiseok Chae, and Jung Hee Cheon. Heaan-stat: a privacy-preserving statistical analysis toolkit for large-scale numerical, ordinal, and categorical data. *IEEE Transactions on Dependable and Secure Computing*, 2023.
- [33] Wen-jie Lu, Zhicong Huang, Cheng Hong, Yiping Ma, and Hunter Qu. PEGASUS: bridging polynomial and non-polynomial evaluations in homomorphic encryption. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 1057–1073. IEEE, 2021.
- [34] Vincent Migliore, Maria Mendez Real, Vianney Lapotre, Arnaud Tisserand, Caroline Fontaine, and Guy Gogniat. Hardware/software co-design of an accelerator for FV homomorphic encryption scheme using karatsuba algorithm. *IEEE Trans. Computers*, 67(3):335–347, 2018.
- [35] Toufique Morshed, Md Momin Al Aziz, and Noman Mohammed. Cpu and gpu accelerated fully homomorphic encryption. In *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 142–153. IEEE, 2020.
- [36] Harika Narumanchi, Dishant Goyal, Nitesh Emmadi, and Praveen Gauravaram. Performance analysis of sorting of FHE data: Integer-wise comparison vs bit-wise comparison. In Leonard Barolli, Makoto Takizawa, Tomoya Enokido, Hui-Huang Hsu, and Chi-Yi Lin, editors, *31st IEEE International Conference on Advanced Information Networking and Applications, AINA 2017, Taipei, Taiwan, March 27-29, 2017*, pages 902–908. IEEE Computer Society, 2017.
- [37] Xuanle Ren, Zhaojun Chen, Zhen Gu, Yanheng Lu, Ruiguang Zhong, Wen-Jie Lu, Jiansong Zhang, Yichi Zhang, Hanghang Wu, Xiaofu Zheng, Heng Liu, Tingqiang Chu, Cheng Hong, Changzheng Wei, Dimin Niu, and Yuan Xie. CHAM: A customized homomorphic encryption accelerator for fast matrix-vector product. In *60th ACM/IEEE Design Automation Conference, DAC 2023, San Francisco, CA, USA, July 9-13, 2023*, pages 1–6. IEEE, 2023.
- [38] M. Sadegh Riazi, Kim Laine, Blake Pelton, and Wei Dai. HEAX: an architecture for computing on encrypted data. In James R. Larus, Luis Ceze, and Karin Strauss, editors, *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 1295–1309. ACM, 2020.
- [39] Ronald L Rivest, Len Adleman, Michael L Dertouzos, et al. On data banks and privacy homomorphisms. 1978.
- [40] Sujoy Sinha Roy, Furkan Turan, Kimmo Järvinen, Frederik Vercauteren, and Ingrid Verbauwhede. Fpga-based high-performance parallel architecture for homomorphic computing on encrypted data. In *25th IEEE International Symposium on High Performance Computer Architecture, HPCA 2019, Washington, DC, USA, February 16-20, 2019*, pages 387–398. IEEE, 2019.
- [41] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald G. Dreslinski, Christopher Peikert, and Daniel Sánchez. F1: A fast and programmable accelerator for fully homomorphic encryption. In *MICRO '21: 54th Annual IEEE/ACM International Symposium on Microarchitecture, Virtual Event, Greece, October 18-22, 2021*, pages 238–252. ACM, 2021.
- [42] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Nathan Manohar, Nicholas Genise, Srinivas Devadas, Karim Eldefrawy, Chris Peikert, and Daniel Sánchez. Craterlake: a hardware accelerator for efficient unbounded computation on encrypted data. In Valentina Salapura, Mohamed Zahran, Fred Chong, and Lingjia Tang, editors, *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, pages 173–187. ACM, 2022.
- [43] Nigel P. Smart and Frederik Vercauteren. Fully homomorphic SIMD operations. *Des. Codes Cryptogr.*, 71(1):57–81, 2014.
- [44] Benjamin Hong Meng Tan, Hyung Tae Lee, Huaxiong Wang, Shu Qin Ren, and Khin Mi Mi Aung. Efficient private comparison queries over encrypted databases using fully homomorphic encryption with finite fields. *IEEE Trans. Dependable Secur. Comput.*, 18(6):2861–2874, 2021.
- [45] Mark A Will and Ryan KL Ko. Computing mod without mod. *Cryptology ePrint Archive*, 2014.
- [46] Fahong Zhang, Chen Yang, Rui Zong, Xinran Zheng, Jianfei Wang, and Yishuo Meng. An efficient and scalable fhe-based pdq scheme: Utilizing fft to design a low multiplication depth large-integer comparison algorithm. *IEEE Transactions on Information Forensics and Security*, pages 1–1, 2023.