RESEARCH-ARTICLE

BoostCom: Towards Efficient Universal Fully Homomorphic Encryption by Boosting the Word-wise Comparisons

ARDHI WIRATAMA YUDHA, University of Central Florida, Orlando, FL, United States

JIAQI XUE, University of Central Florida, Orlando, FL, United States

QIAN LOU, University of Central Florida, Orlando, FL, United States

HUIYANG ZHOU, NC State University, Raleigh, NC, United States

YAN SOLIHIN, University of Central Florida, Orlando, FL, United States

Open Access Support provided by:

University of Central Florida

NC State University

PDF Download
3656019.3676893.pdf
27 December 2025
Total Citations: 1
Total Downloads: 365

Published: 14 October 2024

Citation in BibTeX format

PACT '24: International Conference on Parallel Architectures and Compilation Techniques
October 14 - 16, 2024
CA, Long Beach, USA

Conference Sponsors:
SIGARCH

PACT '24: Proceedings of the 2024 International Conference on Parallel Architectures and Compilation Techniques (October 2024)
https://doi.org/10.1145/3656019.3676893
ISBN: 9798400706318

BoostCom: Towards Efficient Universal Fully Homomorphic Encryption by Boosting the Word-wise Comparisons

Ardhi Wiratama Baskara Yudha
University of Central Florida
USA
Advanced Micro Devices, Inc.
USA
ar755415@ucf.edu

Jiaqi Xue
University of Central Florida
United States of America
jiaqi.xue@ucf.edu

Qian Lou
University of Central Florida
USA
qian.lou@ucf.edu

Huiyang Zhou
North Carolina State University
United States of America
hzhou@ncsu.edu

Yan Solihin
University of Central Florida
United States of America
yan.solihin@ucf.edu

Abstract

Fully Homomorphic Encryption (FHE) allows for the execution of computations on encrypted data without the need to decrypt it first, offering significant potential for privacy-preserving computational operations. Emerging arithmetic-based FHE schemes (ar-FHE), like BGV, demonstrate even better performance in word-wise comparison operations over non-arithmetic FHE (na-FHE) schemes, such as TFHE, especially for basic tasks like comparing values, finding maximums, and minimums. This shows the universality of ar-FHE in effectively handling both arithmetic and non-arithmetic operations without the expensive conversion between arithmetic and non-arithmetic FHEs. We refer to universal arithmetic Fully Homomorphic Encryption as uFHE. The arithmetic operations in uFHE remain consistent with those in the original arithmetic FHE, which have seen significant acceleration. However, its non-arithmetic comparison operations differ, are slow, and have not been as thoroughly studied or accelerated. In this paper, we introduce BoostCom, a scheme designed to speed up word-wise comparison operations, enhancing the efficiency of uFHE systems. BoostCom involves a multi-prong optimizations including infrastructure acceleration (Multi-level heterogeneous parallelization and GPU-related improvements), and algorithm-aware optimizations (slot compaction, non-blocking comparison semantic). Together, BoostCom achieves an end-to-end performance improvement of more than an order of magnitude (11.1 × faster) compared to the state-of-the-art CPU-based uFHE systems, across various FHE parameters and tasks.

CCS Concepts

• **Computer systems organization → Multiple instruction, single data**.

PACT '24, October 14–16, 2024, Long Beach, CA, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0631-8/24/10
https://doi.org/10.1145/3656019.3676893

Keywords

Fully Homomorphic Encryption, Security, GPU, Confidential Computing

ACM Reference Format:
Ardhi Wiratama Baskara Yudha, Jiaqi Xue, Qian Lou, Huiyang Zhou, and Yan Solihin. 2024. BoostCom: Towards Efficient Universal Fully Homomorphic Encryption by Boosting the Word-wise Comparisons. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '24), October 14–16, 2024, Long Beach, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3656019.3676893

1 Introduction

There has been a surge of interest from the industry in Fully Homomorphic Encryption (FHE) [14] acceleration recently [2, 5, 34] as FHE may play a pivotal role in facilitating computation on private data in the cloud without disclosing its plaintext. FHE has been cited to be applicable for many types of computation, including machine learning, and big data analytics, on various application domains that include healthcare, finance, genomics research, secure voting systems, and private information retrieval, where it helps maintain stringent privacy regulations [8, 12, 19, 26, 46]. Figure 1 illustrates the FHE workflow, which includes client-side encoding, encryption, server-side computation, and subsequent client-end decryption and decoding, which assures client-side data confidentiality even on potentially untrusted servers, unlike TEEs that require the user to trust the server [41, 42, 44].

Various FHE schemes have emerged over the past decade, including *arithmetic FHE* (ar-FHE), such as word-wise BGV [7] and CKKS [9], and *non-arithmetic FHE* (na-FHE), such as bit-wise TFHE [11]. Originally, ar-FHEs were adept at performing arithmetic operations like multiplications and additions, while na-FHEs were primarily used for bit-wise comparison operations. Although na-FHEs excel in bitwise comparisons, they show slower performance in conducting arithmetic operations on integers. In contrast, *the ar-FHE scheme BGV [18] has been upgraded recently with new word-wise comparisons*, such that it not only efficiently handles integer arithmetic operations but also supports batched word-wise comparisons, outperforming na-FHEs in speed. This advancement positions the BGV scheme as a solution for *both* arithmetic and non-arithmetic

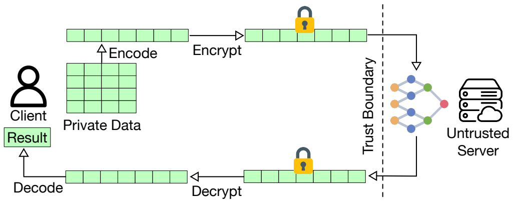

Figure 1: Performing computations on encrypted data transferred to an untrusted server using FHE.

comparisons, a combined capability we henceforth define as *universal FHE* (uFHE). In contrast, CKKS-based polynomial approximation for non-linear operations still suffers from a precision reduction since each operation affects the fractional value of the ciphertext [20]. Furthermore, comparisons within the CKKS framework, when involving approximated polynomials, lead to non-negligible errors [10][25].

Nevertheless, the uFHE scheme based on the new BGV [18] is not without its limitations, particularly the sluggish and complex comparison operation. A comparison operation compares pairs of encrypted data to generate an encrypted result that indicates whether they are equivalent, less than, or greater than. To execute a single comparison, it requires $3p - 5$ non-scalar multiplications along with additions, rotations, and scalar multiplications, with p denoting the plaintext modulus [39], with typical values reaching up to the tens. Despite the costs, a variety of applications, including scientific computations and machine learning, depend heavily on this comparison operation.

Recognizing that a comparison operation may create a performance bottleneck in new BGV-based uFHE, there has been an effort to rely on an algorithmic approach to accelerate it [18]. The algorithmic approach reduces the comparison complexity to $2p - 6$ (Bivariate case) and $\sqrt{p-3}+O(\log p)$ (Univariate case) [1]. Although an algorithmic approach is valuable, we are of the view that it alone may not be adequate to meet the requirements of high performance. Proposals have been made to switch between FHE schemes like TFHE-BGV [6] and TFHE-CKKS [27]. However, these transitions are still costly, with over 70× the latency of BGV [18].

Therefore, in this paper, we propose an infrastructure acceleration approach (called *BoostCom*), where we offload comparison to the Graphics Processing Unit (GPU) and apply various optimizations. We note that infrastructure acceleration approach has been pursued successfully for various other operations such as encryption, decryption, multiplication, bootstrapping, and other power-of-two polynomial ring operations [13, 21, 28, 35, 40], including on FPGA [31, 38], and ASIC [24, 32, 33]. However, they have all neglected the comparison operation, which is the focus of this paper.

However, a single comparison may be up to multiple orders of magnitude slower than multiplication. Hence, accelerating comparison is challenging, requiring us to use several strategies including heterogeneous CPU/GPU parallelization, slot compaction, non-blocking comparison semantics, branch removal, and layout optimization, as detailed below.

First, we introduce a strategy for heterogeneous parallelization, wherein multicore CPUs manage parallelization at a higher, digit-level, while GPUs handle the parallelization of fundamental FHE operations at a more granular, polynomial level. This design approach is inspired by the crucial insight that, although the bulk of BGV comparison operations can be parallelized, the parallelism granularity of certain inner operations is insufficient to outweigh the overheads associated with memory copying, memory allocation, and kernel launching. Consequently, transferring these tasks to GPUs might not result in a net gain in performance. By adopting a heterogeneous, multi-level parallelism strategy, we enable CPUs and GPUs to collaborate effectively, thereby enhancing the efficiency of BGV comparison operations.

Second, we propose multiple GPU-related optimizations for primitive polynomial operations in word-wise comparisons. Profiling detailed in Section 4 reveals that the execution time of word-wise comparison operations is mainly spent on three components: BluesteinNTT, BluesteinFFT, and Element-wise operations. These components are highly parallelizable, suggesting potential efficiency gains by offloading them to GPU. The optimizations include branch removal to increase the parallelism of BluesteinNTT, plan reuse to reduce the computational loads for BluesteinFFT, and memory layout transformation for efficient element-wise operations.

Third, we introduce two algorithmic enhancements in addition to heterogeneous parallelism and GPU optimizations. (I) Slot compaction. uFHE's comparison mechanism leverages SIMD ciphertext batching, allowing the comparison of two vectors through a single ciphertext comparison, given that each vector of size a can be encoded into one ciphertext. This is possible when the number of slots (b) in the ciphertext exceeds a, highlighting the significance of slot utilization ($\frac{a}{b}$) for comparison efficiency. A critical observation is that in prevalent workloads (e.g., machine learning), comparisons typically follow arithmetic operations, leading to low slot utilization in ciphertexts awaiting comparison. This scenario presents an opportunity for slot compaction, enhancing efficiency by increasing slot utilization before comparison. To capitalize on this, we introduce a slot manager designed to track and optimize slot utilization within a ciphertext. This strategy facilitates slot compaction, thereby reducing memory consumption and boosting performance.(ii) non-blocking semantic. We propose *non-blocking semantic* for comparison that allows the overlap of comparison with other computations. The semantic allows comparison to be executed on another CPU thread while the main CPU thread continues executing the next code segment concurrently until the main thread needs to use the result of the comparison. To increase the distance until the use of the result, we perform code straightlining.

We implemented the optimizations on a real-world library (HElib) which enables us to evaluate end-to-end performance (instead of operation-wise evaluation in many prior studies) reliably. Our optimizations do not negatively affect the noise budget, as no additional homomorphic operations were added; they merely enhance GPU efficiency. We evaluate several applications including sorting, finding minimum elements, multi-layer perceptron (MLP), image re-colorizing, and a private query. Our evaluation shows that the proposed acceleration is effective in boosting the performance of

[1] Bivariate and Univariate are two different algorithms used to perform comparison operations.

the comparison and the application that uses it. Across the five benchmarks, it achieves *end-to-end* geometric mean speedup of 11.1× (up to 26.7×), over an industry-standard FHE library running on 16-core CPUs. BoostCom significantly outperforms HE-Booster [40], a state-of-the-art GPU accelerator for BGV scheme that is also implemented in HElib, by 553%.

To summarize, this paper makes the following contributions:

(1) We proposed a multi-level heterogeneous parallelism method as an infrastructure acceleration for comparison in the uFHE scheme.
(2) We proposed multi-prong GPU-related optimizations for accelerating uFHE comparison, including branch-removal, as well as plan reuse and layout optimization. These optimizations are incorporated into a new library called cuHELIB, which builds upon HElib by leveraging GPU technology.
(3) We present new uFHE comparison algorithms featuring slot compaction for ciphertext comparison to lower memory usage and non-blocking comparison to reduce computational dependencies, thereby increasing throughput.
(4) We conducted a comprehensive evaluation of our scheme, considering end-to-end measurements that include CPU-GPU memory copy, kernel launches, and synchronization on five important applications. This approach provides a more holistic assessment compared to extrapolating from operation-wise measurements.

The remainder of the paper is organized as follows. Section 2 discusses the related work, Section 3 presents the background, Section 4 analyzes the performance bottlenecks of comparison operations in BGV, Section 5 discusses the design of BoostCom and our proposed optimizations, Section 6 presents our experimental methodology, Section 7 discusses our results, and Section 8 concludes.

2 Background

2.1 Word-wise Universal FHE Scheme

Originally, ar-FHEs (arithmetic FHEs) including BGV were adept at performing arithmetic operations like multiplications and additions. Recently, the ar-FHE scheme BGV [18] has been upgraded with new word-wise comparisons (arithmetic operations are still the same with prior BGV), such that it not only efficiently handles integer arithmetic operations but also supports batched word-wise comparisons, outperforming na-FHEs in speed. This advancement positions the BGV scheme as a universal FHE solution (uFHE) for both arithmetic and non-arithmetic comparisons.

Other uFHE methods have been made to switch between FHE schemes like TFHE-BGV [6] and TFHE-CKKS [27]. However, these transitions are still costly, with over 70x latency compared to BGV [18]. Thus, the uFHE based on new upgraded BGV [18] is the-start-of-the-art. However, the current computational bottleneck of uFHEs, particularly the BGV comparison operation, is limited to running on a single CPU.

Basics and arithmetic ops. of uFHE-based BGV. The uFHE-based BGV scheme is a lattice-based encryption based on Ring Learning with Errors (RLWE) problem [7]. RLWE is a challenging mathematical problem in lattice-based encryption that creates a foundation for developing safe encryption schemes. Table 1 shows the essential BGV parameters. Key parameters include p, m, and N.

Table 1: Parameters used in BGV and comparison operation.

Parameter	Description
p	Plaintext coefficient modulus.
m	The order of the cyclotomic ring.
N	The degree of the cyclotomic polynomial.
Q	The product of (prime) moduli: $Q = \prod_{i=0}^{L} q_i$.
L	Maximum (multiplicative) level.
λ	Security level of a given BGV instance.
ω	Root of unity of twiddle factor for NTT.
d	The dimension of a vector space over a finite field.
l	The length of vectors to be compared.

p defines the plaintext modulus; a higher p enlarges the plaintext space but slows down comparisons. The roles of m and N will be outlined later.

In the BGV scheme, a plaintext is encoded into a polynomial and encrypted to form a ciphertext polynomial. Computation can be performed on the ciphertext, yielding a result also in ciphertext form, which requires decryption to obtain the plaintext. Both plaintext and ciphertext polynomials reside in the same ring with different coefficient moduli, where the ciphertext modulus is significantly larger than the plaintext modulus. The ciphertext polynomial ring (R_Q) in the BGV scheme is $C = R_Q \times R_Q$, where $R_Q = \mathbb{Z}_Q[x]/(\Phi_m(x))$, and $\Phi_m(x)$ is the m^{th} cyclotomic polynomial with a degree of N. The relationship between m and N is determined by the Euler totient function φ, i.e., $N = \varphi(m)$. While prior works use a power-of-two N for simplicity, non-power-of-two N is suggested for better performance and higher security flexibility [15, 18]. $Q \in \mathbb{Z}$ is the ciphertext coefficient modulus at level L, representing the product of several primes $(q_0, q_1, q_2, ..., q_L)$ that fit into the native integer data type. The value of Q determines the multiplicative depth, i.e., the most extended sequence of homomorphic multiplications during computation. Q is typically much larger than p, influencing the message expansion rate after encryption. The individual primes q_i are part of the modulus chain.

The BGV scheme utilizes SIMD-style processing, storing multiple integers in one ciphertext to optimize operation throughput. Leveraging ring isomorphism of polynomial modulus enables multiple plaintext slots within a ciphertext. Modular arithmetic is used for homomorphic operations including addition, multiplication, and rotation. However, noise introduced during encryption limits operation numbers and requires a large ciphertext modulus (Q).

Non-arithmetic Comparison of uFHE-based BGV. The state-of-the-art comparison algorithm was proposed in [18]. It exploits SIMD-style processing such that many comparisons can be performed in parallel, leading to a small amortized comparison latency. A large integer comparison operand is encoded into an element of $\mathbb{F}^l_{p^d}$. Here, $\mathbb{F}^l_{p^d}$ represents a finite field extension of degree d over a prime field with p elements. The encoding process involves decomposing a large integer into an element in this vector space, where the vector space is of dimension l.

To compare two integers a and b, first, each integer is *decomposed* into multiple slots in the form of $\mathbb{F}_{p^d}$. For example, a is decomposed into $a_0, a_1, \ldots, a_{l-1}$, where a_i occupying the i-th slot. For each slot, using the *mod extract* step, each number is further split into multiple digits in the form of $\mathbb{F}_p$. For example, a_0 is split into a_{00} (the first digit in the first slot), a_{01} (the second digit in the first slot), etc.

To perform comparison, the algorithm first extracts digits of encrypted numbers in $\mathbb{F}_p$, then performs equality (EQ) and less than (LT) functions for each digit using specific equations. The computation of LT and EQ for each digit is independent. The results of the equality and less than functions on the digits are combined through lexicographical order. First, the lexicographical order is computed for each block of d digits, and then the results are combined using a final equation that returns encrypted “1" when $a < b$ or encrypted “0" otherwise. The last two steps that involve ciphertext shifting and multiplying with the result from the equality circuit are called *ShiftMul*, whereas the step for performing a summation of the ciphertext is called *ShiftAdd*. The digit comparison steps are expensive due to repeated ciphertext exponentiations with large exponents for $d \times l$ times, while other steps (*Extraction*, *ShiftMul*, and *ShiftAdd*) are faster. The process represents a Bivariate circuit with separated LT and EQ computations, whereas the Univariate circuit combines LT and EQ circuits differently.

2.2 Efficient Polynomial and NTT

To handle the large ciphertext modulus Q, the BGV scheme uses a Residue Number System (RNS) format, splitting the polynomial into $L + 1$ residue polynomials with coefficients under modulo q_i, where q_i's are pair-wise coprime integers. RNS allows for efficient multiplication and addition of ciphertext polynomials using current hardware systems.

To accelerate polynomial multiplication, the Number Theoretic Transform (NTT) is used, converting the polynomial to an integer Discrete Fourier Transform (DFT) representation using a twiddle factor ω that meets specific conditions. For efficient NTT and INTT, radix-2 NTT implementations are applied when N is a power of two, employing Cooley-Tukey (CT) and Gentleman-Sande (GS) algorithms. The ciphertext polynomial is represented as a matrix of polynomial coefficients in integer DFT representation of size $(L + 1) \times \varphi(m)$, enabling straightforward element-wise operations for multiplication and addition between polynomials.

The BluesteinNTT algorithm is used for polynomial conversion between coefficient representation and integer DFT representation when N is a non-power of two. The algorithm requires two twiddle factors: TF1, the twiddle factors for polynomial ring m, and TF2, the twiddle factors for a power of two polynomial ring. First, the input polynomial is multiplied element-wise by TF1 to generate a polynomial C. Then, the polynomial C is padded with zero to become C_pad and then multiplied by polynomial D_pad. D_pad is a polynomial generated from TF1. Both polynomials C_pad and D_pad have length power of two greater than $2m - 1$. The polynomial multiplication between them is accelerated by the radix-2 NTT algorithm (CT and GS) that requires TF2. The multiplication result (C_pad **x** D_pad) is then truncated to have length m, with the exceeding coefficient being added to the polynomial. The resulting polynomial is then multiplied element-wise by TF1. Finally, the polynomial is filtered to have a length from m to N.

3 Related Works

Infrastructure acceleration is an approach to accelerate FHE operations with the use of hardware accelerators and efficient software implementation. It is used along with algorithmic improvement to achieve desirable performance. Table 2 shows the comparison of the prior works with BoostCom on infrastructure acceleration of FHE. Among all the works on infrastructure accelerations, only ours focuses on boosting the latency of comparison operations on the BGV scheme. Furthermore, the infrastructure acceleration from the prior works could be divided into two categories: operation-wise acceleration and end-to-end acceleration. For the former, the acceleration is only targeting reducing the latency of each primitive FHE operation separately such as multiplication, addition, rotation, etc. Therefore they only estimate the total execution time of an application that runs on their proposal by the latency of each FHE operation. The proposals belonging to this category typically ignore the problem of dynamic memory allocation, different levels of the ciphertext operand, noise estimation, etc. since these problems may not arise when only accelerating each of the operations separately. For the latter, the acceleration takes into account these problems and is typically used to accelerate real-world libraries such as HElib [17] and Microsoft SEAL [34]. The end-to-end acceleration has a more immediate impact than operation-wise acceleration. It can be used to improve the execution time of the application that uses the real-world HE library immediately. In contrast, the operation-wise acceleration needs more work to gather them to be usable to truly run an application on it. Moreover, for both categories, the acceleration is divided by the type of hardware platform such as ASIC, FPGA, CPU (with new instructions), GPU, and mixed CPU/GPU.

Table 2: The comparison of BoostCom vs. prior works.

Name	Scheme	Comparison	End-to-End	Platform
SHARP [22]	CKKS	✓	✗	ASIC
CraterLake [33]	CKKS	✗	✗	ASIC
FxHENN [47]	CKKS	✗	✗	FPGA
TensorFHE [13]	CKKS	✗	✗	GPU
HE on GPU [28]	BFV	✗	✓	GPU
Intel HEXL [5]	BGV	✗	✓	CPU
HE-Booster [40]	BGV	✗	✗	GPU
BoostCom	**BGV**	✓	✓	**CPU/GPU**

Algorithmic acceleration. To boost the comparison operation on BGV/BFV, some algorithmic improvements have been proposed [18, 39]. The scheme results in a slightly faster speed for performing comparison compared to TFHE when the number of messages being compared is large. However, when only comparing a single message in a ciphertext, the comparison latency becomes very expensive. Typically this problem arises when the comparison is used to determine the taken branch path. Our works proposed an optimization to mitigate this problem called *non-blocking comparison*. The optimization overlaps the comparison operation with other works that do not depend on the comparison result.

Operation-wise acceleration with GPUs. The works in [13, 36, 40] propose a GPGPU-based FHE acceleration solution called TensorFHE, GME, and HE-Booster, respectively. TensorFHE utilizes algorithm optimization, Number Theoretic Transform (NTT) optimization, and data layout optimization to achieve significant performance improvement for FHE arithmetic operations. It also utilizes tensor cores to speed up the NTT operation. GME introduces a new NoC that connects all scratchpad memory in the GPU to reduce access to the main memory during NTT operation. HE-Booster improves the FHE arithmetic operation by improving the GPU NTT implementation from [29] with fine-grain synchronization on every iteration of NTT computation.

Operation-wise acceleration with ASIC/FPGA. Several works in this category include [22–24, 32, 33, 47]. These proposals introduced an NTT unit for processing radix-2 NTT. CraterLake [33] is the first FHE accelerator to achieve high performance on unbounded FHE programs while prior accelerators are only efficient on a limited subset of simple FHE computations [32]. CraterLake [33] is a uniprocessor with specialized functional units that span a wide vector space. The design is statically scheduled in order to take advantage of the regularity of FHE computations. SHARP [22], reduces the computation latency of the FHE operation by limiting the size of the prime modulus to only 36-bit. This will translate into lower memory bandwidth demand for the accelerator's memory thus improving the performance. Although the infrastructure acceleration using ASIC/FPGA may offer higher acceleration and better power efficiency than GPUs, they are constrained by higher development time, lower generality and lower flexibility. In contrast, CPUs/GPUs are widely available, allowing quick deployment. Their flexibility allow changing schemes, algorithms, and implementations easily.

End-to-end acceleration. Intel HEXL introduced a new CPU instruction for processing 512-bit vectors, speeding up element-wise operations and NTT. However, this only benefits power-of-two polynomial rings, and comparison operations remain slow in such rings. The work in [28] proposed the acceleration of the BFV scheme in the Microsoft SEAL library for power-of-two polynomial rings, element-wise operation, and key-switching. Compared to our work, this work provides general FHE acceleration, while our proposal focuses on accelerating comparison operation on the BGV scheme.

4 Bottleneck of uFHE Comparison

The state-of-the-art BGV comparison implementation is in HElib [18]. It was reported that it was up to 3× faster than prior work based on BGV/BFV, and achieved even better performance than bit-wise FHE schemes in basic comparison tasks such as less-than, maximum, and minimum operations. However, each comparison still takes up to several seconds, hence we argue for the need for infrastructure acceleration.

To accelerate BGV comparison in HElib, we first identify the bottlenecks in the library component. To achieve this, we perform profiling and measure the execution time breakdown based on the components in the library. Figure 2 shows the execution time breakdown of the comparison operation based on the primitive HElib components. For brevity, the figure only shows the profiling results from the Univariate case, but we note that the Bivariate case exhibits similar results. The platform we used for the profiling is detailed in Table 3.

The figure shows that the execution time is mainly spent on three components: BluesteinNTT, BluesteinFFT, and Element-wise operations. Upon code inspection, we found that they are also highly parallelizable, so offloading them to GPU could be fruitful. In contrast, the "Small loops" component is also quite significant. It consists of many small loops scattered inside the library. While the code is parallelizable, the degree of the parallelism is too small to compensate for the overheads of memory copy, memory allocation, and kernel launch. Additionally, repeated memory copying can be costly [45]. Therefore, offloading these codes to GPUs may not yield net performance improvement. For the Chinese Remainder Theorem (CRT), although the code involves multiple loops, the most time-consuming loops in this component involve the computation of a big integer and storing the final result in it. Currently, there is no support for big integer data types on GPUs, whereas a highly optimized library for CPUs exists[16]. Therefore, both CRT and Small loop components may not benefit from GPU offloading; instead, we will utilize CPU for their parallelization. Note that CRT parallel execution on multiple CPU cores is already the case in HElib, and we keep it that way. Furthermore, we add parallel execution of "Small loops" components on CPUs.

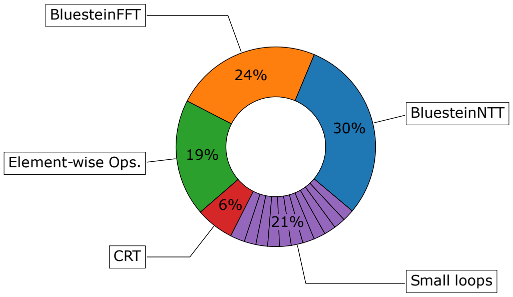

Figure 2: Breakdown of BGV comparison time for Bivariate circuit with parameters $m = 34511$, $p = 3$, and $d = 6$.

5 The Design of BoostCom

This section describes BoostCom, our solution for BGV comparison operation acceleration through the use of GPU and multiple CPU threads. After we conduct the execution time breakdown from the previous section, we discover some primitive components inside the HElib that need to be offloaded to the GPU and what steps in the algorithm to look out for the possibility of acceleration with multi CPU threads.

5.1 Multi-Level Heterogeneous Parallelization

Profiling results (Section 4) identified BluesteinNTT, BluesteinFFT, and element-wise operations in BivarLT/BivarEQ/UnivarLT+EQ as taking roughly three quarters of the execution time. Thus, an obvious acceleration step is to offload them to the GPU to benefit from the massive parallelism on the GPU. However, after offloading, through profiling we found that the GPU utilization is less than 10%. This is because the parallelism granularity of the operations is insufficient to outweigh the overheads associated with memory copying, memory allocation, and kernel launching. Meanwhile, the CPU is mostly idle waiting for GPU computation results. To address both problems, we propose heterogeneous parallelization where higher-level parallelization is performed at the CPU.

To perform parallelization on the CPU, one approach is to only parallelize the most time-consuming operations (i.e., ciphertext exponentiation). However, this approach is challenging as the use of recursion creates loop-carried dependences. Moreover, the exponent of the parameters depends on p, which may exceed the number of CPU threads, making load balancing challenging. The load imbalance between threads can hinder performance [43]. Hence, we explore an alternative approach of parallelizing across digits. As discussed earlier in 2.1, the computation of each digit in LT and EQ has no dependence on the computation of other digits. LT_{ij} computes LT with digit input a_{ij} and b_{ij} only. The computation of each digit is also highly parallel. Hence we adopt a heterogeneous parallelism strategy, where we use GPU for specific computations for each digit in parallel, and utilize multicore CPUs to exploit digit-level parallelism. This is illustrated in Figure 3.

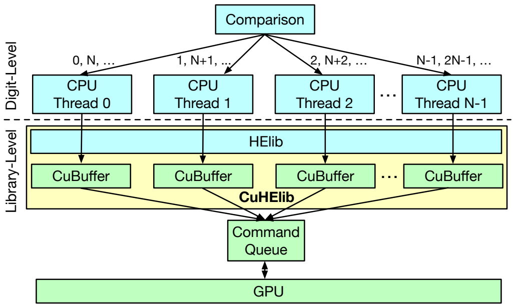

Figure 3: Illustrating Boostcom's heterogeneous parallelism: digits are computed across multiple CPU threads, while primitive operations in each digit are offloaded to the GPU.

The parallelization for digit computation is wrapped inside a library which we name cuHElib, built on top of HElib. We added multiple buffers (called CuBuffers) to hold data in the GPU memory, a command queue to dispatch tasks to the GPU, and changed the GPU task offloading strategy. The library offloads each expensive operation or function as a task (BluesteinNTT, BluesteinFFT, and Element-wise operations) to the GPUs. At the digit level, the computation of LT and EQ of different digits are computed across multiple CPU threads simultaneously. To avoid races and synchronization, we allocate a separate GPU buffer for each CPU thread.

Since the CPU and GPU have separate memories, offloading computation tasks to the GPU requires copying data to a GPU buffer, launching a kernel to compute the task, and then copying the result back to the CPU. If Unified Memory (UM) [30] is supported, the copying may be performed implicitly as the CPU and GPU share virtual memory address space. However, to avoid page thrashing and page faults while a kernel is running, we use explicit copying with careful timing.

5.2 GPU-related Optimizations

Branch Removal for Faster BluesteinNTT. The BluesteinNTT computation involves element-wise multiplication (2 times), radix-2 NTT/INTT conversion, element-wise addition, and polynomial filtering. To accelerate it, we adopted the state-of-the-art radix-2 NTT/INTT implementation [29], applied an optimization [40], and used the Barret reduction for modular operations [37]. We discovered that the remaining performance bottleneck is in polynomial filtering, which is not parallelizable due to loop-carried dependency.

Polynomial filtering alters the polynomial length from m to N. In Listing 1, the update of the variable j is control-dependent on the loop iterator i, creating a loop-carried dependence that hinders loop-level parallelization. If executed sequentially with a single GPU thread, it would be inefficient due to the comparatively slower speed of a single GPU thread compared to a CPU thread[4]. Instead, we propose a *branch removal* optimization by breaking down the code into two phases (Listing 2): the *offline phase* and the *online phase*. The offline phase removes loop-carried dependences by precomputing indices to set the target index for *final_result*. This is achieved by computing the prefix-sum of the value array of *ZmStar*. Additionally, since all the inputs for index pre-computation are available before FHE computation, we can pre-compute it on the CPU. As a result, the online phase, when it performs selective copy, becomes parallelizable as we remove the branch and can benefit from GPU execution. This transformation also leverages efficient GPU pipeline computation and enables the use of multi-streaming to further improve GPU utilization.

```
for (i = 0, j = 0; i < m; i++)
  if (zMStar->inZmStar(i))
    final_result[j++] = coeff(result, i);
```

Listing 1: BluesteinNTT polynomial filtering code showing loop-carried dependence due the if statement and j++.

```
//offline phase: index pre-comp. to remove loop-carried dependence
prefixSum(sumZmStar, inZmStar, getM);
//online phase:selective copy executed in parallel with GPU
__global__ filterBluestein(tmp, inZmStar, sumZmStar, m){
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < m && inZmStar[i] != 0)
      final_result[sumZmStar[i]] = result[i]; }
```

Listing 2: *Branch removal* optimization that removes loop-carried dependence in polynomial filtering.

Plan Reuse for BluesteinFFT Acceleration. BluesteinFFT significantly contributes to the comparison operation latency in HElib. To ensure the correctness of the ciphertext, HElib checks the noise level after each operation using BluesteinFFT. While one could use a very large Q value to prevent noise budget exceedance, this may reduce computation efficiency. Opting for smaller Q values, though requiring noise estimation using BluesteinFFT, may enhance computation efficiency.

HElib utilizes the CPU library PGFFT, which we replace with the cuFFT library for GPU offloading. Before using BluesteinFFT with cuFFT, a configuration step is necessary, involving plan creation for optimal thread organization. Two distinct strategies are under consideration to optimize the utilization of cuFFT: the first involves the creation of the execution plan before every BluesteinFFT operation, a straightforward yet computationally expensive approach; the second strategy, denoted as *plan reuse*, configures the plan once at the initiation of FHE computation. Subsequently, during the execution of BluesteinFFT, pointers for twiddle factors and the GPU execution plan are conveyed, effectively eliminating the need for plan creation on the critical path of the operation.

Layout Transformation for Efficient Element-Wise Ops. Element wise operation in HElib multiplies two matrices of size $(L+1) \times \varphi(m)$ by iteratively multiplying and adding. Each matrix is dynamically allocated because a homomorphic operation may add and/or delete rows during execution. The dynamic allocation may result in non-contiguous memory addresses, which creates a problem for *cudamemcpy* which only copies contiguous memory address range. Thus, copying an entire matrix to the GPU using *cudamemcpy* may lead to copying unrelated data. Moreover, copying the matrix result back is not feasible, as it may overwrite unrelated

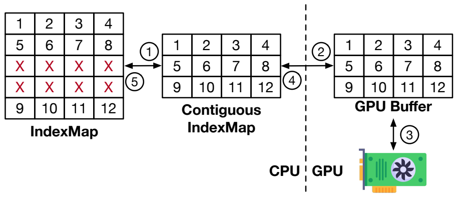

Figure 4: Layout optimization for offloading the element-wise operation to the GPU, utilizing additional copying at the CPU side to maximize the CPU-GPU *memcpy* bandwidth and parallelization degree.

data processed by other threads. We explore several options to address the issue.

One possible approach is to perform the element-wise operation row-by-row, which is an approach implemented for CPUs in Intel HEXL library [5]. If we use this approach for GPU, we may suffer from high memory copying latencies for each row of the matrices and from a low degree of parallelism on the GPU, which may result in underutilized GPU. An alternative approach involves copying the entire matrix to the GPU row-by-row and then executing the element-wise operation for the entire matrix. This generally reduces the total kernel time. However, it still results in PCIe bandwidth wastage since only a small amount of data is copied to the GPU multiple times.

Thus, we use a third approach, which we refer to as *layout transformation*, to create a contiguous memory allocation in the CPU buffer, which allows the element-wise operation for the entire matrix to be offloaded in a single GPU kernel. Figure 4 illustrates the steps for this optimization: ① data from the original matrix with non-contiguous row locations is copied over to a new buffer with contiguous mapping. ② the entire matrix with contiguous rows is transferred to the GPU. ③ element-wise operation is performed on the GPU, producing results in the GPU buffer. ④ data in the GPU buffer is copied back. ⑤ the resulting matrix is copied back to the original buffer. This approach incurs additional CPU-to-CPU memory copying but maximizes PCIe bandwidth utilization and allows a high degree of GPU parallelization.

5.3 Algorithm-level Optimizations

Slot Compaction. SIMD-style processing facilitates the simultaneous manipulation of tens of thousands of numbers placed in slots and encoded within a single ciphertext, whereby an operation on the ciphertext is performed on all numbers. A high slot utilization increases both compute and memory efficiency. However, our analysis reveals that slot utilization is often low, especially for comparison, for three reasons. First, there are often discrepancies between the input size alignment and the available ciphertext slots, which persist even after optimizations. For instance, AlexNet's input size is 224 × 224, while SEAL [34] supports a maximum of 16,384 slots per ciphertext. Consequently, the input is partitioned into $\lceil \frac{224\times 224}{16384} \rceil = 4$ ciphertexts, resulting in 24% of slots being unused.

The second reason for slot under-utilization is slot wastage produced by prior operations. For example, in machine learning workloads, comparison occurs after other operations, such as matrix multiplication or convolution, that produce the waste. The third reason is that standard optimizations, e.g. in Helayers[1], to reduce future operations (by duplicating numbers in different slots) can actually amplify the slot wastage in the comparison operation. These create an opportunity to perform *slot compaction* prior to performing the comparison. Figure 5 illustrates an example of a neural network where convolution (and batch normalization) precedes ReLU in which comparison is performed (part (a)). For convolution between a matrix M and a filter (part (b)), the matrix M fills up all 16 ciphertext slots. Then, the convolution filter is duplicated to fill up slots (part (c)), in order to reduce the number of future multiplications, rotations, and additions, and to improve slot utilization. To obtain the convolution results, the multiplication is followed by only three sets of rotate-and-accumulate. The convolution results occupy the 4th, 8th, 12th, and 16th slots (shown in yellow), while all other slots do not contain useful values (or wasted).

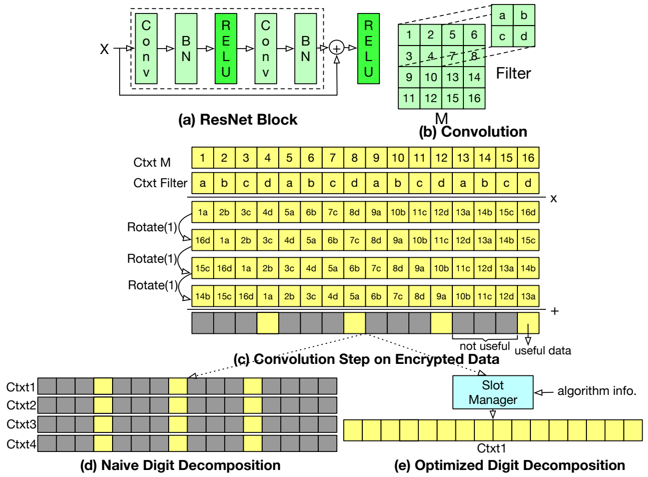

Figure 5: Illustration of: (a) ResNet block containing convolution (Conv), batch normalization (BN), and ReLU; (b) Convolution filter; (c) Convolution steps on encrypted data resulting in unused slots; (d) Naive digit decomposition with many unused slots; and (e) Optimized digit decomposition with slot compaction.

Next, to achieve comparison, Figure 5 (part (d)) illustrates the state-of-the-art practice where each convolution result is decomposed into digits (four digits are illustrated), resulting in amplifying the slot wastage across four ciphertexts, where even useless values are also decomposed into four digits. For comparison, only 25% slots have useful digits that are needed, which presents an opportunity for compaction. Our approach is shown in part (e), where we consolidate digits from all numbers into a single ciphertext. Through slot compaction, the comparison can now work on fewer ciphertext inputs, substantially reducing memory usage and unnecessary computation.

Realizing slot compaction in the Helayer is difficult because it cannot distinguish slots which contain useful data vs. those who do not, hence it must conservatively assume that all slots are useful. Besides, the existence of non-useful slots arises only when comparison is preceded by certain operations like convolution or matrix multiplication, so the Helayer cannot identify slot usefulness without algorithmic information. To overcome this challenge, we design a slot manager (SM) that preserves algorithm information to track slot usefulness to guide slot compaction after digit decomposition. With such information, SM can minimize memory usage by distributing digit decomposition across as few ciphertexts as possible.

When comparison is not preceded by other operations, we just perform slot compaction for the case when the input size does not align with the ciphertext format.

```
privateQuery(q, op1, op2){
 if(q == add)
    Data += op1
 else if(q == mult)
    Data *= op1
 else if(q == power)
    Data = Data^op2
 else
    Data = Data }
```

Listing 3: Private query on plaintext data.

```
privateQuery(q, op1, op2){
 c1 = EQ(q, add)
 c2 = EQ(q, mult)
 c3 = EQ(q, pwr)

 Data1 = Data + op1
 Data2 = Data * op1
 Data3 = Data.Power(op2)
 Data = Data1 * c1 + Data2 *
        c2 + Data3 * c3 }
```

Listing 4: Private query on encrypted data.

```
EvalBranch(c1, c2, c3, q){
 c1 = EQ(q, add)
 c2 = EQ(q, mult)
 c3 = EQ(q, pwr)
}
privateQuery(q, op1, op2) {
 helper_thread(EvalBranch(c1, c2, c3, q))
 Data1 = Data + op1
 Data2 = Data * op1
 Data3 = Data.Power(op2)
 thread_1.join()
 Data = Data1 * c1 + Data2 * c2 + Data3 * c3 }
```

Listing 5: Private query with non-blocking comparison.

Non-Blocking Comparison. When many numbers are compared together, the cost of comparison operation could be amortized using SIMD-style processing. However, when an application only needs to compare a pair of numbers (or a small number of pairs), comparison latency is hard to amortize. This case occurs when the comparison occurs inside an *if* statement.

Listing 3 shows an example code that performs a query without FHE (i.e., on unencrypted data). It takes three inputs: query type (q) and two data operands (op1 and op2). The code performs an operation (addition, multiplication, or exponentiation) based on the query type, with operand value specified by one of the two data operands. It has three comparisons each involving a pair of numbers. The semantic-equivalent FHE version is shown in Listing 4. With FHE, the query type is not in plaintext form, hence we must use the EQ(.) function to test for equality. Furthermore, the comparison results are also in ciphertext, hence conditional branches are replaced by code straightlining, resulting in Listing 4.

To hide the comparison latency that is hard to amortize, we propose *non-blocking comparison*. When the comparison is solely used to determine the taken branch path, there is no dependency relation between the main computation and the branch evaluation. Consequently, we can execute the branch evaluation and the main computation concurrently. Listing 5 shows the resulting code with our *non-blocking comparison* optimization. The branch evaluation that computes equality functions EQ(.) is performed by a helper thread in parallel to the arithmetic operations performed by the main thread. The final data update is performed after the helper thread joins the main thread.

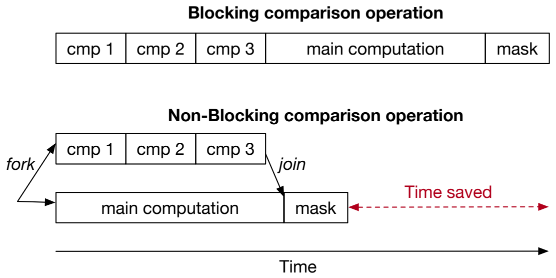

Figure 6: Illustrating the saved cycles due to the non-blocking comparison optimization.

To illustrate the benefit, Figure 6 compares the original straight-lined code performance (top) vs. with our non-blocking optimization (bottom). With non-blocking, the execution of branch evaluation overlaps with the main computation.

Non-Blocking Comparison. The conventional wisdom in BGV parameter selection favors *power-of-two* (PoT) polynomial ring degrees. However, non-PoT degrees, achieved by choosing prime numbers or cyclotomic polynomial ring orders, offer enhanced performance by ensuring cyclic slot permutation groups [15, 18]. While prior works often overlooked the impact of PoT degrees on comparison performance, our experimentation results demonstrate that non-PoT degrees lead to significantly faster comparisons.

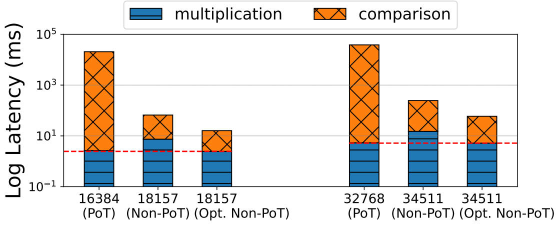

Figure 7: Comparison of the homomorphic operation latencies of using the power of two vs. non-power of two vs. optimized non-power of two polynomial rings.

As a demonstration, Figure 7 presents stacked bars representing the latencies of a single multiplication and a single comparison in *logarithmic scale* for a pair of m values. To qualify this, while ensuring a security level of $\lambda > 128$ bits, we have chosen two non-PoT m values, 18,157 and 34,511, to correspond with two specific PoT m values, 16,384 and 32,768. To ensure a meaningful comparison, these selections have been designed such that the non-PoT m values yield a slightly larger count of SIMD slots and an enhanced security level, as guided by the recommendations from [3, 18]. For each m value, we show the stacked latencies for three cases: PoT, unoptimized non-PoT, and non-PoT with our optimizations.

The figure shows that after optimization non-PoT latencies are over two orders of magnitude faster. Therefore, in this paper we choose non-PoT polynomial ring order due to the computation efficiency.

Table 3: Parameters and Statistics

Params	$(p\ m\ N)$	Circuit	$(d\ l)$	log(Q)	λ	no of int
p1	(3 34511 34510)	B U	(6 7) (16 4)	324 472	298 189	290 507
p2	(5 19531 19530)	B U	(7 4) (7 6)	324 354	155 141	697 465
p3	(7 20197 19116)	B U	(6 4) (8 4)	354 406	137 110	531 531
p4	(11 15797 15796)	B U	(5 4) (5 5)	342 378	162 145	359 287
p5	(13 30941 30940)	B U	(5 4) (4 6)	354 378	338 313	1547 1031
p6	(17 41761 41760)	B U	(4 4) (7 3)	413 472	402 344	1305 1740
p7	(19 29989 29988)	B U	(4 4) (5 4)	378 385	302 296	833 833
p8	(23 37745 30192)	B U	(5 3) (9 2)	413 456	275 245	838 1258
p9	(29 18157 17820)	B U	(5 3) (6 3)	360 413	175 150	990 990
p10	(31 52053 34700)	B U	(5 3) (4 4)	512 512	252 252	2313 1735

6 Methodology

Experiment Platforms. We evaluate BoostCom on a combination of GPU and CPU platforms. The GPU platform has an NVIDIA RTX 3090 GPU with 82 Streaming Multiprocessors (SMs). Each SM contains 128 CUDA cores, operating at a core clock speed of 1695 MHz. The GPU is equipped with a combined 10 MB of L1 data cache and shared memory, along with a separate 6 MB L2 cache. The GPU memory system has a 24GB size and 936 GB/s bandwidth.

The CPU platform has an AMD Ryzen PRO 3955WX CPU with 16 cores and 128 GB of memory. Each core has a clock speed of 3.9 GHz, with a 4.3 GHz maximum turbo frequency. It has 64MB L3 Cache Memory and its main memory has eight-channel ECC DDR4-3200 DRAMs. The CPU runs Ubuntu OS version 22.04 and NVIDIA driver version 525.85.12. We used CUDA version 12.0 and GCC version 7.5.0 for compilation. The key GPU kernel implementations can be accessed at this repository.

Workload Evaluation Methodology. We evaluate BoostCom with full applications to measure overall application performance as well as with microbenchmark to measure comparison performance specifically. In both cases, our evaluation measures end-to-end performance, in contrast to extrapolating from the measurement of each operation that is common in prior works. End-to-end performance measurement gives a fuller and more reliable picture of the performance. For all measurements, we repeat each experiment 10 times and report their average. We use the NVIDIA Nsight system to collect hardware performance statistics.

State-of-the-art BGV accelerator. We compare our work with state-of-the-art FHE GPU-acceleration HE-Booster [40]. HE-Booster accelerates NTT operations by introducing fine granularity of thread synchronization for every iteration inside NTT operations. Additionally, the paper proposes the fusion of operations within the key-switching procedure. We implemented HE-Booster in the HELib library and evaluated it end-to-end.

Microbenchmark. To measure comparison-only performance, we form a microbenchmark that performs a comparison of a pair of 64-bit integers. We vary the BGV parameters to form 10 different configurations following prior work [18]. Each configuration is expressed as a tuple of $(p\ m\ N)$ and was selected to maximize the number of SIMD slots as shown in Table 3. They are sorted in the order of increasing plaintext modulus p values. Each configuration uses bivariate and univariate circuits with differing vector space dimension d, vector length l, and the product of prime moduli Q. The resulting security level λ and number of integers that can fit in one ciphertext are shown in the last two columns.

Applications. As there is currently no standardized benchmark for evaluating comparison operations in BGV, we developed **mlp**, **img_col**, and **private_q**, and adopted **sorting** and **min** from prior work [18]. Below, we provide details for each benchmark:

sorting is an application that sorts an array of 16 encrypted 32-bit integers from [18]. It uses univariate circuit for comparison, utilizes a matrix of Hamming weights to establish the relationship between any pair of elements in the encrypted array.

min is an application that finds a minimum integer from an array of 16 elements of 32-bit integers from [18]. It uses univariate circuit and combines the Hamming weight matrix and the tournament methodology, reducing the circuit's depth for improved efficiency.

mlp is a simple machine learning program utilizing Multi-Layer Perceptron that we wrote to classify images. It has three layers: a fully-connected layer, ReLU, and another fully-connected layer. The bivariate circuit is used for comparison in the ReLU layer. mlp performs inference using encrypted 16-bit integers. The input image has 28x28 pixels, stored in a single ciphertext. It trains on MNIST datasets and outputs ten nodes.

img_col is an image recolorizing application that we developed to calculate the distance of every pixel inside an image to a threshold value. When the distance is below the threshold, it transforms the pixel by multiplying its color value with the pre-set value. The bivariate circuit is used for comparison. This application enables private medical data image analysis on an untrusted cloud server. The input is an encrypted image, threshold value, and pre-set pixel transformation value. The input image is encoded into 16 ciphertexts and each ciphertext consists of 700 pixels.

private_q is a simple application that we developed to perform a private query to manipulate data in encrypted databases, based on Listing 6. The database consists of 100 ciphertexts, and each ciphertext holds 2124 integers. This application helps evaluate the proposed *non-blocking comparison.*

7 Evaluation Results

7.1 Workloads Speedup

Figure 8 illustrates the speedups achieved by BoostCom and HE-booster compared to a 16-core CPU-only baseline (i.e., HElib [17]), for all applications and their geometric mean speedup. BoostCom (second bars) include library-level optimizations, i.e., heterogeneous multi-CPU/GPU parallelization, slot compaction, and non-blocking optimization.

Table 4: GPU Time Utilization.

Scheme	sorting	min	mlp	img_col	private_q
GPU-only	17%	8%	20%	15%	17%
Heterogeneous	35%	16%	41%	32%	33%

It is important to note that the speedups are measured for *end-to-end execution times*, encompassing various operations, not just the

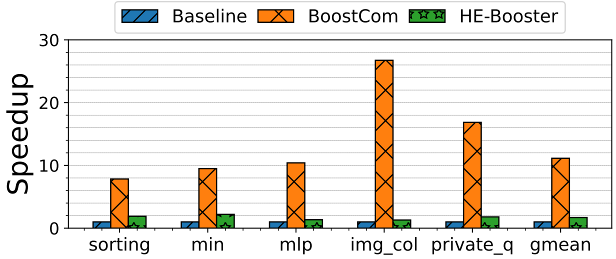

Figure 8: The acceleration achieved by BoostCom in comparison to the baseline for five important workloads.

Table 5: Memory Usage for each Workload (GB).

Scheme	sorting	min	mlp	img_col	private_q	gmean
Heterogeneous	5.5	8.4	1.6	2.8	2.2	3.5
Heterogeneous+SM	4.5	6.5	1.1	1.6	1.4	2.5
Mem. Reduction	19%	23%	32%	44%	35%	29.3%

comparison. This includes all overhead such as CPU-GPU memory copy, kernel launches, synchronization, etc. As shown from the figure, BoostCom achieved a speedup of 11.1× (up to 26.7×). In contrast, the state-of-the-art HE-Booster only achieves an average of 1.7× speedup. Thus, our Boostcom scheme achieves a 553% higher speedup than HE-Booster on average.

Figure 9 illustrates the impact of each BoostCom's optimization. The GPU-only denotes library-level optimizations with offloading the computation-intensive portions of the library to the GPU with only one CPU core, SM denotes the usage of slot manager for compaction, The Heterogeneous denotes the usage of a multicore CPU to submit more work to the GPU, and NB adds the non-blocking optimization. On average, the GPU-only acceleration only achieved a gmean speedup of 3.6× over the baseline. The gmean speedup triples when multi CPU core and slot compaction is added, reaching 11.1×. This demonstrates the effectiveness of Boostcom's heterogeneous scheme and the slot compaction that reduces the number of ciphertexts involved in comparison. As depicted in the figure, each optimization demonstrates a significant effect on the speedup, highlighting the effectiveness of each of the optimization.

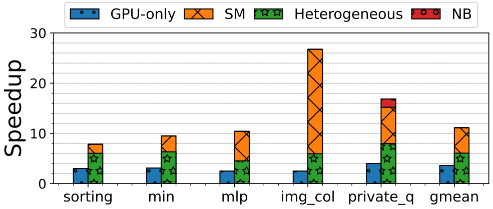

Figure 9: The breakdown of speedup for each optimization compared to the baseline for five critical workloads.

Boostcom's multi-level heterogeneous parallelization strategy roughly doubles the GPU utilization (Table 4) while slot compaction reduces the memory usage by 29.3% on average (Table 5). The reduction clearly correlates with the speedups; the greater the memory usage reduction, the higher the speedup.

In the subsequent subsection, we analyze the effect of each optimization at the library level employed in BoostCom concerning only the comparison operation.

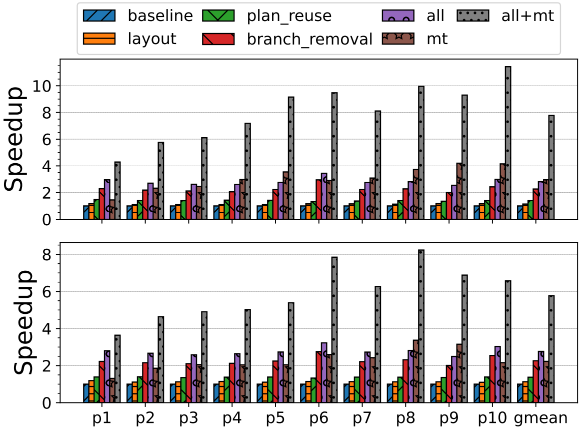

Figure 10: Speedups of the comparison ops for the Bivariate (top) and Univariate (bottom) circuit over the baseline.

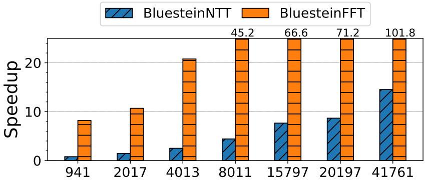

Figure 11: The comparison between BluesteinNTT and BluesteinFFT speedup over each baseline with the increasing parameter *m*.

7.2 Comparison Operation Speedup

Figure 10 compares the end-to-end execution time of comparison of encrypted 64-bit int, over the 16-core CPU-only baseline for Bivariate (top) and Univariate (bottom) circuits, across 10 different BGV configurations from Table 3. For each configuration, six bars are shown with increasing optimization levels, starting from the baseline, layout transformation, branch removal, plan reuse, the combination of three said optimizations (*all*), digit level parallelization with CPU multithreading (*mt*), and all optimizations including multithreading (*all+mt*). Note that slot compaction and non-blocking comparison optimizations are not applicable here since there is no other computation aside from the comparison itself.

For both circuits across all configurations, each optimization adds additional speedups, indicating their effectiveness. With *all*, the geometric mean (gmean) speedup is 2.8× for both circuits. On its own, multithreading for digit-level parallelization is somewhat effective (gmean speedup of 2.9× (Bivariate) and 2.2× (Univariate)). But when combined with all other optimizations, multithreading enables much higher speedups, reaching 7.8× (Bivariate) and 5.8× (Univariate), due to the synergistic effect where multithreading significantly improving the GPU utilization (by between 30% and 260%).

Roughly, as *p* increases, the effectiveness of multithreading increases whereas that of other optimizations remains unchanged. This is because as the degree *d* increases, the fraction of execution

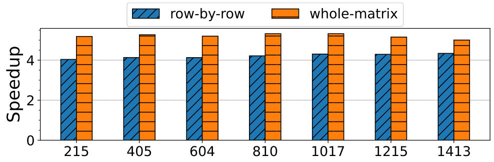

Figure 12: Element-wise ops speedups of whole matrix approach vs. row-by-row GPU offloading, as $log(Q)$ increases.

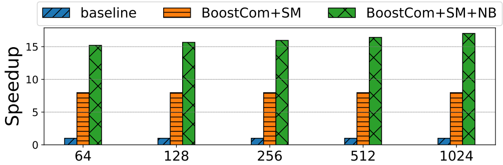

Figure 13: Speedups of optimizations without vs. with non-blocking as the branch evaluation computation increases with larger exponent values.

time spent on the *BivarCircuit*, *EqualityCircuit*, and *UnivarCircuit* increases.

7.3 BluesteinNTT and BluesteinFFT Sensitivity Study

Impact of the parameter m. Increasing multiplicative depth without sacrificing security may lead to larger m. To evaluate its effect on BoostCom, we vary m from 941 to 41,761, resulting in polynomial size expansions ranging from 2,048 to 131,072. The resulting speedups of BluesteinNTT and BlusteinFFT, calculated over CPU-only execution are shown in Figure 11 (top). The figure shows that the larger the m, the higher the speedups, indicating BoostCom's scalability.

7.4 Element-wise Sensitivity Study

Figure 12 compares the speedups of BoostCom's *layout transformation* compared to performing element-wise operation row-by-row, as Q increases. A larger Q increases the noise budget and allows a more complex application but with slower computation. The figure shows that the speedups of our layout optimization is quite stable across all values of Q.

7.5 Non-blocking Comparison Sensitivity Study

To evaluate the sensitivity of BoostCom's *non-blocking* optimization performance, we vary the exponent (op2) from 64 to 1024 as exponentiation is the most expensive operation. (Figure 13). The figure shows the speedups are stable, with increasing non-blocking effectiveness (as a larger portion of the branch evaluation is hidden).

8 Conclusion

We proposed accelerating uFHE-based BGV scheme's non-arithmetic comparisons on CPU/GPU systems through innovative optimizations, including multi-level heterogeneous parallelization, GPU optimizations, and algorithmic designs. This combination of optimizations proved highly effective, achieving an 11.1× speedup (with peaks up to 26.7×) across five key FHE applications, significantly outperforming the prior approach.

Acknowledgments

We thank the anonymous reviewers for their valuable comments. We also thank Rex McCrary for his feedback on this work. This work is supported in part by an AMD gift fund.

References

[1] Ehud Aharoni, Allon Adir, Moran Baruch, Nir Drucker, Gilad Ezov, Ariel Farkash, Lev Greenberg, Ramy Masalha, Guy Moshkowich, Dov Murik, Hayim Shaul, and Omri Soceanu. 2023. HeLayers: A Tile Tensors Framework for Large Neural Networks on Encrypted Data. *Privacy Enhancing Technology Symposium (PETs) 2023* (2023). https://petsymposium.org/popets/2023/popets-2023-0020.php

[2] Ehud Aharoni, Nir Drucker, and Hayim Shaul. 2022. Advanced HE packing methods with applications to ML. In *ACM Annual Conference on Computer and Communications Security.*

[3] Martin R. Albrecht, Rachel Player, and Sam Scott. 2015. On the concrete hardness of Learning with Errors. *Journal of Mathematical Cryptology* 9, 3 (10 2015), 169–203. https://doi.org/10.1515/jmc-2015-0016

[4] Ardhi Wiratama Baskara Yudha, Keiji Kimura, Huiyang Zhou, and Yan Solihin. 2020. Scalable and Fast Lazy Persistency on GPUs. In *2020 IEEE International Symposium on Workload Characterization (IISWC).* 252–263. https://doi.org/10.1109/IISWC50251.2020.00032

[5] Fabian Boemer, Sejun Kim, Gelila Seifu, Fillipe D.M. de Souza, and Vinodh Gopal. 2021. Intel HEXL: Accelerating Homomorphic Encryption with Intel AVX512-IFMA52. In *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography* (Virtual Event, Republic of Korea) *(WAHC '21).* Association for Computing Machinery, New York, NY, USA, 57–62. https://doi.org/10.1145/3474366.3486926

[6] Christina Boura, Nicolas Gama, Mariya Georgieva, and Dimitar Jetchev. 2018. CHIMERA: Combining Ring-LWE-based Fully Homomorphic Encryption Schemes. Cryptology ePrint Archive, Paper 2018/758. https://eprint.iacr.org/2018/758 https://eprint.iacr.org/2018/758.

[7] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2012. (Leveled) Fully Homomorphic Encryption without Bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference* (Cambridge, Massachusetts) *(ITCS '12).* Association for Computing Machinery, New York, NY, USA, 309–325. https://doi.org/10.1145/2090236.2090262

[8] CapePrivacy. 2021. Cape Privacy: Privacy & trust management for machine learning. https://capeprivacy.com/.

[9] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic Encryption for Arithmetic of Approximate Numbers. In *Advances in Cryptology – ASIACRYPT 2017,* Tsuyoshi Takagi and Thomas Peyrin (Eds.). Springer International Publishing, Cham, 409–437.

[10] Jung Hee Cheon, Dongwoo Kim, and Duhyeong Kim. 2019. Efficient Homomorphic Comparison Methods with Optimal Complexity. Cryptology ePrint Archive, Paper 2019/1234. https://eprint.iacr.org/2019/1234 https://eprint.iacr.org/2019/1234.

[11] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2016. Faster Fully Homomorphic Encryption: Bootstrapping in Less Than 0.1 Seconds. In *Advances in Cryptology – ASIACRYPT 2016,* Jung Hee Cheon and Tsuyoshi Takagi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 3–33.

[12] DualityTechnologies. 2022. DualityTechnologies: Data encryption technology and secure collaboration. https://dualitytech.com/.

[13] S. Fan, Z. Wang, W. Xu, R. Hou, D. Meng, and M. Zhang. 2023. TensorFHE: Achieving Practical Computation on Encrypted Data Using GPGPU. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA).* IEEE Computer Society, Los Alamitos, CA, USA, 922–934. https://doi.org/10.1109/HPCA56546.2023.10071017

[14] Craig Gentry et al. 2009. Fully homomorphic encryption using ideal lattices.. In *ACM Symposium on Theory of Computing,* Vol. 9. 170–178.

[15] Craig Gentry, Shai Halevi, and Nigel P. Smart. 2012. Fully Homomorphic Encryption with Polylog Overhead. In *Advances in Cryptology – EUROCRYPT 2012,* David Pointcheval and Thomas Johansson (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 465–482.

[16] Torbjrn Granlund and Gmp Development Team. 2015. *GNU MP 6.0 Multiple Precision Arithmetic Library.* Samurai Media Limited, London, GBR.

[17] Shai Halevi and Victor Shoup. 2020. Design and implementation of HElib: a homomorphic encryption library. Cryptology ePrint Archive, Paper 2020/1481. https://eprint.iacr.org/2020/1481 https://eprint.iacr.org/2020/1481.

[18] Ilia Iliashenko and Vincent Zucca. 2021. Faster homomorphic comparison operations for BGV and BFV. Cryptology ePrint Archive, Paper 2021/315. https://eprint.iacr.org/2021/315 https://eprint.iacr.org/2021/315.

[19] Inpher. 2022. Inpher: Secret computing and privacy-preserving analytics. https://www.inpher.io/.

[20] Lei Jiang and Lei Ju. 2022. FHEBench: Benchmarking Fully Homomorphic Encryption Schemes. arXiv:2203.00728 [cs.CR]

[21] Wonkyung Jung, Sangpyo Kim, Jung Ho Ahn, Jung Hee Cheon, and Younho Lee. 2021. Over 100x Faster Bootstrapping in Fully Homomorphic Encryption through Memory-centric Optimization with GPUs. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2021, 4 (2021), 114–148. https://doi.org/10.46586/tches.v2021.i4.114-148

[22] Jongmin Kim, Sangpyo Kim, Jaewan Choi, Jaiyoung Park, Donghwan Kim, and Jung Ho Ahn. 2023. SHARP: A Short-Word Hierarchical Accelerator for Robust and Practical Fully Homomorphic Encryption. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 18, 15 pages. https://doi.org/10.1145/3579371.3589053

[23] Jongmin Kim, Gwangho Lee, Sangpyo Kim, Gina Sohn, Minsoo Rhu, John Kim, and Jung Ho Ahn. 2022. ARK: Fully Homomorphic Encryption Accelerator with Runtime Data Generation and Inter-Operation Key Reuse. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1237–1254. https://doi.org/10.1109/MICRO56248.2022.00086

[24] Sangpyo Kim, Jongmin Kim, Michael Jaemin Kim, Wonkyung Jung, John Kim, Minsoo Rhu, and Jung Ho Ahn. 2022. BTS: An Accelerator for Bootstrappable Fully Homomorphic Encryption. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 711–725. https://doi.org/10.1145/3470496.3527415

[25] Eunsang Lee, Joon-Woo Lee, Young-Sik Kim, and Jong-Seon No. 2022. Optimization of Homomorphic Comparison Algorithm on RNS-CKKS Scheme. *IEEE Access* 10 (2022), 26163–26176. https://doi.org/10.1109/ACCESS.2022.3155882

[26] Qian Lou, Muhammad Santriaji, Ardhi Wiratama Baskara Yudha, Jiaqi Xue, and Yan Solihin. 2023. vFHE: Verifiable Fully Homomorphic Encryption with Blind Hash. arXiv:2303.08886 [cs.CR] https://arxiv.org/abs/2303.08886

[27] Wen-jie Lu, Zhicong Huang, Cheng Hong, Yiping Ma, and Hunter Qu. 2021. PEGASUS: bridging polynomial and non-polynomial evaluations in homomorphic encryption. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1057–1073.

[28] Ali Sah Ozcan, Can Ayduman, Enes Recep Turkoglu, and Erkay Savas. 2023. Homomorphic Encryption on GPU. *IEEE Access* (2023), 1–1. https://doi.org/10.1109/ACCESS.2023.3265583

[29] Ozgun Ozerk, Can Elgezen, Ahmet Can Mert, Erdinc Ozturk, and Erkay Savas. 2021. Efficient Number Theoretic Transform Implementation on GPU for Homomorphic Encryption. Cryptology ePrint Archive, Paper 2021/124. https://eprint.iacr.org/2021/124 https://eprint.iacr.org/2021/124.

[30] NVIDIA Pascal. 2016. NVIDIA Tesla P100 Whitepaper.

[31] M. Sadegh Riazi, Kim Laine, Blake Pelton, and Wei Dai. 2020. HEAX: An Architecture for Computing on Encrypted Data. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1295–1309. https://doi.org/10.1145/3373376.3378523

[32] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald Dreslinski, Christopher Peikert, and Daniel Sanchez. 2021. F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 238–252. https://doi.org/10.1145/3466752.3480070

[33] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Nathan Manohar, Nicholas Genise, Srinivas Devadas, Karim Eldefrawy, Chris Peikert, and Daniel Sanchez. 2022. CraterLake: A Hardware Accelerator for Efficient Unbounded Computation on Encrypted Data. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 173–187. https://doi.org/10.1145/3470496.3527393

[34] SEAL 2023. Microsoft SEAL (release 4.1). https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA..

[35] Shiyu Shen, Hao Yang, Yu Liu, Zhe Liu, and Yunlei Zhao. 2022. CARM: CUDA-Accelerated RNS Multiplication in Word-Wise Homomorphic Encryption Schemes for Internet of Things. *IEEE Trans. Comput.* (2022), 1–12. https://doi.org/10.1109/TC.2022.3227874

[36] Kaustubh Shivdikar, Yuhui Bao, Rashmi Agrawal, Michael Shen, Gilbert Jonatan, Evelio Mora, Alexander Ingare, Neal Livesay, José L Abellán, John Kim, et al. 2023. GME: GPU-based Microarchitectural Extensions to Accelerate Homomorphic Encryption. *2023 56th IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2023).

[37] K. Shivdikar, G. Jonatan, E. Mora, N. Livesay, R. Agrawal, A. Joshi, J. L. Abellan, J. Kim, and D. Kaeli. 2022. Accelerating Polynomial Multiplication for Homomorphic Encryption on GPUs. In *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*. IEEE Computer Society, Los Alamitos, CA, USA, 61–72. https://doi.org/10.1109/SEED55351.2022.00013

[38] Sujoy Sinha Roy, Furkan Turan, Kimmo Jarvinen, Frederik Vercauteren, and Ingrid Verbauwhede. 2019. FPGA-Based High-Performance Parallel Architecture for Homomorphic Computing on Encrypted Data. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 387–398. https://doi.org/10.1109/HPCA.2019.00052

[39] Benjamin Hong Meng Tan, Hyung Tae Lee, Huaxiong Wang, Shuqin Ren, and Khin Mi Mi Aung. 2021. Efficient Private Comparison Queries Over Encrypted Databases Using Fully Homomorphic Encryption With Finite Fields. *IEEE Transactions on Dependable and Secure Computing* 18, 6 (2021), 2861–2874. https://doi.org/10.1109/TDSC.2020.2967740

[40] Zhiwei Wang, Peinan Li, Rui Hou, Zhihao Li, Jiangfeng Cao, XiaoFeng Wang, and Dan Meng. 2023. HE-Booster: An Efficient Polynomial Arithmetic Acceleration on GPUs for Fully Homomorphic Encryption. *IEEE Transactions on Parallel and Distributed Systems* 34, 4 (2023), 1067–1081. https://doi.org/10.1109/TPDS.2022.3228628

[41] Shougang Yuan, Amro Awad, Ardhi Wiratama Baskara Yudha, Yan Solihin, and Huiyang Zhou. 2022. Adaptive Security Support for Heterogeneous Memory on GPUs. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 213–228. https://doi.org/10.1109/HPCA53966.2022.00024

[42] Shougang Yuan, Ardhi Wiratama Baskara Yudha, Yan Solihin, and Huiyang Zhou. 2021. Analyzing Secure Memory Architecture for GPUs. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 59–69. https://doi.org/10.1109/ISPASS51385.2021.00017

[43] Ardhi Yudha and Reza Pulungan. 2017. A Load-Balanced Parallelization of AKS Algorithm. *TELKOMNIKA Indonesian Journal of Electrical Engineering* 15 (12 2017), 1884. https://doi.org/10.12928/telkomnika.v15i4.6049

[44] Ardhi Wiratama Baskara Yudha, Jake Meyer, Shougang Yuan, Huiyang Zhou, and Yan Solihin. 2022. LITE: a low-cost practical inter-operable GPU TEE. In *Proceedings of the 36th ACM International Conference on Supercomputing* (Virtual Event) *(ICS '22)*. Association for Computing Machinery, New York, NY, USA, Article 7, 13 pages. https://doi.org/10.1145/3524059.3532361

[45] Ardhi Wiratama Baskara Yudha, Reza Pulungan, Henry Hoffmann, and Yan Solihin. 2020. A Simple Cache Coherence Scheme for Integrated CPU-GPU Systems. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1109/DAC18072.2020.9218664

[46] Zama. 2022. Concrete ML. https://www.zama.ai/concrete-ml.

[47] Yilan Zhu, Xinyao Wang, Lei Ju, and Shanqing Guo. 2023. FxHENN: FPGA-based acceleration framework for homomorphic encrypted CNN inference. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 896–907. https://doi.org/10.1109/HPCA56546.2023.10071133