IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received June 14, 2021, accepted June 26, 2021, date of publication July 12, 2021, date of current version July 19, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3096189

Accelerating Fully Homomorphic Encryption Through Architecture-Centric Analysis and Optimization

WONKYUNG JUNG[1], (Graduate Student Member, IEEE), EOJIN LEE[2], SANGPYO KIM[1], (Graduate Student Member, IEEE), JONGMIN KIM[3], NAMHOON KIM[1], KEEWOO LEE[4], CHOHONG MIN[5], JUNG HEE CHEON[4,6], (Associate Member, IEEE), AND JUNG HO AHN[1,7], (Senior Member, IEEE)

[1]Department of Intelligence and Information, Seoul National University, Seoul 08826, South Korea
[2]Samsung Electronics, Suwon 16677, South Korea
[3]Interdisciplinary Program in Artificial Intelligence, Seoul National University, Seoul 08826, South Korea
[4]Department of Mathematical Sciences, Seoul National University, Seoul 08826, South Korea
[5]Department of Mathematics, Ewha Womans University, Seoul 03760, South Korea
[6]Crypto Lab Inc., Seoul 08826, South Korea
[7]Inter-university Semiconductor Research Center, Seoul National University, Seoul 08826, South Korea

Corresponding author: Jung Ho Ahn (gajh@snu.ac.kr)

This work was supported in part by Samsung Electronics Company Ltd., under Grant IO201207-07812-01, in part by the National Research Foundation of Korea (NRF) Grant through the Korean Government (MSIT) under Grant 2020R1A2C2010601 (50%), and in part by the Institute of Information and communications Technology Planning and evaluation (IITP) grants through the Korean Government by the Artificial Intelligence Graduate School Program (Seoul National University) under Grant 2021-0-01343 (10%) and the Development and Library Implementation of Fully Homomorphic Machine Learning Algorithms supporting Neural Network Learning Over Encrypted Data under Grant 2020-0-00840 (40%).

ABSTRACT Homomorphic Encryption (HE) has drawn significant attention as a privacy-preserving approach for cloud computing because it allows computation on encrypted messages called ciphertexts. Among the numerous HE schemes proposed thus far, HE for Arithmetic of Approximate Numbers (HEAAN) is rapidly gaining in popularity across a wide range of applications, as it supports messages that can tolerate approximate computations with no limit on the number of arithmetic operations applicable to the ciphertexts. A critical shortcoming of HE is the high computation complexity of ciphertext arithmetic; specifically, HE multiplication (HE Mul) is more than 10,000 times slower than the corresponding multiplication between unencrypted messages. This has led to a large body of HE acceleration studies, including those that exploit FPGAs; however, a rigorous analysis of the computational complexity and data access patterns of HE Mul is lacking. Moreover, the proposals mostly focused on designs with small parameter sizes, making it difficult accurately to estimate the performance of the HE accelerators when conducting a series of complex arithmetic operations. In this paper, we first describe how HE Mul of HEAAN is performed in a manner friendly to non-crypto experts. Then, we conduct a disciplined analysis of its computational and memory-access characteristics, through which we (1) extract parallelism in the key functions composing HE Mul and (2) demonstrate how to map the parallelism effectively to popular parallel processing platforms, CPUs and GPUs, by applying a series of optimizations such as transposing matrices and pinning data to threads. This leads to performance improvements of HE Mul on a CPU and a GPU by 2.06× and 4.05×, respectively, over the reference HEAAN running on a CPU with 24 threads.

INDEX TERMS Computer applications, computer architecture, cryptography, multicore processing.

I. INTRODUCTION

As cloud computing becomes an increasingly dominant means of providing computing resources, numerous computations are performed on datacenter servers rather than on personal devices [7], [41]. This enables a client without expensive hardware to receive services requiring complex computations. However, security and privacy issues are emerging with the growth of cloud computing [26], [65]. When a client sends private data to a server, security issues

The associate editor coordinating the review of this manuscript and approving it for publication was Chien-Ming Chen.

during data transfers can be resolved by sending the data after encryption. However, data encoded by a conventional encryption method must be decrypted to perform computation in the server. Therefore, a user has no choice but to use the cloud service with the risk of security or privacy attacks (e.g., abuses) that can occur during the computation of unencrypted data.

Homomorphic Encryption (HE) [62], an encryption scheme that enables computation between encrypted data, has attracted significant attention as a solution to this privacy problem. The concept of HE was initially suggested in 1978 [62]. However, early proposals of HE were either unsafe [62] or supported only one type of HE operations, namely HE addition (HE Add) or HE multiplication (HE Mul) (e.g., ElGamal [32] and Paillier [60]). Accordingly, it was difficult to put HE into serious applications. However, the fully HE (FHE) [34] proposed in 2009 made a major breakthrough by supporting both HE Add and HE Mul. Moreover, FHE supports bootstrapping, a method of initializing noise in encrypted data, enabling an unbounded number of HE Add and Mul operations without decryption.

Among numerous FHE schemes [14], [20], [21], [31], [33], HE for Arithmetic of Approximate Numbers (HEAAN [20]), also known as CKKS (Cheon-Kim-Kim-Song) is rapidly gaining popularity [42], [43] as it supports the approximate computation. HEAAN enables the HE Add and Mul of approximate data, with the result nearly identical to that of the original operation with only a tiny degree of error. However, the execution time for computation on encrypted data (ciphertext) increases by 100s to 10,000s of times compared to that on native, unencrypted messages. Therefore, it is highly desired to reduce the computation time of HE operations before HE can be used practically.

Numerous studies have focused on accelerating HE operations. However, FPGA-based acceleration studies target HE schemes (e.g., BGV [14], LTV [51], and BFV [33]) that only support computations of integer numbers [24], [28], [59], [63], [64], or that operate with only limited parameter sizes [61]. They all target performing a small number of HE Mul without considering bootstrapping, inhibiting their applicability to a wide range of applications requiring hundreds to thousands of multiplications (e.g., deep learning). GPU implementation studies [6], [9], [10], [25] do not exploit the algorithm's internal parallelism sufficiently, operate on only limited parameters, or do not optimize modulo operations.

In this paper, we demystify HEAAN, a representative FHE scheme, by describing/analyzing/optimizing it in a way friendly for non-crypto experts. We explain the pertinent details of the encryption/decryption/computation aspects of HEAAN, demonstrating that the following four functions account for more than 95% of HE Mul, the most computationally expensive operation of HE: CRT (Chinese Remainder Theorem), NTT (Number Theoretic Transform), iNTT (inverse NTT), and iCRT (inverse CRT). We conduct an in-depth and disciplined analysis of these primary functions to grasp their computational complexity and access patterns on input, output, and precomputed data (critical for the strength reduction of operations such as modular multiplication), across a range of key HE parameters.

The parallelism exposed through the analysis is exploited to accelerate HE Mul on a CPU and GPU, the most popular computing platforms, which are already equipped with hundreds to thousands of ALUs. On a CPU, we utilize multiple cores (inter-core parallelism) and AVX-512 instructions supported by the latest Intel architectures (intra-core parallelism). On a GPU, we utilize massive thread-level parallelism expressible through the CUDA programming model. We improve the performance further by proposing a series of optimizations, in this case matrix transposition to better exploit access locality, loop reordering to expose more parallelism, and to utilize the synergy between precomputation and delayed modulo operations. We achieve $2.06\times$ and $4.05\times$ speedups of HE Mul on a single CPU and a GPU, respectively, compared to the reference HEAAN [1] on a 24-core CPU.

II. BACKGROUND: COMPUTATIONAL CHALLENGES OF HE

HE can be categorized into two groups, somewhat HE (SHE) and fully HE (FHE), according to whether there is a limitation on the number of arithmetic operations applicable to the ciphertext. In a HE scheme, noise is accumulated during each operation; this makes the ciphertext of a SHE scheme indecipherable after performing a certain number of operations. In contrast, FHE schemes support a bootstrapping algorithm [19], which refreshes the accumulated noise. Therefore, although there is an upper bound with regard to the number of arithmetic operations that can be consecutively applied to a ciphertext, by periodic bootstrapping, we can continue to manipulate the ciphertext without decrypting it. This property makes FHE well-tailored to meet the demands of a wide range of applications (e.g., deep neural networks [8], [13], [22], [30], [48]) that require a massive number of operations applied to encrypted data.

Representative FHE schemes include Brakerski-Gentry-Vaikuntanathan (BGV) [14], Lopez-Alt, Tromer, and Vaikuntanathan (LTV) [51], Brakerski/Fan-Vercauteren (BFV) [33], fast FHE over the torus (TFHE) [21], and Cheon-Kim-Kim-Song (CKKS) [20]. Among these, only CKKS supports approximate computations on real numbers, making it a top candidate for many real-world applications requiring numerous operations on data that can tolerate tiny errors due to the approximate computation characteristic. CKKS is rapidly gaining in popularity in a wide range of applications exploiting HE, such as machine learning [48]. For example, the winner and the most runner-ups of a recent HE challenge involving a secure genome analysis competition (iDASH 2018 [42] and 2019 [43]) used CKKS or corresponding hybrid versions. Therefore, we investigate the HEAAN (HE for Arithmetic of Approximate Numbers) scheme [1] developed by the authors of CKKS.

HEAAN can perform arbitrary types of computations by combining HE multiplication (*HE Mul*) and HE addition (*HE Add*), which are correspondingly multiplication (mul) and addition on ciphertexts. However, the execution time of HE operations increases significantly compared to the corresponding times of the original unencrypted messages. Table 1 compares the execution time for the addition/mul operations of original messages and ciphertexts using a single core from the system specified in Section VI. We measure the execution time of the addition/mul of a complex number in a message consisting of 32,768 complex numbers. HE Mul (Add) is 27,721× (168×) slower than the native message operation. When the message consists of fewer numbers, the slowdown is even greater.

TABLE 1. Execution time for addition and multiplication of messages vs. ciphertexts in HEAAN.

Operation type	Message	Ciphertext	Slowdown
Addition	2.1 ns	348.2 ns	168.2×
Multiplication	4.3 ns	119201.7 ns	27721.3×

Considering that most approximate number operations consist of mul/addition, the long execution times for HE operations represent an obstacle preventing the practical use of HE. Figure 1 shows the time-breakdown when training a binary classification logistic regression model with HE [37], [38] with a subset of an MNIST dataset [50]. One training iteration with a mini-batch (whose size being 1024) takes 109.2 seconds, which is 68,250× slower than that without HE (i.e., training with unencrypted messages [37]). Out of the total training time, HE Mul and HE Rotation (HE Rot) account for 80%. HE Rot is a homomorphic operation that performs a circular shift on a message, which is a vector of complex numbers in HEAAN. HE Rot is similar but slightly less computationally expensive than HE Mul, as they share the same expensive core functions as HE Mul. Therefore, to use HE practically, it is essential to accelerate the HE operations, especially HE Mul as it is 342× slower than HE Add.

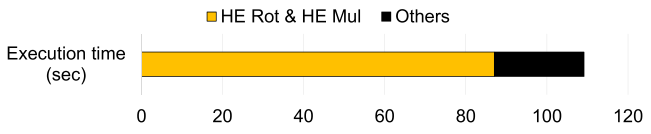


FIGURE 1. Time-breakdown of one iteration in training a binary logistic regression model using HE [38]. We trained the model with an MNIST dataset and a batch size of 1024 as described in [38] by running their code [37] on a system specified in Section VI.

III. A BRIEF INTRODUCTION TO HEAAN

Prior to accelerating HE Mul, we introduce the pertinent details of HEAAN, focusing on how to convert an input message to a ciphertext through encoding/encryption steps and how to perform arithmetic operations on ciphertexts.

A. HEAAN ENCODING AND ENCRYPTION

HEAAN converts an input message to a ciphertext through encoding and encryption steps. An input message consists of n complex numbers, each composed of a double-type real and imaginary number. The encoding step initially converts an input message to a plaintext (t), a polynomial of at most degree $(N-1)$ with N integer coefficients. t is placed in a cyclotomic polynomial ring ($R = \mathbb{Z}[X]/(X^N+1) = c_0X^0 + c_1X^1 + \ldots + c_{(N-1)}X^{(N-1)}$) space with the magnitude of each coefficient bounded by the ciphertext modulus, the integer q ($t \in R/qR$). Therefore, each coefficient (c_k) is the residue number according to q, where q is a BigInt (big integer) much larger than 2^{64}. The encoding step converts the floating-point numbers of a message to integer numbers after multiplication with a scaling factor (Δ), after which the remaining fraction numbers are rounded down. Each coefficient is a $\log q$-bit BigInt, and $n \leq \frac{N}{2}$.

Then, the encryption step converts a plaintext to a ciphertext consisting of a pair of polynomials $c.ax$ and $c.bx$ using a public key pair (pk_0 and pk_1) as follows:

$$c.ax = u \times pk_1 + e_1$$
$$c.bx = u \times pk_0 + e_0 + t$$

In the equations above, u is a polynomial of at most degree $(N-1)$, where each coefficient is either -1, 0, or 1 following the distribution described in [20]. e_0 and e_1 are also polynomials of at most degree $(N-1)$ with random error values to ensure security, following a Gaussian distribution with a small standard deviation value (e.g., $\sigma = 3.2$ in [20]). pk_1 is a polynomial of at most degree $(N-1)$ with random coefficients bounded by q, whereas pk_0 is a product of pk_1 and a secret key (sk) plus a polynomial having small (similar to e_0 and e_1) random errors.

To extract the original message from a ciphertext, first we convert the ciphertext to plaintext by exploiting the following relationship between $c.ax$ and $c.bx$:

$$c.bx = c.ax \times sk + t + e'$$

Then, the plaintext t can be returned to the original message through decoding; in this case, the inverse of scaling factor ($1/\Delta$) is multiplied to obtain the approximate values.

HEAAN limits the maximum size of the ciphertext modulus q to a constant Q. HEAAN chooses p^L for Q, where L is the multiplicative depth, the number of consecutive HE Mul operations applicable to a ciphertext before it loses encrypted data, and p is the rescaling factor. The message size in a ciphertext increases exponentially as we multiply the ciphertext repeatedly. To prevent an explosion of the message size, HEAAN performs rescaling after each HE Mul operation by dividing the coefficients of the output ciphertext by p. The size of q, the ciphertext modulus, is then adjusted to q', where $q' = q/p$. Therefore, $\log q$ of a just encrypted ciphertext starts at $\log Q$, decreases by $\log p$ upon each HE Mul, and becomes 0 after experiencing L HE Mul operations, losing data. When p is fixed, more HE Mul operations can be applied to a ciphertext with a larger Q value.

TABLE 2. Multiplicative depth (L) and required N over log$Q$ to guarantee 80-bit security level (an attacker needs $2^{80}\times$ of operations in polynomial ring with the current best algorithm).

$\log Q$	Multiplicative depth (L)	required N
300	10	2^{14}
600	20	2^{15}
1,200	40	2^{16}
2,400	80	2^{17}

To apply more HE Mul operations to a ciphertext, FHE re-initializes the ciphertext through bootstrapping. However, bootstrapping is costly (reportedly takes minutes [17]) because it consists of dozens of HE Mul and shift operations, reinforcing the importance of accelerating HE Mul. To reduce the overhead of bootstrapping for the practical use of HE, it is necessary to use large Q values to increase L. Moreover, N should increase as Q increases to guarantee a certain level of security in HE (see Table 2). As larger Q and N values require more computation and incur higher data storage costs per HE Mul operation, it is not efficient to use too large a value of Q; we discuss in further detail this trade off in Section VIII. We use (p, L, Q, N) with corresponding values of $(2^{30}, 40, 2^{1200}, 2^{16})$, respectively, the default values of the official HEAAN repository [1].

B. HEAAN COMPUTATION

Arithmetic operations in HEAAN, HE Add and Mul, are conducted through computation between the polynomials of operand ciphertexts. Here we assume that the two operand ciphertexts in a HE operation have the same ciphertext modulus value q.

HE Add computes an output ciphertext ($c3$) from two input ciphertexts ($c1$ and $c2$) through the following operations:

$$c3.ax = \text{mod}(c1.ax + c2.ax, q)$$
$$c3.bx = \text{mod}(c1.bx + c2.bx, q)$$

HE Add is relatively simple because it performs the element-wise addition of BigInt coefficients and then the modulo of q for each output coefficient; mod(x, y) means x modulo y.

HE Mult consists of two steps (see Figure 2). We compute a tensor product first:

$$c3.ax = \text{mod}(c1.ax \cdot c2.ax, q)$$
$$c3.bx = \text{mod}(c1.ax \cdot c2.bx + c1.bx \cdot c2.ax, q)$$
$$c3.cx = \text{mod}(c1.bx \cdot c2.bx, q)$$

Then, we update the output ciphertext with the evaluation key (evk), which is the encryption of a square of sk multiplied by Q:

$$c3.ax = \text{mod}(c3.ax + 1/Q \cdot c3.cx \cdot evk.ax, q)$$
$$c3.bx = \text{mod}(c3.bx + 1/Q \cdot c3.cx \cdot evk.bx, q)$$

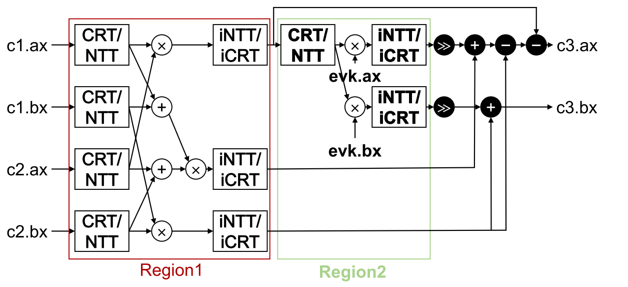


FIGURE 2. The overall flow of HE Mul in HEAAN. A white (black) filled symbol represents an operation conducted in an RNS (BigInt) domain. Region 1 and region 2 use different moduli because the former multiplies two log$q$-bit numbers whereas the latter multiplies a log$q$-bit number with an evaluation key polynomial composed of logQ^2-bit numbers.

This second step, called *relinearization*, enables the output ciphertext to continue to be representable with a polynomial pair.

HE Mul is much more costly than HE Add, as polynomial mul requires the multiplication of BigInt coefficients by N^2 times. We assume that β is the size of the integer data type (called a word size) that a computer natively supports with high performance (e.g., $\beta = 2^{64}$ for 64-bit CPUs). A log$q$-bit BigInt is represented as $qLimbs(= \lceil \log q / \log \beta \rceil)$ words. Then, one BigInt mul consists of $(qLimbs)^2$ logβ-bit word mul and $2(qLimbs)^2 - 1$ word addition (simply add) operations. For example, because $qLimbs$ is 19 when using the representative parameters ($N = 2^{16}$, $\log q = 1,200$, $\log\beta = 64$), 361 64-bit mul and 721 64-bit add operations are required in addition to carry propagation per BigInt mul. As described above, polynomial mul requires N^2 (= 4.3 billion) BigInt mul, which requires at least 4.6 Tera 64-bit operations. To reduce the complexity of this polynomial mul, HEAAN and other HE schemes use the *Chinese Remainder Theorem* (CRT [27]) and *Number Theoretic Transform* (NTT [23]).

1) CRT TO REDUCE THE COMPLEXITY OF BIGINT MUL

CRT states that for m coprime integers $\{m_i | 0 \le i < m\}$, the residue set $\{x_i = \text{mod}(X, m_i) | 0 \le i < m\}$ of any integer $0 \le X < \prod_{i=0}^{m-1} m_i$ is unique. HEAAN exploits CRT by defining a set of m integers $\{p_i | 0 \le i < m\}$, where each modulus p_i is a prime number smaller than β and $\prod_{i=0}^{m-1} p_i = P \ge q^2$. Then, a log$q$-bit BigInt number B, which is the coefficient of the ciphertext polynomial, can be represented in the residue number system (RNS) by the set of remainders $\{b_0, b_1, \ldots, b_{m-1}\}$ where $b_i = \text{mod}(B, p_i)$.

A key property of RNS is that when adding, subtracting, and multiplying numbers represented in RNS, it is sufficient to perform the same modular operation on each residue pair (a *congruence relation*). For a pair of log$q$-bit BigInt numbers (A, B) and their corresponding RNS representations ($\{a_i | 0 \le i < m\}$, $\{b_i | 0 \le i < m\}$), the product of A and B is C represented by $\{c_i | 0 \le i < m\}$ such that $c_i = \text{mod}(a_i \cdot b_i, p_i)$. This relation holds because we set $P \ge q^2$ and because the product of two log$q$-bit BigInt numbers is smaller than q^2.

TABLE 3. Representative parameters in homomorphic encryption (HE).

Symbol	Description	Representative values
Δ	**Scaling factor** is multiplied to the floating-point number of message to convert it to integer number.	$\log \Delta = 30$
p	**Rescaling factor** linearly reduces the size of messages that grow exponentially during computation.	$\log p = 30$
L	**Multiplicative depth** is the maximum number of possible mul on a ciphertext without bootstrapping.	40
Q	**The maximum ciphertext modulus** is equal to the initial ciphertext modulus after encryption.	$\log Q = 1200$
q	**Ciphertext modulus** starts from Q and is divided by a rescaling factor p at each mul.	$\log q = 1200, 1170, \ldots, 0$
N	**The number of coefficients of ciphertext polynomial**. The degree of ciphertext polynomial is N-1.	$2^{16} = 65536$
n	**The number of messages**. n messages are encrypted in one ciphertext.	32, 64,..., 32,768
β	**Word size**. It is machine-dependent (2^{64} for CPU, 2^{32} for GPU).	2^{64} or 2^{32}
$qLimbs$	**The number of limbs of** q. To represent $\log q$-bit integer, $qLimbs$ limbs are required.	$\lceil 1200/64 \rceil = 19$
m	**The number of prime numbers**. We use m prime numbers to represent big integer in RNS domain.	$\lceil 2400/58 \rceil = 42$
P	**Product of prime numbers** that are used to represent big integer.	$P = \prod_j p_j$
$PLimbs$	**The number of limbs of** P. To represent $\log(P/p_j)$-bit integer, $PLimbs$ limbs are used.	$\max_j(\log(P/p_j)/\log\beta)$

Therefore, a log$q$-bit BigInt number is converted into m logβ-bit data in the RNS domain, where we refer to the conversion as *a CRT function* or simply as `CRT`, and a BigInt mul is changed to m logβ-bit modular mul; hence, the time complexity per BigInt mul is changed from $\mathcal{O}(qLimbs^2)$ to $\mathcal{O}(m)$. In general, $qLimbs^2 \gg m$ (see Table 3), meaning that the number of operations required for BigInt mul can be greatly reduced by `CRT`. However, multiplying two BigInt polynomials still has complexity of $\mathcal{O}(m \cdot N^2)$ because polynomial mul requires N^2 coefficient mul operations.

2) NTT TO REDUCE THE COMPLEXITY OF POLYNOMIAL MUL

NTT is a discrete Fourier transform over a finite field (integer). Fast polynomial mul can be implemented with the Fast Fourier Transform (FFT) [12]. We can translate the polynomial mul with $\mathcal{O}(N^2)$ complexity into the element-wise mul with $\mathcal{O}(N)$ complexity with fast NTT, a variant of FFT that is limited to integer values. Although fast NTT (simply `NTT`) incurs a transformation cost of $\mathcal{O}(N \log N)$, it is beneficial to use `NTT` when N is large enough.

Figure 2 depicts the overall flow of HE Mul in HEAAN, which includes regions 1 and 2. Region 1 multiplies and adds the polynomials of the input ciphertexts, while region 2 conducts relinearization. In total, HE Mul consists of five polynomial mul operations, each performing (1) `CRT`, (2) `NTT`, and (3) element-wise modular mul operations, followed by (4) inverse fast NTT (`iNTT`) and (5) RNS-to-BigInt conversion (`iCRT`) to return to the polynomial with BigInt coefficients.

Region 1 configures m to deal with logq^2-bit BigInt, the intermediate result of polynomial mul between two input ciphertexts, whose coefficients are log$q$-bit long. Region 2 sets m to be larger to represent (log$q$+logQ^2)-bit BigInt as the coefficient size of *evk* is logQ^2-bit long. The shift operations in region 2 ($\gg$) reduce the amount of error accumulated during mul. The following adds and subtractions among the results of polynomial mul produce the result of HE Mul ($c3.ax$ and $c3.bx$); we can obtain an approximate value of mul between two original messages by decrypting this result using *sk*.

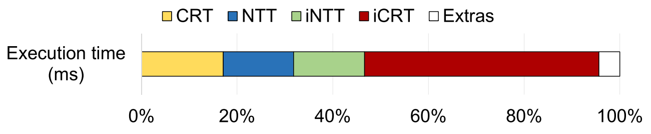


FIGURE 3. HE Mul time breakdown (total 3,907 ms).

Figure 3 shows the execution time breakdown of HE Mul using a single-threaded reference HEAAN in the system and configuration described in Section VI. `CRT`, `NTT`, `iNTT`, and `iCRT` account for 95.8% of the total execution time. The remaining operations, such as element-wise modular mul, account for only 4.2%. The total execution time is 3,907 ms, approximately 36,000× slower than the original message mul. *Therefore, accelerating HE Mul is essential to realize the practical use of HE, and it is necessary to accelerate* `CRT`, `NTT`, `iNTT`, *and* `iCRT`.

IV. AN IN-DEPTH ANALYSIS OF MAJOR FUNCTIONS IN THE HEAAN MULTIPLICATION OPERATION

To accelerate the primary functions (`CRT`, `NTT`, `iNTT`, and `iCRT`) in HE Mul, first we conduct an in-depth analysis of how each function works. In the following descriptions, $\mathbf{IN/OUT_{function}}(X, Y)$ represents an X by Y matrix used as

Algorithm 1 CRT

```
Require: $\mathbf{IN_{CRT}}(N, qLimbs)$, $\mathbf{TB_{CRT}}(m, qLimbs)$
Ensure: $\mathbf{OUT_{CRT}}(N, m)$
1: for ($i = 0;\ i < N;\ i \leftarrow i + 1$) do
2:     for ($j = 0;\ j < m;\ j \leftarrow j + 1$) do
3:         $\mathbf{accum} \leftarrow 0$
4:         for ($k = 0;\ k < qLimbs;\ k \leftarrow k + 1$) do
5:             $\mathbf{accum} \mathrel{+}= \mathbf{IN_{CRT}}[i][k] \times \mathbf{TB_{CRT}}[j][k]$
6:         $\mathbf{OUT_{CRT}}[i][j] = \mathrm{mod}(\mathbf{accum}, p_j)$
```

Algorithm 2 Shoup's Modular Multiplication (ModMul)

```
Require: $X, Y, p_j, Y_{Shoup}$
Ensure: $r = \mathrm{mod}(X \times Y, p_j)$
1: $Qu_{hi} = (X \times Y_{Shoup}) \gg log\beta$
2: $r = X \times Y - Qu_{hi} \times p_j$
3: if $r > p_j$ then
4:     $r = r - p_j$
```

the input/output of a function, while $\mathbf{TB_{function}}(X, Y)$ represents a precomputed table of the X by Y matrix.

`CRT` (Algo. 1) takes $\mathbf{IN_{CRT}}(N, qLimbs)$ representing N $\log q$-bit BigInt numbers and produces $\mathbf{OUT_{CRT}}(N, m)$, the result of a modulo operation on each BigInt with m different primes $\{p_j | 0 \le j < m\}$. The operation consists of two stages: (1) computing the matrix-matrix mul of $\mathbf{IN_{CRT}}$ with $\mathbf{TB_{CRT}}^\intercal$ and (2) applying modulo operations to each output element.

A BigInt A is expressed by $qLimbs$ $\log\beta$-bit words; i.e., $\sum_{k=0}^{qLimbs-1} a_k \cdot \beta^k$, where $\{a_k | 0 \le k < qLimbs\}$. The modulo operation on the BigInt is then as follows:

$$\begin{aligned}\mathrm{mod}(A, p_j) &= \mathrm{mod}(\textstyle\sum_{k=0}^{qLimbs-1} a_k \cdot \beta^k, p_j)\\ &= \mathrm{mod}(\textstyle\sum_{k=0}^{qLimbs-1} a_k \cdot \mathrm{mod}(\beta^k, p_j), p_j)\end{aligned}$$

Here because β and p_j are independent of the input, HEAAN precomputes $\mathbf{TB_{CRT}}$, $\mathrm{mod}(\beta^k, p_j)$ for all k and j. Therefore, $\sum_{k=0}^{qLimbs-1} a_k \cdot \mathrm{mod}(\beta^k, p_j)$ is performed by multiplying $\mathbf{IN_{CRT}}$ and $\mathbf{TB_{CRT}}$.

We can exploit Shoup's modular mul (Shoup's ModMul [64]) for the modulo operation in line 6 of Algo. 1. Shoup's ModMul (Algo. 2) computes $\mathrm{mod}(X \cdot Y, p_j)$ with three muls and a single correction step if the value of Y_{Shoup} $(= \lfloor \frac{Y \cdot \beta}{p_j} \rfloor)$ is known in advance. This replaces a costly division operation with relatively inexpensive mul, comparison, and subtraction operations. We apply the algorithm for the modular mul on *accum* spanning up to three limbs ($accum_0 + accum_1 \cdot \beta + accum_2 \cdot \beta^2$) using precomputed Y_{Shoup} values of $Y = \{1, \beta, \beta^2\}$. The operations of `CRT` can be performed in parallel for each coefficient (total N) and each prime number (total m).

`NTT` implements the Cooley-Tukey algorithm [23], which recursively divides an N-point FFT into n N/n-point FFTs and combines their results (called *radix-n* FFT). An exemplar radix-2 NTT in Algo. 3 takes the matrix $\mathbf{IN_{NTT}}(m, N$ as an input and runs the butterfly algorithm `butt`. It uses a precomputed table ($\mathbf{TB_W}$) of powers of the $2N$-th root of unity for all m prime numbers. For each prime, `butt` (Algo. 4) is called $\log N \cdot \frac{N}{2}$ times. As `butt` requires modular mul, it also uses Shoup's ModMul, as was done in `CRT`.

Algorithm 3 NTT

```
Require: $\mathbf{IN_{NTT}}(m, N) \leftarrow \mathbf{OUT_{CRT}}^\intercal$, $\mathbf{TB_W}(m, N)$
Ensure: $\mathbf{OUT_{NTT}}(m, N)$
1: for ($i = 0;\ i < m;\ i \leftarrow i + 1$) do
2:     $t = N$
3:     $\mathbf{IN} = \mathbf{IN_{NTT}}[i]$
4:     $\mathbf{TBW} = \mathbf{TB_W}[i]$
5:     for ($j = 1;\ j < N;\ j \leftarrow j \times 2$) do
6:         $t = t\ /\ 2$
7:         for ($k = 0;\ k < j;\ k \leftarrow k + 1$) do
8:             for ($l = k \times 2t;\ l < k \times 2t + t;\ l \leftarrow l + 1$) do
9:                 $butt(\mathbf{IN}[l], \mathbf{IN}[l + t], p_i, \mathbf{TBW}[j + k])$
10:    $\mathbf{OUT_{NTT}}[i] = \mathbf{IN}$
```

Algorithm 4 Butt

```
Require: $A, B, p_i, W$
Ensure: $A, B$
1: $U = \mathrm{mod}(B \times W, p_i)$
2: $B = A - U$
3: $A = A + U$
```

Algorithm 5 iCRT

```
Require: $\mathbf{IN_{iCRT}}(m, N) \leftarrow \mathbf{OUT_{iNTT}}(m, N)$
    $\mathbf{TB_{invP}}(m)$, $\mathbf{TB_{Pdivp}}(m, PLimbs)$
Ensure: $\mathbf{OUT_{iCRT}}(N, m)$
1: for ($i = 0;\ i < N;\ i \leftarrow i + 1$) do
2:     for ($j = 0;\ j < m;\ j \leftarrow j + 1$) do
3:         $\mathbf{temp}[j][i] = \mathrm{mod}(\mathbf{IN_{iCRT}}[j][i] \times \mathbf{TB_{invP}}[j], p_j)$
4: for ($i = 0;\ i < N;\ i \leftarrow i + 1$) do
5:     $\mathbf{accum} = 0$
6:     for ($j = 0;\ j < m;\ j \leftarrow j + 1$) do
7:         for ($k = 0;\ k < PLimbs;\ k \leftarrow k + 1$) do
8:             $\mathbf{accum} += \mathbf{temp}[j][i] \times \mathbf{TB_{Pdivp}}[j][k] \times \beta^k$
9:     $\mathbf{OUT_{iCRT}}[i] = \mathrm{mod}(\mathrm{mod}(\mathbf{accum}, \mathbf{P}), \mathbf{q})$
```

`iNTT` is slightly different from `NTT`. It has a different loop order, calls inverse butterfly (`ibutt`) instead of `butt`, deals with a different precomputed table (consisting of the inverse of the powers of the primitive root of unity ($\mathbf{TB_{invW}}$)), and finally divides each element by N. However, except for the last element-wise division by N, `iNTT` is symmetric to `NTT` in terms of the numbers and types of operations. Both `NTT` and `iNTT` are completely parallelizable for each prime number.

`iCRT` converts the matrix $\mathbf{OUT_{iNTT}}(m, N)$, where each element is a remainder smaller than β, back to N $\log q$-bit BigInts (see Algo. 5). It starts with (1) the Hadamard product between an input matrix and a precomputed table $\mathbf{TB_{invP}}$, whose elements are modular inverses of P/p_j for all p_j, followed by an element-wise modular mul with each p_j. Shoup's ModMul can also be applied here for an efficient modular mul operation. (2) Then, each output element of (1), a scalar value, is multiplied by a BigInt P/p_j according to its j and accumulated to a temporary BigInt *accum*. Here, each instance of P/p_j is precomputed and stored in table $\mathbf{TB_{Pdivp}}(m, PLimbs)$, where $PLimbs = \max_j(\log(P/p_j)/\log\beta)$. (3) A reduction of *accum* modulo P and q ($q \cdot Q$ in region 2) is performed.

TABLE 4. The number of arithmetic operations and computational complexity of major functions of HE Mul.

	CRT	NTT	iNTT	iCRT
Multiplication	$N \times qLimbs \times m$	-	-	$N \times m \times PLimbs$
Modular mul	$N \times m$	$m \times N/2 \times \log N$	$m \times (N/2 \times \log N + N)$	$2 \times N \times m$
ADC (add with carry)	$N \times qLimbs \times m$	-	-	$N \times m \times PLimbs$
Add, Sub	-	$m \times N \times \log N$	$m \times N \times \log N$	-
Computation complexity	$\mathcal{O}(N \times qLimbs \times m)$	$\mathcal{O}(N \times logN \times m)$	$\mathcal{O}(N \times logN \times m)$	$\mathcal{O}(N \times m \times PLimbs)$

TABLE 5. Input and precomputed data size of major functions of HE Mul. The unit of data size is β.

	CRT	NTT & iNTT	iCRT
Input data	$N \cdot qLimbs$	$N \cdot m$	$N \cdot m$
Precomp. data	$m \cdot qLimbs$	$N \cdot m \cdot 2$	$m \cdot (PLimbs + 1)$

Algorithm 6 iCRT Algorithm in the Matrix-Matrix Mul Form. The First Three Lines Are the Same as Algo. 5.

```
   ...
4: for ($i = 0$; $i < N$; $i \leftarrow i + 1$) do
5:   accum $= 0$
6:   for ($k = 0$; $k < PLimbs$; $k \leftarrow k + 1$) do
7:     $\textbf{accum}_{\textbf{small}} = 0$
8:     for ($j = 0$; $j < m$; $j \leftarrow j + 1$) do
9:       $\textbf{accum}_{\textbf{small}} += \textbf{temp}[j][i] \times \textbf{TB}_{\textbf{Pdivp}}[j][k]$
10:    accum $=$ accum $+ \textbf{accum}_{\textbf{small}} \times \beta^k$
11:  $\textbf{OUT}_{\textbf{iCRT}}[i] = \text{mod}(\text{mod}(\textbf{accum}, \textbf{P}), \textbf{q})$
```

We summarize the number of arithmetic operations needed for each major function in Table 4 and the size of input and precomputed data for each function in Table 5.

V. ARCHITECTURE-CENTRIC OPTIMIZATIONS TO MAXIMIZE HE MUL PERFORMANCE ON CPUs AND GPUs

Previous HE studies [61], [64] sought to propose new hardware architectures (e.g., through FPGA implementation) for performance improvements. In contrast, we initially improve the performance of HE by utilizing the most popular computation platforms, CPUs and GPUs, which are already equipped with hundreds to thousands of ALUs.

All of the major functions of HE Mul have massive parallelism that can be exploited by CPUs and GPUs. All residual numbers ($N \times m$) can be computed in parallel on CRT. NTT and iNTT perform m independent transformations and leverage the algorithmic optimization of FFT, where $N/2$ pairs can be computed in parallel at each individual stage during FFT. Henceforth, we identify the key challenges and solutions we devise when accelerating HE Mul on CPUs/GPUs.

A. LOOP REORDERING TO EXPOSE MASSIVE PARALLELISM IN iCRT

iCRT recombines the residual numbers into integers of size $\log q$ for each coefficient of the resulting ciphertext; hence, it may be regarded that the degree of parallelism is smaller than CRT (N vs. $N \cdot m$). However, the limited N-degree parallelism can be expanded to $N \cdot PLimbs$-degree parallelism by reordering two loops in iCRT (lines 6 and 7 in Algo. 5; see the modified algorithm in Algo. 6). After reordering, the sequence of the original mul between a scalar and a BigInt becomes a matrix-matrix mul between a *temp* matrix and a $\textbf{TB}_{\textbf{Pdivp}}$ matrix. Then, iCRT should be modified such that the partial sum in the inner-most loop is accumulated into $accum_{small}$ (double or triple word), rather than *accum* (BigInt), which is aggregated to *accum* at the end of the loop. With our loop reordering, the resulting matrix-matrix mul exposes a massive parallelism of degree $N \cdot PLimbs$ in iCRT, providing abundant parallelization opportunities to contemporary hardware platforms.

B. ACCELERATING HE MUL ON CPUs

The strategies that a modern CPU utilizes to exploit parallelism from an application are twofold: populating (1) multiple cores and (2) ALUs supporting short-SIMD instructions in each core. For example, the Intel Xeon CPU we use has 24 cores per socket, with each core supporting AVX-512 instructions capable of executing eight 64-bit (64b) integer operations [29]. We take advantage of the innate parallelism in the major functions of HE Mul by initially distributing operations to multiple threads and then to AVX-512 SIMD lanes in each thread while minimizing the performance drop by frequent cache ($) misses from poor data access patterns; during this process, whether an input matrix follows a column- or a row-major order can vastly affect the performance.

Figure 4 shows the output of the four major functions and how we parallelize them to multiple threads and then to AVX-512 lanes. During CRT, a CPU thread takes responsibility for a portion of the N coefficients (line 1 in Algo. 1), whereas each lane of an AVX-512 port performs operations on different prime numbers (line 2 in Algo. 1). During NTT/iNTT, a thread does its job on a portion of the prime numbers, (line 1 in Algo. 3), whereas each lane of AVX-512 computes some of the coefficients (line 8 in Algo. 3). In the iCRT case, we take different approaches for the two iteration phases. During phase 1 (lines 1-3 in Algo. 5), each thread and an AVX-512 lane undertake computation on some of the N coefficients (both on line 1). During phase 2 (lines 4-11 in Algo. 6), each thread also computes some of the coefficients (line 4), but

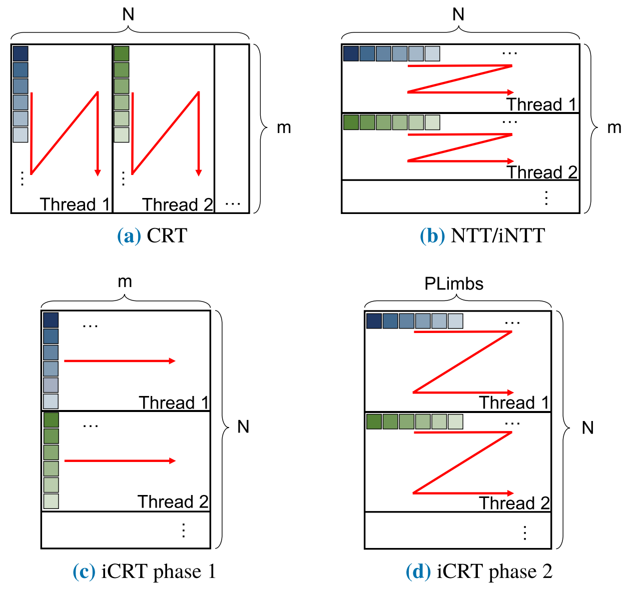


FIGURE 4. Output of the major functions and their processing order in our CPU implementation. Data are represented in row major order. Each colored square represents an AVX-512 lane of a thread producing an output in the inner-most loops. Red arrows represent the order of loop iteration.

each lane is on a different k, the positional index on the limbs of P/p_j (line 6).

The reference HEAAN [1] also uses multi-threading but without SIMD. It relies on a different strategy in `CRT` from ours; each thread operates on different prime numbers (resulting in m-degree parallelism).

1) MATRIX TRANSPOSITION IN `iCRT`

SIMD instructions can lead to a poor \$ utilization if they access a matrix storing elements in a row-major order according to the column direction (or vice versa), as this demands multiple \$ lines at once. The resulting performance degradation is more prominent when the access stride is too large for a hardware prefetcher to be effective. `iCRT` experiences this issue because the matrix-matrix mul (line 9 in Algo. 6) accesses the *temp* matrix that stores the elements in a row-major order according to the column direction. We implicitly transpose the *temp* matrix using the *scatter* instructions in AVX-512 to address this issue.

2) EMULATING ARITHMETIC OPERATIONS

AVX-512 does not support parallel 64b mul or 64b ADC (addition with carry) yet. Therefore, 64b mul is emulated with four parallel 32b mul, five 64b add, and five 64b shift instructions. Also, one 64b compare and one additional 64b add are required to handle the carry operation per addition. This emulation narrows the performance gap between the AVX-512 implementation and the reference HEAAN (no AVX-512). To improve the performance further under this constraint, we modify Shoup's ModMul as follows.

The original Shoup's ModMul requires three operations: one 64b mulhi to compute Qu_{hi} and two 64b mullo to compute $Qu_{hi} \cdot p_j$ and $X \cdot Y$, where Qu is the estimation of the quotient, and 64b mulhi (mullo) returns the upper (lower) 64b of the mul result, which is 128b long. A single 64b mulhi can be emulated with four 32b mul ($hi{\cdot}hi$, $hi{\cdot}lo$, $lo{\cdot}hi$, $lo{\cdot}lo$), four 64b add, and five 64b shift operations. In this case, the estimated remainder lies in the range of $[0, 2p_j)$.

However, one of the 32b mul operations ($lo \cdot lo$) for emulating 64b mulhi is used only for computing a carry from the low 64b of mul. We can remove this $lo \cdot lo$ mul if the carry is ignored and produce an approximated 64b mulhi ($Qu_{hi'}$) with only three muls instead of four. By applying this optimization, the estimated remainder lies in the range of $[0, 4p_j)$. As the upper bound of a remainder grows to $4p_j$, one additional correction step (conditional subtraction) is needed, but the number of more expensive operations is reduced to one 32b mul, two 64b add, and one 64b shift instruction.

3) REDUCING THE SIZE OF β TO 2^{32}

We can remove the overhead of emulating 64b operations by using β of 2^{32} instead of 2^{64}, as AVX-512 naturally supports 32b mul and ADC. Using the smaller β changes the number of instructions for mul and ADC to one, and reduces the number of instructions for modular mul to less than half. However, this strategy also has shortcomings in that *qLimbs* should roughly be doubled to express numbers whose sizes are up to q. m must also be doubled because the upper bound of each prime number is β. Larger *qLimbs* and m imply larger precomputed tables and more iterations for `CRT`, `NTT`, `iNTT`, and `iCRT`. It also increases the number of operations other than mul, modular mul, and ADC. A preliminary implementation shows no improvement in performance when using a small β for CPUs.

C. ACCELERATING HE MUL ON GPUs

Modern GPUs (e.g., Volta [54] and Turing [55]) have as many integer (INT32) units as single-precision floating-point (FP32) units, resulting in multiply-accumulate throughput for INT32 numbers identical to that for FP32 numbers. Such massive throughput of integer operations makes the GPU an attractive candidate for accelerating HE operations.

1) PARALLELIZATION STRATEGIES

The CUDA programming model [56] for GPUs has the following hierarchical structure of threads: multiple GPU threads are grouped to form a *thread block* and multiple thread blocks comprise a *grid*. A thread block is allocated to one Streaming Multiprocessor (SM). The threads in a thread block share the resources (e.g., shared memory) of the SM. Each thread block in a grid is allocated to each SM in a round-robin fashion, and the number of thread blocks (*grid dimension*) and number of threads in a block (*block dimension*) are configured at each GPU kernel launch.

Algorithm 7 Pseudocode of Baseline iCRT With CUDA

```
Require: $\mathbf{IN_{iCRT}}(m, N)$, $\mathbf{TB_{invP}}(m)$, $\mathbf{TB_{Pdivp}}(m, PLimbs)$
Ensure: $\mathbf{OUT_{iCRT}}(N, m)$
 1: Launch $N$ threads. Each thread performs the below steps.
 2: $i = threadIdx.x + blockDim.x \cdot blockIdx.x$
 3: accum = uint32_t[MAX_NUM_PRIMES_TO_USE]
 4: for $j \in [0, m)$ do
 5:     $temp = (\mathbf{IN_{iCRT}}[j][i] \cdot \mathbf{TB_{invP}}[j])$ % $p_j$
 6:     // accumulate low parts
 7:     mad.lo.cc.u32 accum[0], $temp$, $\mathbf{TB_{Pdivp}}[j][0]$, accum[0]
 8:     for $k \in [1, PLimbs)$ do
 9:         madc.lo.cc.u32 accum[$k$], $temp$, $\mathbf{TB_{Pdivp}}[j][k]$, accum[$k$]
10:     // accumulate high parts
11:     mad.hi.cc.u32 accum[1], $temp$, $\mathbf{TB_{Pdivp}}[j][0]$, accum[1]
12:     for $k \in [1, PLimbs)$ do
13:         $k' = k + 1$
14:         madc.hi.cc.u32 accum[$k'$], $temp$, $\mathbf{TB_{Pdivp}}[j][k]$, accum[$k'$]
15: Do accum % $P$ by subtracting $P \cdot \frac{m}{2}, P \cdot \frac{m}{4}, \ldots, P$ conditionally.
16: for $j \in [0, m)$ do
17:     $\mathbf{OUT_{iCRT}}[i][j]$ = accum[$j$]
```

Algorithm 8 Pseudocode of iCRT in Algo. 6 With CUDA

```
Require: $\mathbf{IN_{iCRT}}(m, N)$, $\mathbf{TB_{invP}}(m)$, $\mathbf{TB_{Pdivp}}(m, PLimbs)$
Ensure: $\mathbf{OUT_{iCRT}}(N, m)$
 1: Launch $N \cdot m$ threads.                                   // the 1st kernel
 2: temp[$j$][$i$] = $\mathbf{IN_{iCRT}}[j][i] \cdot \mathbf{TB_{invP}}[j]$ % $p_j$ for each $i, j$.
 3: Launch $N \cdot PLimbs$ threads.                              // the 2nd kernel
 4: $gid = threadIdx.x + blockDim.x \cdot blockIdx.x$
 5: $i = gid$ / $PLimbs$
 6: $k = gid$ % $PLimbs$
 7: $(carry, hi, lo) = (0, 0, 0)$
 8: for $j \in [0, m)$ do
 9:     $in$ = temp[$j$][$i$]
10:     $tb = \mathbf{TB_{Pdivp}}[j][k]$
11:     mad.lo.cc.u32 $lo, in, tb, lo$
12:     madc.hi.cc.u32 $hi, in, tb, hi$
13:     addc.cc.u32 $carry, carry$, 0
14: $\mathbf{accum}_{lo}[i][k] = lo$
15: $\mathbf{accum}_{hi}[i][k] = hi$
16: $\mathbf{accum}_{carry}[i][k] = carry$
17: Launch $N$ threads.                                           // the 3rd kernel
18: Merge $\mathbf{accum}_*$ arrays into accum.
19: Do line 15-17 in Algo. 7.
```

The basic parallelization strategy is to assign each independently computable output element to a thread. We launch $N \cdot m$ threads for CRT so that each thread computes one output element (a residue). NTT and iNTT launch $N/2 \cdot m$ threads each, where each thread performs a butterfly operation (Algo. 4) per butterfly step using a simple radix-2 iterative NTT algorithm [40].

In the iCRT case, a naïve parallelism strategy uses N-degree parallelism, where one thread takes charge of producing one output BigInt type of coefficient. Prior studies [10], [25] utilized the same strategy. Algo. 7 shows the baseline iCRT implementation with CUDA based on the results of prior studies. However, by changing the loop order as described above, we transform the corresponding core operation into a matrix-matrix mul operation, thereby taking advantage of $N \cdot PLimbs$-degree parallelism to maximize thread-level parallelism (see line 3-6 in the modified implementation in Algo. 8).

Algorithm 9 Pseudocode of CRT With CUDA

```
Require: $\mathbf{IN_{CRT}}(N, qLimbs)$, $\mathbf{TB_{CRT}}(m, qLimbs)$
Ensure: $\mathbf{OUT_{CRT}}(N, m)$
 1: Launch $N \cdot m$ threads. Each thread performs the below steps.
 2: $gid = threadIdx.x + blockIdx.x \cdot blockDim.x$
 3: $i = gid$ / $m$
 4: $j = gid$ % $m$
 5: $(carry, hi, lo) = (0, 0, 0)$                            // accum in Algo. 1
 6: for $k \in [0, qLimbs)$ do
 7:     // perform $(carry, hi, lo)$ += $\mathbf{IN_{CRT}}[i][k] \cdot \mathbf{TB_{CRT}}[j][k]$
 8:     $in = \mathbf{IN_{CRT}}[i][k]$
 9:     $tb = \mathbf{TB_{CRT}}[j][k]$
10:     mad.lo.cc.u32 $lo, in, tb, lo$
11:     mad.hi.cc.u32 $hi, in, tb, hi$
12:     addc.cc.u32 $carries, carries$, 0
13: $temp = (carries \ll 32) \mid hi$ % $p_j$                 // 64b modulo a 32b
14: $temp = (temp \ll 32) \mid lo$ % $p_j$
15: $\mathbf{OUT_{CRT}}[j][i] = temp$
```

2) 64B EMULATION VS. 32B WORDS

As opposed to the most CPUs that natively support 64b words, modern GPUs natively support 32b words and emulate 64b integer operations. To avoid the overhead of 64b emulation (whose throughput is more than one order of magnitude lower than that of the 32b counterpart), prior studies accelerating HE on GPUs [10], [25] use 32b words ($\beta = 2^{32}$) and the prime numbers smaller than β. We also use 32b words and operations. Another advantage of using 32b words on GPU is that the operations with carry-in and carry-out processes can be executed without emulation; modern NVIDIA GPUs support carry operations (e.g., addc, subc, and madc in assembly-like virtual ISA, PTX [57], where the operations are called *extended-precision integer arithmetic instructions*). The throughput of these instructions is identical to that of those without a carry in recent GPUs [54], [55], enabling efficient computations of large integers without emulation.

3) DIFFERENT STRATEGIES FOR BIGINT MODULO IN CRT

A naïve BigInt modulo is done by repetitive $\log\beta$-bit shift, add, and modulo operations; for example, cuHE [25], takes this approach. In contrast, HEAAN accumulates the result of modulo on each $a_k \cdot \beta^k$ using a precomputed table in CRT. In this case, the BigInt variable *accum* (line 3 in Algo. 1) can span two or three words depending on m and the size of each prime number. In the CPU implementation case, *accum* is guaranteed to span two words when using the representative parameters specified in Table 6, as an overflow does not occur for $m \leq 2^{64-58} = 64$ with prime numbers smaller than 2^{58}. To guarantee that *accum* is two words long, we use 2^{57} as a lower bound of the prime for AVX-512 implementation instead of 2^{59}, the default value of the reference HEAAN. However, on a GPU with a 32b β, with primes smaller than 2^{30}, only up to four ($= 2^{32-30}$) accumulations are allowed to guarantee that the overflow issue does not arise, which is nearly impossible as m is 90 or higher when using the representative parameters.

To prevent an overflow, one may (1) use a three-word *accum* with an additional ADC operation included in the inner-most loop (see line 12 of the CRT implementation shown in Algo. 9) to avoid expensive modulo operations, or (2) may conduct modulo operations intermittently in the inner-most loop (e.g., for every four accumulations in our case) to ensure that *accum* spans only two words [9]. We implemented these two strategies and found that using a three-word *accum* performs better as it requires fewer modulo operations.

4) PER-THREAD STORAGE FOR ACCUMULATION IN iCRT

The baseline iCRT with N-degree parallelism allocates a BigInt *accum* (line 8 in Algo. 5) as a long array in a per-thread manner. If *accum* is not carefully allocated to fast storage, frequent $ thrashing may occur, degrading the performance significantly. The latest NVIDIA GPUs [54], [55] have a variety of storage types (e.g., register, L1/L2 $, device memory, and read-only constant memory). In the original iCRT implementation (Algo. 7), $temp[j][i]$ in Algo. 5 is stored in register memory (*temp* in Algo. 7), so that it can be loaded quickly in a single cycle. On the other hand, because *accum* is declared as a thread-local array that is dynamically indexed in the algorithm (i.e., $accum[idx]_{0 \leq idx < PLimbs}$ where *PLimbs* is given as an input variable to the kernel), it is not stored in the register, the fastest storage on a GPU. Instead, the CUDA compiler stores it in global memory and caches it into L1 and L2 (CUDA calling this *local memory* [56]).

However, the heavy use of local memory can lead to $ thrashing when the grid and block dimensions increase, causing a number of threads to compete for $ and degrading the overall performance. To mitigate the $ miss penalty, we suggest two different optimizations when using N parallelism in iCRT: (1) using fewer threads by simply reducing the block and grid dimensions via the grid-stride loop method [39], or (2) pinning each *accum* array (line 3 in Algo. 7) in L1 $ by allocating the array in shared memory; this is possible as the shared memory shares capacity with the L1 unified $. We compare the methods on cuHE's iCRT kernel [25] in Algo. 7, which implements Algo. 5 with N parallelism, along with loop reordering with $N \cdot PLimbs$ parallelism in Algo. 8.

5) HIGH-RADIX NTT AND iNTT

For radix-2 NTT, GPU reads and writes a large input $\mathbf{IN_{NTT}}(m, N)$ (dozens of megabytes with typical m and N values specified in Table 3) by $\log_2 N$ times. At each butt in Algo. 3, a GPU thread reads two values of **IN** from device memory and writes two output values back to the device memory. By increasing the radix k, each thread reads and writes k values within **IN** for butt, performing k-point NTT. It changes the number of elements of **IN** to transfer from $\log_2 N$ above to $\log_k N$, reducing the number of main memory accesses needed for NTT. Because the device memory bandwidth is a scarce resource, this reduction typically translates to higher performance. However, increasing the radix is not always beneficial; as each thread takes more inputs, the register pressure on each thread increases. Using registers that exceed the register file size of an SM causes register spillover to local memory, degrading the performance due to additional data loads from the main memory. Given this constraint, we use a proper radix size (radix-32) to balance the pressure from the memory bandwidth and register spillover.

D. SCALABILITY STUDY

We also extend our implementations of HE Mul to support distributed-memory multi-GPU systems and study the corresponding scalabilities on CPUs and GPUs. For CPU implementation with multi-socket processors, we can use the same parallelization approach described in Section V-B and populate more cores. However, we take a different approach for the implementation that exploits multiple GPUs because they take a distributed memory architecture.

It is critical to determine how to distribute the work and data to multiple GPUs. We dedicate one polynomial to the memory of a single GPU and let the GPU operate on the polynomial (see Figure 5). As we dedicate each polynomial to a GPU and an HE Mul takes four polynomials as its inputs, there exist four *streams* in an HE Mul, each being a sequence of operations executed on a GPU. When performing unary operations, such as CRT, iCRT, NTT, and iNTT, a stream does not require data transfers or synchronization to another stream. A stream only synchronizes to another stream for a binary operation, waiting for one of its operands to be transferred from the other stream.

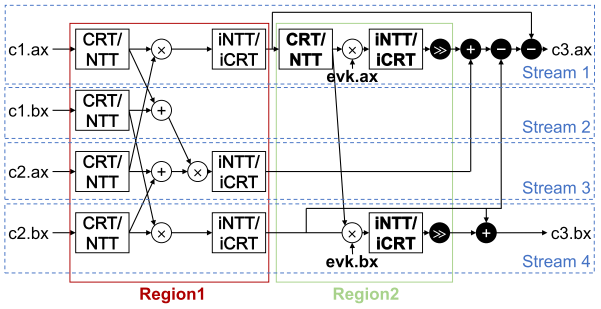


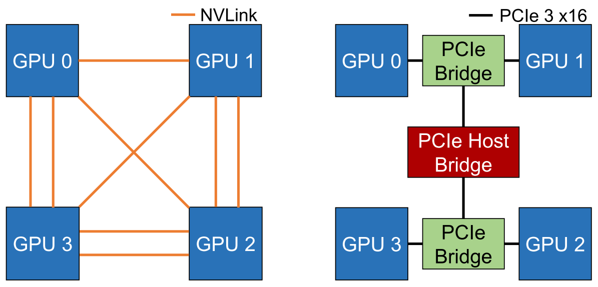
FIGURE 5. The workflow of multi-GPU HE Mul in HEAAN. Each stream is assigned to a GPU and is entirely executed on that GPU. Other details are the same with Figure 2.

Our technique can exploit up to four GPUs for a single HE Mul by assigning one stream to a GPU. If we utilize fewer than four GPUs, we assign more than one stream to the same GPU; for example, one should decide how to assign the four streams to the two GPUs appropriately when exploiting only two GPUs. To assign the streams to GPUs, we consider the ideal latency of an HE Mul, which is determined by the critical path in stream 1, the most computationally expensive stream. We assign stream 1 to one GPU and streams 2, 3, and 4 to the other; this arrangement performs best on the multi-GPU configurations used in Section VII.

VI. EXPERIMENTAL SETUP

We compared the performance of the reference HEAAN [1], our AVX-512 implementation with multi-threading, and GPU implementation. We used an Intel Xeon CPU (Cascade Lake-based Xeon Platinum 8260 operating at 2.4 GHz) and NVIDIA GPUs (Turing Titan RTX operating at 1.35 GHz for the single-GPU case, and Pascal Titan X and Volta V100 operating at 1.405 GHz and 1.245 GHz, respectively, for the multi-GPU cases). The CPU consists of 24 cores per socket and each core has two AVX-512 FMA units, achieving peak 64-bit integer performance of 1.84 TOPS per socket. Each core has a 32 KB L1I and L1D cache and a 1 MB L2 cache. A 35.75 MB L3 cache is shared by all the cores within a socket, and each socket has six memory channels, each equipped with DDR4-3200 DRAM modules. We did not use HyperThreading. In each setup, the Titan X, Titan RTX, and V100 GPUs consist of 28, 72, and 80 SMs, each with 128, 64, and 64 CUDA cores, performing up to 3,584, 4,608, and 5,120 32-bit integer operations per cycle, respectively. The corresponding sizes of each per-SM L1 cache are 48 KB, 64 KB, and 128 KB, and those of the per-GPU L2 shared caches are 3 MB, 6 MB, and 6 MB, respectively. All the hardware performance metrics of the GPU kernels were measured with NVIDIA's profiling tool (Nsight Compute [58] version 2021.1). For functions that include multiple kernels (e.g., `iCRT` in Algo. 8), we took the weighted average of the measurements using each kernel's execution time as the weight.

We also evaluated our multi-GPU implementation of HE Mul on two multi-GPU systems, one with four NVIDIA V100 GPUs and the other with four Titan X GPUs. Their network topologies and the corresponding peer-to-peer bandwidth measurements are shown in Figure 6. The number of orange lines between a pair of two GPUs indicates the number of NVLinks bonded to the two GPUs, which is proportional to the peer-to-peer bandwidth.



(a) V100 (b) Titan X (pascal)

FIGURE 6. The interconnect topologies of two different multi-GPU systems. (a) In V100, each NVLink offers the peer-to-peer bandwidth of 48.5 GB/s, whereas (b) in Titan X, PCIe offers 20.4 GB/s (26.1 GB/s) when passing (not passing) through the PCIe host bridge.

Table 6 includes the key parameters for HE Mul on a CPU and a GPU. We measured the execution time of HE Mul, excluding the time for memory operations, in this case malloc, free, and data transfers from host to the device for GPUs.[1] We conducted each experiment 32 times and reported the average.

TABLE 6. HEAAN parameter settings for CPU and GPU.

Parameters	CPU (AVX-512)	GPU
N	2^{16}	
Q & q	2^{1200}	
β & $qLimbs$	2^{64} & 19	2^{32} & 38
Prime number size	$2^{57} < p_i < 2^{60}$	$2^{27} < p_i < 2^{30}$
m (Region 1)	42 (43)	90
m (Region 2)	63 (64)	134

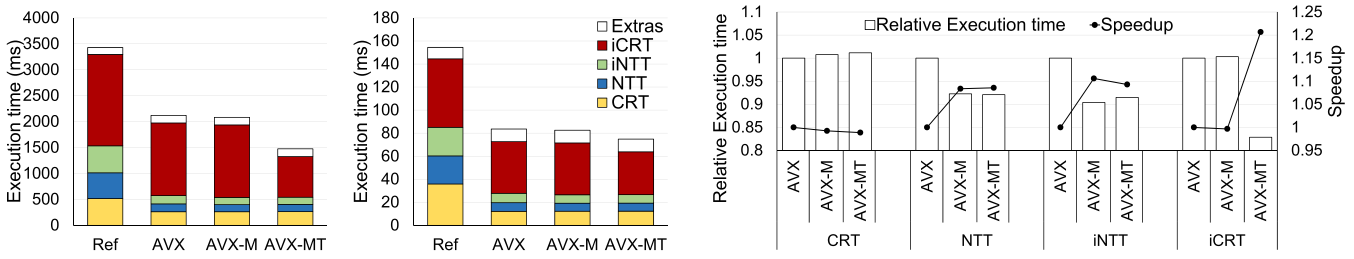
VII. EVALUATION

A. IMPLEMENTATIONS

We evaluated the effectiveness of the proposed optimizations for accelerating HEAAN mul in a comparison against the performance of the reference HEAAN (**Ref**). For the CPU case, we compared the basic implementation utilizing AVX-512 (**AVX**), the setup with the modified Shoup's ModMul on top of **AVX** (**AVX-M**), and the setup transposing the *temp* matrix on top of **AVX-M** (**AVX-MT**). In the basic GPU implementation (**GPU**), we adopted radix-2 NTT for `NTT` and `iNTT`. We modified the CRT kernel of cuHE [25], which only exploits N-degree parallelism, to exploit $N \cdot m$-degree parallelism. Also, we used the iCRT kernel of cuHE. We compared **GPU** with the following: the implementation optimizing `CRT` by using ADC instead of intermittently conducting modulo operations (**GPU-C**), that which adjusted the number of launching threads on top of **GPU-C** (**GPU-CT**), the implementation using shared memory to pin the arrays of each thread to the L1 unified $ on top of **GPU-C** (**GPU-CP**), the implementation applying loop reordering (Algo. 6) to translate a majority of the `iCRT` computations into matrix-matrix mul to use $N \cdot PLimbs$-degree parallelism on top of **GPU-C** (**GPU-CL**), and the implementation with high-radix `NTT` and `iNTT` to reduce main memory accesses and utilize the GPU's computing power more efficiently on top of **GPU-CL** (**GPU-CLH**).

We made the following key observations. *First, exploiting the massive parallelism supported by modern CPUs and GPUs results in an approximate performance improvement of 4.05× in HE Mul.* Table 7 shows the execution time and the relative speedup of the CPU and GPU implementations after applying a series of architecture-centric optimizations. **AVX-MT** and **GPU-CLH**, the implementations giving the best performance for the CPU and GPU cases, achieve 2.06× and 4.05× speedup, respectively, compared to the reference HEAAN (**Ref-24**). **GPU-CLH** performs 3.0× and 1.8× better than **AVX-MT** on `CRT` and `iCRT` owing to more ALUs

[1]In a single HE Mul with the parameters in Table 6, the cost of data transfers between CPU and GPU is around 40%. However, the overhead can be mostly amortized as GPU can reuse data during a series of HE Muls of real applications, and memcpy operations are overlapped with GPU kernel computation when the application executes several HE Muls concurrently.



(a) Execution time on single-thread (b) Execution time on 24-threads (c) Relative execution time and speedup per function

FIGURE 7. Comparing HE Mul execution time among Ref, AVX-512 implementation (AVX), and optimized AVX (AVX-M and AVX-MT) when (a) one and (b) 24 threads are utilized, and (c) per-function speedup when using 24 threads.

TABLE 7. Comparing the execution time of HE Mul among a single- and 24-thread reference HEAAN (Ref-1 and Ref-24), a 24-thread optimized AVX-512 implementation (AVX-MT-24), and an optimized GPU implementation (GPU-CLH).

	Execution time (ms) and [Speedup vs. **Ref**-24]			
Func	**Ref**-1	**Ref**-24	**AVX-MT**-24	**GPU-CLH**
`CRT`	515.9	35.9	12.3 [2.91×]	4.1 [8.74×]
`NTT`	497.4	24.4	7.0 [3.49×]	3.7 [6.59×]
`iNTT`	520.5	24.8	7.4 [3.35×]	4.7 [5.27×]
`iCRT`	1761.6	59.4	37.2 [1.60×]	19.4 [3.06×]
Extra	133.9	10.0	11.0 [0.91×]	6.2 [1.61×]
Total	3429.4	154.5	74.8 [**2.06×**]	38.1 [**4.05×**]

populated on the GPU. Also, by reducing the main memory accesses by increasing the radix, **GPU-CLH** achieves 1.9× and 1.6× performance improvements in `NTT` and `iNTT`, respectively, compared to the **AVX-MT**'s implementation.

Second, our CPU implementations are highly scalable across both intra-core and inter-core dimensions. **AVX** is effective regardless of the number of CPU threads populated, providing 1.6× and 1.8× performance gains over **Ref** when a single and 24 threads are utilized, respectively (see Figure 7(a) and (b)). Among the primary functions, `NTT` is best in terms of scalability, leading to a 3.2× speedup for **AVX** over **Ref** when 24 threads are populated. Overall, **AVX** experiences a 25.2× speedup when the number of populated threads increases from 1 to 24, exhibiting better scalability than **Ref** (22.2×) for the following reasons. **AVX** and **Ref** exploit parallelism in different ways for `CRT`, as described in Section V. In **Ref**, each thread operates on different prime numbers (m-degree parallelism), where m is not large (e.g., 42 or 63). Hence, **Ref** is more susceptible to a load imbalance across the threads. In contrast, each thread operates on different coefficients (N-degree parallelism) in **AVX**, exhibiting better scalability. For `iCRT`, data accesses for the matrix occur in the column direction during matrix-matrix mul, causing the performance to be memory-bound because hardware prefetching becomes ineffective. However, with 24 threads being utilized, hardware prefetching hits more frequently because a thread may access the data in adjacent columns that are prefetched by other threads, leading to even superlinear speedups.

TABLE 8. Cache ($) hit rates in `iCRT` implementations. GPU-CP case is not shown here because it mostly exploits shared memory and accesses L1 $ much less.

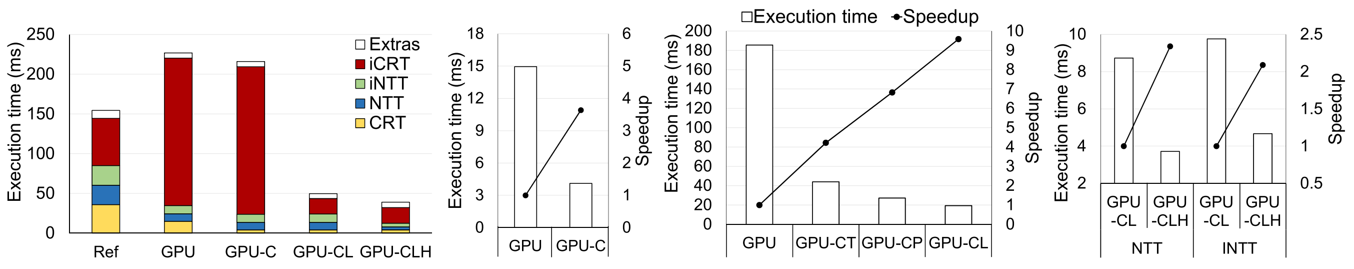
	GPU	GPU-CT	GPU-CL
L1 $	26%	98%	88%
L2 $	53%	99%	82%

The additional optimizations applied to the AVX-512 implementation are effective as well. Figure 7(c) shows the impact of these optimizations on each major function when 24 threads are used. In **AVX-M**, `NTT` is 8% faster and `iNTT` is 10% faster than **AVX** because these functions compute modular mul frequently. `iCRT` experiences a 21% speedup in **AVX-MT** compared to **AVX-M** because the matrix transposition alleviates the memory-bound issue.

Third, the performance of GPU reaches nearly full potential through our architecture-centric optimizations. Figure 8(a) shows the execution time of HE Mul on various GPU implementations compared to that of the reference HEAAN running on a CPU with 24 threads (**Ref**-24). The baseline GPU implementation (**GPU**) is slower than **Ref**-24 by 1.47×, as `iCRT` in **GPU**, whose implementation we adopt from cuHE [25], performs poorly; it is 3.12× slower than that in **Ref**-24 and takes 81.8% of the total HE Mul execution time.

To reduce the execution time of `iCRT`, we devised the following optimizations and compared their performance in Figure 8(c) and cache ($) hit rates in Table 8. The inferior performance of `iCRT` in the **GPU** case is due to cache thrashing. Because the thread-local array *accum* (line 3 in Algo. 7) is stored in local memory, all of the in-flight threads compete for a cache resource, leading to low cache hit rates (see Table 8). The baseline `iCRT` issues only 0.03 instructions per cycle per warp scheduler on average (1.0 is the machine peak); the major reason for a stall is the long scoreboard dependency, resulting in 193.0 stall cycles between issuing two instructions on average.

By adjusting the number of launching threads, we reduced the degree of the performance impact due to $ thrashing, achieving a speedup of 4.22× (**GPU-CT**) compared to that of **GPU**. **GPU-CT** increases the instruction issue



(a) HE Mul execution time breakdown (b) CRT optimization (c) iCRT optimization (d) NTT/iNTT optimization

FIGURE 8. Comparing HE Mul execution time (a) among Ref-24, the baseline GPU (GPU), and optimized GPU (GPU-C, GPU-CL and GPU-CLH), and the execution time and relative speedup of (b) GPU and GPU-C for `CRT`, (c) GPU and GPU-C[T/P/L] for `iCRT`, and (d) GPU-CL and GPU-CLH for `NTT/iNTT`.

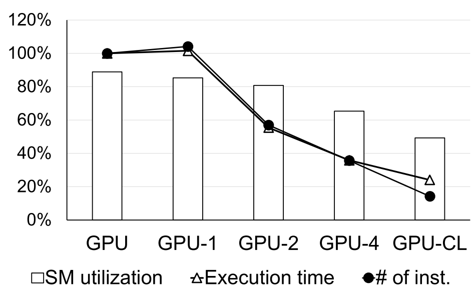


FIGURE 9. SM utilization, the relative execution time, and the relative number of executed instructions on various `CRT` kernels. GPU-*x* means one modulo operation is applied at every *x* iteration. We used the parameters in region 1.

throughput from 0.03 to 0.16 while reducing the scoreboard stall cycle to 2.0.

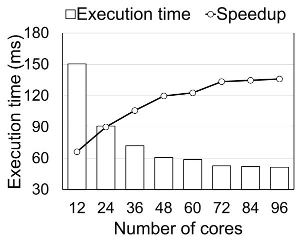
Although **GPU-CT** increases the hit rates of both L1 and L2 \$, it exhibits a low *occupancy*, the average ratio of the number of in-flight warps to the maximum, of 11% as it launches fewer threads. Pinning thread-local arrays to L1 \$ (**GPU-CP**) performs better than **GPU-CT**, resulting in a speedup of 6.79×. **GPU-CP** increases the occupancy to 16%; here, the occupancy is limited due to the heavy use of shared memory. Finally, **GPU-CL** was the best among all the `iCRT` optimizations (a 9.58× speedup) because it effectively exploited $N \cdot m$-degree parallelism through the loop reordering (explained in Section V). **GPU-CL** hits \$ better than **GPU** because *PLimbs* threads in **GPU-CL** share one input coefficient, whereas the threads in **GPU** each takes one input coefficient. This reduces the \$ contention between warps, as each warp demands fewer input coefficients, which are much larger than the precomputed table ($\mathbf{TB_{Pdivp}}$ in Algo. 5), while also offering higher occupancy (67.3% on average).

For `CRT`, performing fewer modulo operations led to better performance. Figure 9 compares the various implementations of `CRT` kernels. **GPU** does not precompute mod (β^k, p_j) and performs a modulo operation on every limb (one limb for β) of the BigInt, exhibiting a SM utilization rate of 88.9% (specifically, the utilization of FP64 (double-precision floating-point) pipes). The modulo operations, each being translated to hundreds of instructions, place significant pressure on arithmetic pipes. **GPU-1**, which transforms the repetitive $\log\beta$−bit shift and add operations into muls with the precomputed mod(β^k, p_j), performs even worse than **GPU** by 1.02× because the number of modulo operations remains the same. By applying modulo operations intermittently (**GPU-2** to **GPU-4**), both the number of instructions and the execution time are drastically decreased; **GPU-4** outperforms **GPU** by 2.79×. Finally, by letting the partial sum (*accum*) span three words instead of two words while utilizing ADC in every iteration, **GPU-C** performs best with a speedup of 3.64×.

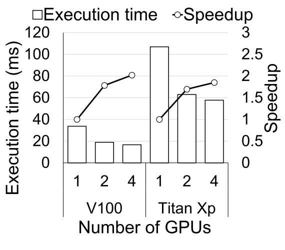
B. PERFORMANCE-LIMITING FACTORS

We also analyzed the performance limiting factors of a single HE Mul on the CPU and GPU implementations that perform best. In **AVX-MT**, we measured the instructions per cycle (IPC) for AVX-512 instructions (whose maximum value is two per core [44]. `CRT`, `NTT`, `iNTT`, and `iCRT` achieved 87.6%, 59.4%, 66.9%, and 80.7% of the peak IPC, respectively. While the utilization rates are all high, `NTT` and `iNTT` present lower numbers because they are more often bound to the memory bandwidth.

In **GPU-CLH**, the main performance bottlenecks come from the bandwidth of L1 \$ and the main memory. `CRT`, `iCRT`, `NTT`, and `iNTT` utilize the main-memory bandwidth at rates of 8.7%, 24.1%, 69.9%, and 79.3%, respectively, where the peak bandwidth reaches 672 GB/s [2]. Large precomputed tables of `NTT` and `iNTT` having $O(N \cdot m)$ sizes (see Table 5) demand high memory bandwidth, lowering the arithmetic intensity. In contrast, `CRT` is mainly bottlenecked by the L1 data \$ load bandwidth. `CRT` consumes 61% of the L1 \$ bandwidth as measured from a microbenchmark stressing L1 \$ (56.2B per cycle per SM [45], [46]). The gap from the peak comes from the transposing of an output at the end of `CRT` and from cold start misses. Three phases of `iCRT` stress different parts of the GPU microarchitecture. 1) The Hadamard product (lines 1-2 in Algo. 8) is limited by the main memory bandwidth, utilizing 64% of the peak bandwidth. 2) The phase computing the partial sums of each BigInt coefficient (lines 3-16 in Algo. 8) is mainly bottlenecked by L1 data \$, reaching 77.2% of the L1 \$ load bandwidth. Its computational characteristic is similar to that of `CRT` (matrix-matrix mul); it achieves higher utilization due to the absence of output transposition. 3) The last phase aggregates the partial sums and reduces them to BigInt coefficients (lines 17-19 in Algo. 8). Here, the parallelism we can exploit is



(a) multi-CPU



(b) multi-GPU

FIGURE 10. HE Mul execution time and speedup of (a) AVX-MT on a various number of CPU cores, and (b) GPU-CLH implementation on multi-GPU systems with different numbers of GPUs utilized.

limited to just N. We use the best-performing method of pinning thread-local, similar to **GPU-CP**. This offers a low occupancy (16%) due to the heavy use of shared memory, where its limited capacity becomes a bottleneck.

C. MULTI-CPU AND MULTI-GPU SCALABILITY STUDY

We also evaluated the scalability of our CPU and GPU implementations of HE Mul. For the CPU implementation, we populated up to 96 cores on a four-socket server with the node interleaving option enabled. Figure 10(a) shows the performance of **AVX-MT** when utilizing different numbers of cores. As the number of cores increases, the performance improves; however, owing to the synchronization overhead, it becomes saturated after 72 cores, eventually resulting in an overall speedup of 1.76× on 96 cores compared to that with 24 cores.

Figure 10(b) shows the speedup of HE Mul with multiple GPUs compared to the execution time with a single GPU (**GPU-CLH**), showing outcomes of 106.9 ms and 33.7 ms on Titan X and V100, respectively. Using two GPUs results in a 1.78× speedup for V100 and a 1.70× speedup for Titan X, while using four GPUs results in corresponding outcomes of 2.02× and 1.85×. The scalability of our multi-GPU implementation is primarily limited by the workload imbalance between the streams. As described in Section 5, the sequence of computational kernels in stream 1, which is the critical path of the workflow, takes much longer than the other streams. In the cases with two GPUs, as we assign stream 1 to only one GPU and streams 2, 3, and 4 to the other GPUs, the performance degradation due to the workload imbalance is less significant, with the speedup as high as 1.78×. In contrast, in the cases with four GPUs, the GPUs in charge of stream 2 or stream 3 are mostly idle, waiting for stream 1 and stream 4 to finish the operations in region 2.

The communication overhead limits the performance less significantly; with four V100 (Titan X) GPUs, 12.34% (12.75%) of the execution time of an HE Mul is blocked by synchronization for data acquisition.

VIII. DISCUSSION & RELATED WORK

A. OVERHEAD OF EMULATION ON AVX-512

As described in Section V, the cost of emulating 64-bit mul, modular mul, and ADC operations is significant in the AVX-512 implementation. To evaluate the emulation overhead in these cases, we analyzed the changes in the number of AVX-512 instructions when the CPU supports single native instructions for these operations. Table 9 summarizes the number of AVX-512 instructions required to perform each major function by comparing the cases where each case of 64-bit mul, modular mul, and ADC is supported by either emulation or a single native instruction. `CRT` and `iCRT` require 17.3% and 15.8% of AVX-512 instructions if a CPU supports these instructions natively, not through emulation. `NTT` and `iNTT` require one-third of the instructions via instruction extension. These results show why the use of AVX-512 achieves an insufficient performance improvement, also implying that we can substantially accelerate HE Mul by natively supporting these SIMD instructions. Previous work [49], [53] also showed that SIMD can effectively accelerate `NTT` and `iNTT` on CPUs and GPUs.

TABLE 9. The number of required AVX-512 instructions (add, sub, mul, shift, and cmp) for each function with (*m*, *qLimbs*) of (43, 19) when 64-bit mul, modular mul, and ADC is supported by emulation or by a single native instruction.

	CRT	NTT	iNTT	iCRT
by emulation	155M	48M	47M	319M
by single native instruction	27M	17M	17M	51M

B. IMPACT OF Q ON THE CHARACTERISTICS OF HE MUL

Q determines the multiplicative depth L; a larger depth requires a larger Q. However, N must increase proportionally to log$Q$ to ensure a certain level of security (see Table 2). Also, *qLimbs*, *m*, and *PLimbs* increase in proportion to log$Q$. Based on these relationships, the computational complexity shown in Table 4 can be expressed in terms of Q. The complexity of `CRT` and `iCRT` is $\mathcal{O}((\log Q)^3)$ whereas that of `NTT` and `iNTT` is $\mathcal{O}(\log(\log Q)\cdot(\log Q)^2)$. Figure 11 shows the estimated number of operations for HE Mul according to log$Q$. When Q is small (e.g., $\log Q = 150$), all major functions require a similar number of operations, but as Q increases, `CRT` and `iCRT` become more dominant. Overall, the total number of operations for HE Mul is proportional to $(\log Q)^3$. When an application requires a large number (e.g., billions) of HE Mul operations, using a large value of Q amortizes the cost of the expensive bootstrapping. However, using too large a Q value is costly because the maximum number of messages (n) that can be multiplied together by a HE Mul is $N/2$, where the complexity of a HE Mul is super-linear, $\mathcal{O}((\log Q)^3) = \mathcal{O}(N^3)$. The log$Q$ value we mainly target is 1,200, which is large enough to amortize the cost of bootstrapping. In contrast, other HE accelerators [61], [64] focused on much smaller Q values (e.g., [64] used a log$Q$ value of 180 without considering bootstrapping).

C. IMPACT OF q ON THE CHARACTERISTICS OF HE MUL

As described in Section III, rescaling, which decreases log$q$ by log$p$, is performed after each HE Mul to prevent

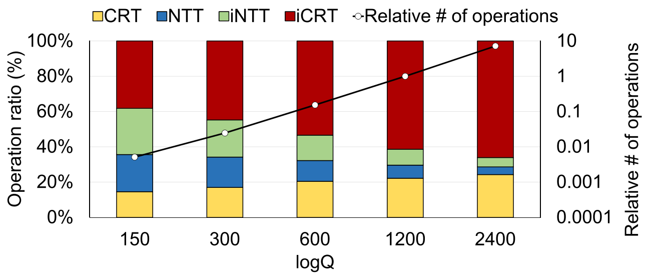


FIGURE 11. Distribution on the number of operations across functions and the relative number of operations for HE Mul on various logQ values. The reference logQ is 1,200.

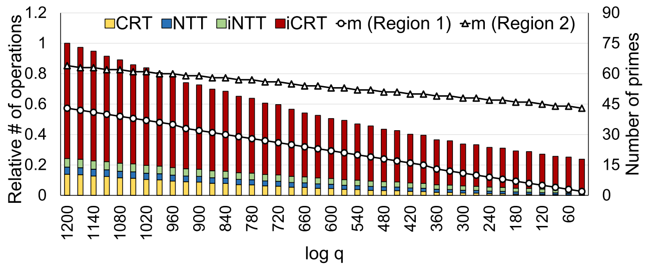


FIGURE 12. The size of *m* and the relative number of operations per function according to log*q*.

the amount of message information in the ciphertext from increasing exponentially. As HE Mul is repeated, q decreases, as do *qLimbs* and m. In region 1 of HE Mul, m decreases linearly with $\log q$ as two $\log q^2$-bit BitInt numbers are multiplied per ciphertext coefficient. In contrast, in region 2, m should be set to represent $(\log q + \log Q^2)$-bit BigInt (to multiply over the evaluation key polynomial). The trend for *PLimbs* is identical to that of m. Figure 12 shows the amount of computation for HE Mul according to $\log q$ in the AVX-512 configuration described in Section VI. As m is proportional to $(\log q + \log Q^2)$ in region 2, the number of operations for HE Mul when $\log q$ becomes 30 (the smallest number where no additional HE Mul is applicable) is still 24% of that when $\log q$ is 1,200. Also, `iCRT` is dominant regardless of the size of q.

D. ACCELERATING OTHER CKKS LIBRARIES

Besides HEAAN, there are several other libraries [3], [4], [16], [36] that support a full-RNS variant of CKKS [18]. Full-RNS CKKS makes each coefficient of the polynomials remain in an RNS domain during all HE operations. Compared to non-RNS CKKS, full-RNS CKKS is typically faster but less flexible because there exist rigid limitations when choosing p and Q. In full-RNS CKKS, each prime number composing an RNS representation of Q should be set such that it is close to p to suppress approximation errors [18]. In non-RNS CKKS, on the other hand, one can freely choose p independent of Q. Moreover, these parameter limitations make the multiplicative depth of full-RNS variants lower than that of non-RNS schemes for a given security bit and error bound.

We can apply the optimization techniques described in Section V to these other CKKS libraries. Full-RNS variants do not require `CRT` and `iCRT`, but the optimizations for `NTT` and `iNTT` can be applied. Also, we can partially apply our techniques in `CRT` to functions that change the number of primes in the RNS domain (`mod up` and `mod down`). `mod up` increases the number of primes for a given big integer, whereas `mod down` decreases it with an additional division operation [11], [18]. Both can take advantage of our optimizations, as their core functions are similar to when `CRT` is applied immediately after `iCRT`. **FPGA-based HE accelerators:** Numerous studies [24], [28], [59], [61], [63], [64] have attempted to accelerate HE operations using FPGA. [24], [28], [59] accelerate LTV-based FHE schemes whereas [63], [64] accelerate FV-based FHE schemes. However, LTV and FV schemes are of limited practical use because they cannot perform approximate computations. HEAX [61] uses FPGA to accelerate Microsoft SEAL, which supports a full-RNS variant of the CKKS scheme; however, HEAX considers only small parameter sizes ($\log Q \leq 438$ and $N \leq 2^{14}$), and the full-RNS variant it targets is not as versatile as the original HEAAN we accelerate in this paper due to the limitations that exist when choosing the rescaling factors and prime numbers.

E. GPU LIBRARIES FOR HE

[6], [9], [10], [25] propose to accelerate HE operations using GPUs. Because they target schemes other than CKKS and use different parameter sets, we can only compare individual kernels and the algorithms adopted. References [10] and [9] adopt Garner's iCRT algorithm. Garner's algorithm achieves a speedup of 33.1% over cuHE's iCRT on $(N, \log Q) = (2^{14}, 744)$ [10]. However, the parallelism in iCRT is still limited to N-degrees. Also, these methods do not adopt ADC operations to reduce the number of modular reductions in CRT.

For NTT and iNTT, [49] characterizes various NTT implementations, including the high-radix approach in this paper, suggesting on-the-fly twiddle factor generation. Another approach [10] is to exploit the Discrete Galois Transform (DGT) rather than NTT, reducing the N-point datapath to $N/2$-points with the use of Gaussian integers.

F. MULTI-GPU IMPLEMENTATION

A prior study [5] that implemented the HE operations of a variant of BFV [35] on a multi-GPU cluster took a different strategy while exploiting data parallelism. Instead of allocating a polynomial to a GPU, [5] represents each polynomial as a 2D matrix and partitions it in a column- and row-wise manner, distributing the polynomial over all GPUs. Whenever needed, this method transposes the polynomial so that each GPU holds either rows or columns of a polynomial. For example, considering the HE Mul of HEAAN in Figure 2, a transposition should occur between `CRT` and `NTT` and also between `iNTT` and `iCRT`. Their approach can parallelize a HE Mul to an arbitrary number of GPUs at the cost of communication and synchronization; all GPUs should communicate in an all-to-all manner per transposition. One can combine their approach with ours, enjoying more parallelism with a lower communication cost.

G. ROAD TO THE PRACTICAL USE OF HE

There are numerous machine-learning based applications that target inference tasks using HE [15], [22], [30], [43], [47], [52]. However, they are still too slow to be used practically; for example, the state-of-the-art HE-enabled CNN model [52] reports a latency of 96.4 seconds for a single inference with the CIFAR-10 dataset. We believe that our implementations can be applied to such machine learning tasks and reduce the latencies to few seconds, which could be considered real-time.

IX. CONCLUSION

We have demystified the key operations of HEAAN, a representative and popular FHE scheme. After identifying that multiplying encrypted data (ciphertext) is the most computationally demanding operation, we accelerated the major functions of HE Mul (`CRT`, `NTT`, `iNTT`, and `iCRT`) on CPUs and GPUs. To accelerate the major functions on a CPU, we populate multiple cores by means of multi-threading (inter-core parallelism) and with AVX-512 instructions (intra-core parallelism). We accelerate HE Mul on a GPU by effectively exploiting the massive thread-level parallelism. Moreover, based on an in-depth analysis of the major functions for HE Mul, we introduced a series of architecture-centric optimization techniques, specifically loop reordering and matrix transposition for `iCRT`, using the synergy between precomputation and the delayed modulo operations for `CRT`. Our accelerated HEAAN on a single CPU and single GPU outperforms the reference HEAAN on a CPU with 24 threads by 2.06× and 4.05×, respectively.

ACKNOWLEDGMENT

This work was done when Eojin Lee was at Seoul National University.

REFERENCES

[1] (Sep. 2018). *HEAAN With Faster Multiplication*. [Online]. Available: https://github.com/snucrypto/HEAAN/releases/tag/2.1

[2] (May 2019). *NVIDIA TITAN RTX Product Overview*. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/titan/documents/tita%n-rtx-for-creators-us-nvidia-1011126-r6-web.pdf

[3] (Feb. 2020). *Lattigo 1.3.1*. ePFL-LDS. [Online]. Available: http://github.com/ldsec/lattigo

[4] (Jan. 2020). *PALISADE LAttice CRyptography LIbrary (Release 1.7.4)*. [Online]. Available: https://palisade-crypto.org/

[5] A. Al Badawi, B. Veeravalli, J. Lin, N. Xiao, M. Kazuaki, and A. K. M. Mi, "Multi-GPU design and performance evaluation of homomorphic encryption on GPU clusters," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 2, pp. 379–391, Feb. 2021.

[6] P. Alves and D. Aranha, "Efficient GPGPU implementation of the leveled fully homomorphic encryption scheme YASHE," Tech. Rep., Jun. 2016. [Online]. Available: https://www.researchgate.net/publication/320619287_Efficient_GPGPU_implementation_of_the_Leveled_Fully_Homomorphic_Encryption_scheme_YASHE

[7] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[8] A. Al Badawi, J. Chao, J. Lin, C. F. Mun, J. J. Sim, B. H. M. Tan, X. Nan, K. M. M. Aung, and V. R. Chandrasekhar, "Towards the AlexNet moment for homomorphic encryption: HCNN, theFirst homomorphic CNN on encrypted data with GPUs," 2018, *arXiv:1811.00778*. [Online]. Available: http://arxiv.org/abs/1811.00778

[9] A. Al Badawi, Y. Polyakov, K. M. M. Aung, B. Veeravalli, and K. Rohloff, "Implementation and performance evaluation of RNS variants of the BFV homomorphic encryption scheme," *IEEE Trans. Emerg. Topics Comput.*, vol. 9, no. 2, pp. 941–956, Apr. 2021.

[10] A. A. Badawi, B. Veeravalli, C. F. Mun, and K. M. M. Aung, "High-performance FV somewhat homomorphic encryption on GPUs: An implementation using CUDA," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2018, pp. 70–95, May 2018.

[11] J.-C. Bajard, J. Eynard, M. A. Hasan, and V. Zucca, "A full RNS variant of FV like somewhat homomorphic encryption schemes," in *Proc. Int. Conf. Sel. Areas Cryptogr.*, 2016, pp. 432–442.

[12] D. J. Bernstein, "Fast multiplication and its applications," in *Proc. Int. Algorithmic Number Theory Symp.*, vol. 44, 2008, pp. 325–384.

[13] F. Boemer, A. Costache, R. Cammarota, and C. Wierzynski, "NGraph-HE2: A high-throughput framework for neural network inference on encrypted data," in *Proc. 7th ACM Workshop Encrypted Comput. Appl. Homomorphic Cryptography (WAHC)*, 2019, pp. 45–56.

[14] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," *ACM Trans. Comput. Theory*, vol. 6, no. 3, pp. 1–36, Jul. 2014.

[15] A. Brutzkus, O. Elisha, and R. Gilad-Bachrach, "Low latency privacy preserving inference," 2018, *arXiv:1812.10659*. [Online]. Available: http://arxiv.org/abs/1812.10659

[16] H. Chen, K. Laine, and R. Player, "Simple encrypted arithmetic library-SEAL v2.1," in *Proc. Int. Conf. Financial Cryptogr. Data Secur.*, 2017, pp. 3–18.

[17] J. Cheon, K. Han, and M. Hhan, "Faster homomorphic discrete Fourier transforms and improved FHE bootstrapping," *Int. Assoc. Cryptologic Res. Cryptol. ePrint Arch.*, vol. 2018, p. 1073, Nov. 2018.

[18] J. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "A full RNS variant of approximate homomorphic encryption," in *Proc. Int. Conf. Sel. Areas Cryptogr.*, 2018, pp. 347–368.

[19] J. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "Bootstrapping for approximate homomorphic encryption," in *Proc. Annu. Int. Conf. Theory Appl. Cryptograph. Techn.*, 2018, pp. 360–384.

[20] J. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *Proc. Int. Conf. Theory Appl. Cryptol. Inf. Secur.*, 2017, pp. 409–437.

[21] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: Fast fully homomorphic encryption over the torus," *J. Cryptol.*, vol. 33, no. 1, pp. 34–91, Jan. 2020.

[22] E. Chou, J. Beal, D. Levy, S. Yeung, A. Haque, and L. Fei-Fei, "Faster CryptoNets: Leveraging sparsity for real-world encrypted inference," 2018, *arXiv:1811.09953*. [Online]. Available: http://arxiv.org/abs/1811.09953

[23] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Math. Comput.*, vol. 19, no. 90, pp. 297–301, May 1965.

[24] D. B. Cousins, K. Rohloff, and D. Sumorok, "Designing an FPGA-accelerated homomorphic encryption co-processor," *IEEE Trans. Emerg. Topics Comput.*, vol. 5, no. 2, pp. 193–206, Apr. 2017.

[25] W. Dai and B. Sunar, "cuHE: A homomorphic encryption accelerator library," in *Proc. Int. Conf. Cryptography Inf. Secur.*, 2015, pp. 169–186.

[26] T. Dillon, C. Wu, and E. Chang, "Cloud computing: Issues and challenges," in *Proc. IEEE Int. Conf. Adv. Inf. Netw. Appl.*, Apr. 2010, pp. 27–33.

[27] P. Dingyi, S. Arto, and D. Cunsheng, *Chinese Remainder Theorem: Applications in Computing, Coding, Cryptography*. Singapore: World Scientific, 1996.

[28] Y. Doröz, E. Öztürk, E. Savaş, and B. Sunar, "Accelerating LTV based homomorphic encryption in reconfigurable hardware," in *Proc. Int. Workshop Cryptograph. Hardw. Embedded Syst.*, 2015, pp. 185–204.

[29] J. Doweck, W.-F. Kao, A. K.-Y. Lu, J. Mandelblat, A. Rahatekar, L. Rappoport, E. Rotem, A. Yasin, and A. Yoaz, "Inside 6th-generation intel core: New microarchitecture code-named skylake," *IEEE Micro*, vol. 37, no. 2, pp. 52–62, Mar. 2017.

[30] N. Dowlin, R. Gilad-Bachrach, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing, "CryptoNets: Applying neural networks to encrypted data with high throughput and accuracy," in *Proc. 33rd Int. Conf. Int. Conf. Mach. Learn.*, 2016, pp. 201–210.
[31] L. Ducas and D. Micciancio, "FHEW: Bootstrapping homomorphic encryption in less than a second," in *Proc. Annu. Int. Conf. Theory Appl. Cryptograph. Techn.*, 2015, pp. 617–640.
[32] T. Elgamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *IEEE Trans. Inf. Theory*, vol. IT-31, no. 4, pp. 469–472, Jul. 1985.
[33] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," *Int. Assoc. Cryptologic Res. Cryptol. ePrint Arch.*, vol. 2012, p. 144, Mar. 2012.
[34] C. Gentry, "A fully homomorphic encryption scheme," Ph.D. dissertation, Dept. Comput. Sci., Stanford Univ., Stanford, CA, USA, 2009.
[35] S. Halevi, Y. Polyakov, and V. Shoup, "An improved RNS variant of the BFV homomorphic encryption scheme," in *Proc. Cryptographers' Track RSA Conf.*, 2019, pp. 83–105.
[36] S. Halevi and V. Shoup. (Jan. 2020). *HElib: An Implementation of Homomorphic Encryption*. [Online]. Available: https://github.com/homenc/HElib/releases/tag/v1.0.0
[37] K. Han. (Apr. 2019). *HELR: Homomorphic Logistic Regression on Encrypted Data*. [Online]. Available: https://github.com/KyoohyungHan/HELR
[38] K. Han, S. Hong, J. H. Cheon, and D. Park, "Logistic regression on homomorphic encrypted data at scale," in *Proc. AAAI Conf. Artif. Intell.*, vol. 33, Jul. 2019, pp. 9466–9471.
[39] M. Harris. (Apr. 2013). *CUDA Pro Tip: Write Flexible Kernels With Grid-Stride Loops*. [Online]. Available: https://devblogs.nvidia.com/cuda-pro-tip-write-flexible-kernels-grid-st%ride-loops
[40] D. Harvey, "Faster arithmetic for number-theoretic transforms," *J. Symbolic Comput.*, vol. 60, pp. 113–119, Jan. 2014.
[41] B. Hayes, "Cloud computing," *Commun. ACM*, vol. 51, no. 7, pp. 9–11, 2008.
[42] iDASH Privacy Protection Challenge 2018. (2018). *Secure Genome Analysis Competition*. [Online]. Available: http://www.humangenomeprivacy.org/2018/
[43] iDASH Privacy Protection Challenge 2019. (2019). *Secure Genome Analysis Competition*. [Online]. Available: http://www.humangenomeprivacy.org/2019/
[44] *Intel 64 and Ia-32 Architectures Optimization Reference Manual*, Intel Corporation, Santa Clara, CA, USA, May 2020.
[45] Z. Jia, M. Maggioni, J. Smith, and D. Scarpazza, "Dissecting the NVidia turing T4 GPU via microbenchmarking," 2019, *arXiv:1903.07486*. [Online]. Available: https://arxiv.org/abs/1903.07486
[46] Z. Jia, M. Maggioni, B. Staiger, and D. Scarpazza, "Dissecting the NVIDIA volta GPU architecture via microbenchmarking," 2018, *arXiv:1804.06826*. [Online]. Available: https://arxiv.org/abs/1804.06826
[47] X. Jiang, M. Kim, K. Lauter, and Y. Song, "Secure outsourced matrix computation and application to neural networks," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2018, pp. 1209–1222.
[48] A. Kim, Y. Song, M. Kim, K. Lee, and J. H. Cheon, "Logistic regression model training based on the approximate homomorphic encryption," *BMC Med. Genomics*, vol. 11, no. S4, pp. 23–31, Oct. 2018.
[49] S. Kim, W. Jung, J. Park, and J. H. Ahn, "Accelerating number theoretic transformations for bootstrappable homomorphic encryption on GPUs," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Oct. 2020, pp. 264–275.
[50] Y. Lecun. (1998). *The MNIST Database of Handwritten Digits*. [Online]. Available: http://yann.lecun.com/exdb/mnist
[51] A. López-Alt, E. Tromer, and V. Vaikuntanathan, "On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption," in *Proc. 44th Symp. Theory Comput. (STOC)*, 2012, pp. 1219–1234.
[52] Y. Lu, J. Lin, C. Jin, Z. Wang, K. Mi Mi Aung, and X. Li, "FFConv: Fast factorized neural network inference on encrypted data," 2021, *arXiv:2102.03494*. [Online]. Available: http://arxiv.org/abs/2102.03494
[53] D. Natarajan and R. Dreslinski, "Performance characterization of lattice-based cryptography workloads," in *Proc. Int. Symp. Perform. Anal. Syst. Softw.*, Aug. 2020, pp. 220–222.
[54] "NVIDIA tesla V100 GPU architecture," NVIDIA Corp., Santa Clara, CA, USA, Tech. Rep. WP-08608-001_v1.1, Aug. 2017.
[55] "NVIDIA turing GPU architecture," NVIDIA Corp., Santa Clara, CA, USA, Tech. Rep. WP-09183-001_v01, Sep. 2018.
[56] NVIDIA Corporation. (Aug. 2019). *CUDA C++ Programming Guide*. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide
[57] NVIDIA Corporation. (Aug. 2019). *Parallel Thread Execution ISA*. [Online]. Available: https://docs.nvidia.com/cuda/parallel-thread-execution
[58] NVIDIA Corporation. (May 2021). *NVIDIA Nsight Compute*. [Online]. Available: https://developer.nvidia.com/nsight-compute
[59] E. Öztürk, Y. Doröz, E. Savaş, and B. Sunar, "A custom accelerator for homomorphic encryption applications," *IEEE Trans. Comput.*, vol. 66, no. 1, pp. 3–16, Jan. 2017.
[60] P. Paillier, "Public-key cryptosystems based on composite degree Residuosity classes," in *Proc. Int. Conf. Theory Appl. Cryptograph. Techn.*, 1999, pp. 223–238.
[61] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "HEAX: An architecture for computing on encrypted data," in *Proc. 25th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Mar. 2020, pp. 1295–1309.
[62] R. L. Rivest, L. Adleman, and M. L. Dertouzos, "On data banks and privacy homomorphisms," *Found. Secure Comput.*, vol. 4, no. 11, pp. 169–180, 1978.
[63] S. S. Roy, K. Jarvinen, J. Vliegen, F. Vercauteren, and I. Verbauwhede, "HEPCloud: An FPGA-based multicore processor for FV somewhat homomorphic function evaluation," *IEEE Trans. Comput.*, vol. 67, no. 11, pp. 1637–1650, Nov. 2018.
[64] S. S. Roy, F. Turan, K. Jarvinen, F. Vercauteren, and I. Verbauwhede, "FPGA-based high-performance parallel architecture for homomorphic computing on encrypted data," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2019, pp. 387–398.
[65] S. Subashini and V. Kavitha, "A survey on security issues in service delivery models of cloud computing," *J. Netw. Comput. Appl.*, vol. 34, no. 1, pp. 1–11, Jan. 2011.

WONKYUNG JUNG (Graduate Student Member, IEEE) received the B.S. degree in electrical engineering from the Pohang University of Science and Technology, in 2017. He is currently pursuing the Ph.D. degree with the Graduate School of Convergence Science and Technology, Seoul National University. His research interests include optimizing emerging applications exploiting the characteristics of the latest and future hardware.

EOJIN LEE received the B.S. degree in electrical and computer engineering from Seoul National University, in 2013, and the Ph.D. degree from the Graduate School of Convergence Science and Technology, Seoul National University, in 2020. He is currently a Staff Engineer with the DRAM Design Team, Samsung Electronics. He is interested in in-memory processing for high performance DRAMs.

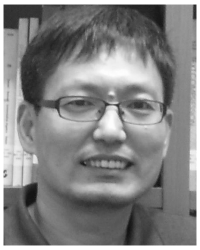
SANGPYO KIM (Graduate Student Member, IEEE) received the B.S. degree in naval architecture and ocean engineering from Seoul National University, in 2019, where he is currently pursuing the M.S. degree with the Graduate School of Convergence Science and Technology. His research interests include parallel processing, combinatorial optimization, data security technologies, and computer architecture for accelerating emerging applications.

JONGMIN KIM received the B.S. degree in electrical engineering from Seoul National University, in 2021, where he is currently pursuing the M.S. degree with the Interdisciplinary Program in Artificial Intelligence. His research interests include architecture-centric optimization, hardware accelerator design of emerging applications, and applied cryptography.

NAMHOON KIM received the B.S. degree in electrical engineering from the Korea Advanced Institute of Science and Technology (KAIST) and the M.S. degree in electrical and computer engineering from Seoul National University, where he is currently pursuing the Ph.D. degree with the Graduate School of Convergence Science and Technology. His research interest includes emerging memory system optimization.

KEEWOO LEE received the B.S. degree in mathematical sciences from Seoul National University, where he is currently pursuing the Ph.D. degree. His research interests include cryptographic primitives for secure computation and their applications.

CHOHONG MIN was born in Anyang, South Korea, in 1972. He received the B.S. degree from the Department of Mathematics, Korea Advanced Institute of Science and Technology, in 1997, and the Ph.D. degree from the Department of Mathematics, University of California at Los Angeles, in 2004. He was a Visiting Assistant Professor with the Department of Mathematics, University of California at Santa Barbara, from 2004 to 2007, and an Assistant Professor with Kyung Hee University, South Korea. He has been an Associate Professor with Ewha Womans University, since 2010. Recently he proved Gustafsson conjecture, a long-time unsolved problem in the case irregular domains. During the last ten years, he published 21 articles in applied mathematics and computational physics journals and 18 therein are published in top 10% SCI journals in the areas. His main research interest includes mathematical analysis. He was a recipient of the Excellent Paper Award in *Korean Society for Industrial and Applied Mathematics*, in 2017.



JUNG HEE CHEON (Associate Member, IEEE) received the Ph.D. degree in mathematics from KAIST. He is currently a Professor of mathematics and the Director of IMDARC (the center for industrial math), Seoul National University. He is also working on computational number theory, cryptology, and information security. He is also a Co-Inventor of braid cryptography and approximate homomorphic encryption. He is also one of two invited speakers in Asiacrypt 2020. He received the Best Paper Award in Asiacrypt 2008 and Eurocrypt 2015, and the POSCO Science Prize, in 2019. He was selected as a Scientist of the Month by Korean Government, in 2018. He served as a Program Committee Member for Crypto/Eurocrypt. He was the Program Co-Chair of Asiacrypt 2015 and 2016. He is also an Associate Editor of *Journal of Cryptology* that is the flagship journal in cryptology.

JUNG HO AHN (Senior Member, IEEE) received the B.S. degree in electrical engineering from Seoul National University and the M.S. and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, USA.

He is currently a Professor with the Graduate School of Convergence Science and Technology, Seoul National University. He is interested in bridging the gap between the performance demand of emerging applications and the performance potential of modern and future massively parallel systems.