IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2021, No. 4, pp. 114–148. DOI:10.46586/tches.v2021.i4.114-148

Over 100x Faster Bootstrapping in Fully Homomorphic Encryption through Memory-centric Optimization with GPUs

Wonkyung Jung[1], Sangpyo Kim[1], Jung Ho Ahn[1], Jung Hee Cheon[1,2] and Younho Lee[3]

[1] Seoul National University, Seoul, Republic of Korea,
{jungwk,vnb987,gajh,jhcheon}@snu.ac.kr
[2] Crypto Lab. Inc, Seoul, South Korea,
[3] SeoulTech, Seoul, Republic of Korea, younholee@seoultech.ac.kr

Abstract. Fully Homomorphic encryption (FHE) has been gaining in popularity as an emerging means of enabling an unlimited number of operations in an encrypted message without decryption. A major drawback of FHE is its high computational cost. Specifically, a *bootstrapping* step that refreshes the noise accumulated through consequent FHE operations on the ciphertext can even take minutes of time. This significantly limits the practical use of FHE in numerous real applications.
By exploiting the massive parallelism available in FHE, we demonstrate the first instance of the implementation of a GPU for bootstrapping CKKS, one of the most promising FHE schemes supporting the arithmetic of approximate numbers. Through analyzing CKKS operations, we discover that the major performance bottleneck is their high main-memory bandwidth requirement, which is exacerbated by leveraging existing optimizations targeted to reduce the required computation. These observations motivate us to utilize memory-centric optimizations such as kernel fusion and reordering primary functions extensively.
Our GPU implementation shows a 7.02× speedup for a single CKKS multiplication compared to the state-of-the-art GPU implementation and an amortized bootstrapping time of 0.423us per bit, which corresponds to a speedup of 257× over a single-threaded CPU implementation. By applying this to logistic regression model training, we achieved a 40.0× speedup compared to the previous 8-thread CPU implementation with the same data.

Keywords: Fully Homomorphic Encryption · Bootstrapping · Logistic regression · GPU · Kernel fusion

1 Introduction

Homomorphic encryption (HE) enables one to perform operations on encrypted data without decrypting them and the result can be decrypted only with the secret key. As HE reveals nothing about the input or output except for the corresponding sizes during the computation step, it has been spotlighted as a core technology for applications such as privacy-preserving computations. HE schemes in the early stages had restrictions on the number and type of operations. After Gentry's breakthrough, however, these restrictions were removed, leading to Fully Homomorphic Encryption (FHE), where unlimited operations are allowed with the help of a bootstrapping operation [Gen09].

CKKS (Cheon-Kim-Kim-Song), an FHE scheme based on the ring learning with errors (RLWE) problem, is also a method that has recently attracted attention given its

Licensed under Creative Commons License CC-BY 4.0.
Received: 2021-04-15 Accepted: 2021-06-15 Published: 2021-08-11

efficient approximate computation [CHK+19]. As opposed to other FHE schemes, this scheme is equipped with rounding operations as well as the addition and multiplication of real numbers as primitive operations, thus providing efficient fixed-point arithmetic on ciphertexts. Its advantage is prominent in the fields of privacy-preserving data analysis and machine learning. In a secure genome analysis competition *iDash*, for instance, most of the winners and runner-ups on the HE track have employed CKKS since 2017 [WTW+18, KJT+20, JHK+20]. However, its performance is still far from satisfactory for industry practitioners [Cra19, YZLT19, ZLX+20, BHS19, HLF+19]. Bootstrapping, the most inefficient operation in CKKS, requires, for instance, more than 60 seconds in a conventional PC environment [CHK+18] on a 128-bit security parameter set.

In this paper, we address these challenges by improving the performance of CKKS with a Residue Number System (RNS) [CHK+19], one of the most efficient FHE schemes for many applications. Furthermore, we strive to maximize the parallelization of RLWE-based FHE operations by exploiting a high-performance commodity platform, the GPU. That is, all of the operations of CKKS are implemented here to run on GPUs, including the HE multiplication and bootstrapping steps, which are correspondingly the most frequently used primitive operation and the slowest one.

Through an analysis, we discover that RLWE-based FHE operations have high memory bandwidth and capacity requirements. Most RLWE-based FHE schemes require many evaluation keys for manipulating ciphertexts (e.g., multiplication key and rotation keys); the key sizes are large as each is a ciphertext of a certain value related to the secret key. When examining the multiplication operation of CKKS (`HMULT`), we observe that the arithmetic intensity (OP/B, operation per byte) of `HMULT`'s primary functions is low. Hence, they are not bottlenecked by the limited number of arithmetic units (compute-bound) but by the limited memory bandwidth (*memory-bound*) on modern GPUs. In particular, Tensor-product and Inner-product operations during key-switching exhibit very low arithmetic intensity levels (lower than 3 OP/B in Figure 2(b)). We also identify that most existing performance optimization techniques incur further increases in the memory bandwidth and capacity requirements; these include RNS-decomposition [BEHZ16a], efficient mod-up/down operations [CHK+19], improved linear transformation [HHC19], and efficient bootstrapping [HK20]. In particular, using a large decomposition number (dnum) greatly increases the memory capacity and bandwidth requirement.

These key observations lead us to devise *memory-centric* optimizations to improve the performance of CKKS on GPUs substantially. First, we fuse the functions that compose an individual HE operation, such as `HMULT` (intra-HE-fusion), and those across HE operations (inter-HE-fusion), by which we reduce accesses to the main memory (*global memory*) on the GPU. Second, in contrast to earlier work that involves a tradeoff between the computation cost and the multiplicative level (the number of subsequent `HMULT` operations on a ciphertext) [HK20], we also consider the main-memory bandwidth and capacity requirements while finding the optimal dnum for `HMULT`.

Owing to the aforementioned memory-centric optimizations, our `HMULT` and rescaling implementations perform 7.02× and 1.36× better, respectively, against a recently published result [BHM+20] while using an identical GPU generation. We are the first to report the performance of bootstrapping in CKKS implemented on a GPU after applying recently proposed algorithm-level optimizations [HK20]. The proposed GPU implementation exceeds the speed of a single-thread CPU implementation by 257×. The wall-clock time for bootstrapping is reduced to 526.96ms in a 173-bit security parameter set with 65,536 slots and with 19-bit precision bit, which translates into 0.423us in terms of the amortized time per bit. Our result is superior by orders of magnitude to recent works that produce 298us [HK20] and 519us [CCS19] with 128-bit security parameter sets. To demonstrate the effectiveness of the proposed optimizations at an application level, we implement Logistic Regression identically to an earlier study [HHCP19] on a GPU. The evaluation confirms

that compared to the implementation on an 8-threaded CPU, there is a 40.0× improvement (from 31.0 to 0.775 seconds per iteration) in speed when training with the same data used in the aforementioned earlier work [HHCP19].

Finally, we propose what is termed the 'amortized FHE-mult' time as a new unit of measurement for comparing HE multiplication performance. Amortized FHE-mult refers to the average time required per multiplication when an unlimited number of `HMULT`s are possible by bootstrapping. We calculate this time by measuring the average `HMULT` time plus the bootstrapping time divided by the maximum number of subsequent multiplications between two bootstrapping operations. The amortized FHE-mult time of our implementation is 24.35 ms, with which we believe that practical privacy-preserving applications can be developed.

2 Background

2.1 Notation for Homomorphic Encryption

We follow the notation definition in an earlier study [HK20] with minor modifications. For a polynomial ring $\mathcal{R} = \mathbb{Z}[X]/(X^N + 1)$, where N is a power-of-two integer, we denote the residue ring modulo an integer q as $\mathcal{R}_q = \mathcal{R}/q\mathcal{R}$. χ_{key} is the distribution of the secret key and χ_{err} is the error distribution over $\mathcal{R}$. $a \xleftarrow{\$} S$ means that a is sampled from set S uniformly or distribution S. We denote closed and open intervals using parentheses and brackets (e.g., $[a, b) = \{n \in \mathbb{Z} | a \leq n < b\}$). $[\cdot]_q$ denotes the modular reduction by q, respectively. We represent an element in $\mathcal{R}$ as $m(X) = \sum_{i=0}^{N-1} m_i X^i$ and an element in $\mathcal{R}_q$ as $[m(X)]_q = \sum_{i=0}^{N-1} [m_i]_q X^i$. We extend this notation for the RNS basis. For $\mathcal{C}_i = \{q_0, \ldots, q_i\}$, where the $\{q_j\}_{0 \leq j \leq i}$ values are coprime to each other, we define $[s(X)]_{\mathcal{C}_i}$ as $([s(X)]_{q_0}, \cdots, [s(X)]_{q_i}) \in \prod_{j=0}^{i} \mathcal{R}_{q_j}$. We denote two disjoint subsets of set Y as $\mathcal{S}_Y$ and $\mathcal{S}'_Y$. $\odot$ denotes Hadamard multiplication. A concatenation of two vectors or sets is denoted as $||$. $L(\ell)$ represents the maximum (current) level of a ciphertext, the number of multiplications one can perform on the ciphertext without decryption. Let $\hat{q}_{\ell,j} = \prod_{i=0 \wedge i \neq j}^{\ell} q_j$, where we omit the first subscript ℓ when $\ell = L$.

2.2 Polynomial Arithmetic in CKKS

Each ciphertext of CKKS is represented as a pair of polynomials of degree $< N$ in $\mathcal{R}_Q = \mathbb{Z}_Q[X]/(X^N + 1)$ and a FHE operation consists of polynomial operations. Given that polynomial degree N is very high (e.g., degree $N = 2^{16}$ or 2^{17}) and the coefficient size is very big (e.g., $\log Q = 2,000$), this type of polynomial arithmetic is very costly in FHE. For example, FHE-mult [CLP17, Cry20, HS14] becomes slower by a factor of thousands in terms of latency, compared to the native multiplication in the unencrypted domain.

Due to the high multiplication cost between polynomials, RLWE-based FHE schemes [BEHZ16a, CHK+19, BPA+19] typically have their own variants with the Residue Number System (RNS) [GHS12] to reduce their computational cost. First, by exploiting the Chinese Remainder Theorem (CRT), they represent each big-integer coefficient as a set of residues, each being a coefficient modulo a distinct prime number. Multiplication between two big integers is translated into point-wise multiplication between the two sets of residues, avoiding expensive multi-word arithmetic.

Second, they use Number Theoretic Transform (NTT) [AB75, Har14] and its inverse transformation (iNTT) for polynomial ring multiplication. NTT is a variant of Discrete Fourier Transform (DFT) [CCF+67] performed on a finite field of integers. Let ω_{q_i} be the primitive N-th root of unity in R_{q_i}. For a polynomial $[a(X)]_{q_i} \in \mathcal{R}_{q_i}$, the NTT function $\texttt{NTT}([a(X)]_{q_i})$ returns $([a(\omega_{q_i}^0)]_{q_i}, \cdots, [a(\omega_{q_i}^{N-1})]_{q_i}) \in (\mathbb{Z}_{q_i}^*)^N$. The inverse NTT function `iNTT` takes the output of `NTT` and returns $[a(X)]_{q_i}$.

A naïve multiplication of the two polynomials $a(X) = \sum_{i=0}^{N-1} a_i X^i, b(X) = \sum_{i=0}^{N-1} b_i X^i \in \mathcal{R}_{q_i}$ outputs a polynomial $c(X) = \sum_{i=0}^{N-1} c_i X^i \in \mathcal{R}_{q_i}$ whose coefficients $(c_0, \ldots, c_{N-1})$ are a negacylic convolution of two sequences $(a_0, \ldots, a_{N-1})$ and $(b_0, \ldots, b_{N-1})$. The negacyclic convolution is replaced with two NTTs performed on each integer sequence, one element-wise modular multiplication between the two sequences in the NTT domain and one iNTT on the output of the element-wise multiplication. Because the computational complexity levels of NTT and iNTT are $\mathcal{O}(N \log N)$ and the element-wise multiplication is $\mathcal{O}(N)$, the total complexity is reduced from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log N)$.

For a polynomial with a RNS basis $\mathcal{C}_i = \{q_0, ..., q_i\}$, NTT and iNTT are applied to each modulus. We extend the notations $\texttt{NTT}(\cdot)$ and $\texttt{iNTT}(\cdot)$; given $[a(X)]_{\mathcal{C}_i}$, we denote $[a]_{\mathcal{C}_i} = \texttt{NTT}([a(X)]_{\mathcal{C}_i}) = (a^{(0)}, a^{(1)}, \cdots, a^{(i)})$, where $a^{(j)} = \texttt{NTT}([a(X)]_{q_j})$. Similarly, we denote $\texttt{iNTT}([a]_{\mathcal{C}_i}) = [a(X)]_{\mathcal{C}_i}$. Then, $[a(X)]_{\mathcal{C}_i} \cdot [b(X)]_{\mathcal{C}_i}$ equals $\texttt{iNTT}([a]_{\mathcal{C}_i} \odot [b]_{\mathcal{C}_i})$. Henceforth, we omit the subscript for simplicity (e.g., $a \odot b$).

2.3 RNS Basis Conversion and Decomposition

We target a recent RNS variant of the CKKS scheme [HK20]; it is an improved version of [CHK+19], which reduces the operation complexity by adopting a generalized key-switching technique through RNS decomposition. Throughout this paper, we refer to this scheme as *CKKS*.

We describe RNS basis decomposition for the generalized key-switching and the basis conversion in CKKS. Fast basis conversion (`Conv`) converts the RNS basis of a polynomial into a new basis that is coprime to the original basis [CHK+19, BEHZ16a]. It is used in approximate modulus raising (`ModUp`) and approximate modulus reduction (`ModDown`), extending and shrinking the RNS basis of a polynomial, respectively.

Let $\mathcal{B} = \{p_0, \ldots, p_{k-1}\}$, $\mathcal{C}_i = \{q_0, \ldots, q_i\}$ for $i \in [0, L]$, where $\{p_j\}_{j \in [0,k)}, \{q_j\}_{j \in [0,L]}$ are coprime to each other. Let $\hat{Q}_j''' = \prod_{q_i \in \mathcal{S}_{\mathcal{C}_L}} q_i \times \prod_{p_i \in \mathcal{S}_\mathcal{B} \wedge i \neq j} p_i$ and $\hat{Q}_j'' = \prod_{q_i \in \mathcal{S}_{\mathcal{C}_L} \wedge i \neq j} q_i \times \prod_{p_i \in \mathcal{S}_\mathcal{B}} p_i$. `Conv`(see Algorithm 1) changes the basis of an input polynomial $[a(X)]_{\mathcal{S}_{\mathcal{C}_L} \cup \mathcal{S}_\mathcal{B}}$ to $\mathcal{S}'_{\mathcal{C}_L} \cup \mathcal{S}'_\mathcal{B}$ approximately [BEHZ16a, CHK+19].

Given an integer `dnum`, let $\alpha = (L+1)/\texttt{dnum}$ (for simplicity, we assume that $(L+1)$ is divided into `dnum`). The parameter `dnum` divides the basis $\mathcal{C}_L$ into the bases $\{\mathcal{C}'_i = \{q_j\}_{j \in A_i = [i\alpha, i\alpha+\alpha)}\}$, each having α primes [HK20].

For a given level ℓ of a ciphertext, let β be $\lceil (\ell+1)/\alpha \rceil \leq \texttt{dnum}$. Before key-switching, CKKS [HK20] first decomposes the RNS basis $\mathcal{C}_\ell$ of a polynomial into $\{\mathcal{C}'_i\}_{i \in [0,\beta)}$, each of which is then extended to basis $\mathcal{D}_\beta$ where $\mathcal{D}_i$ is defined as $\mathcal{B} \bigcup (\bigcup_{0 \leq j < i} \mathcal{C}'_j)$. After key-switching, the basis is reduced to $\mathcal{C}_\ell$.

We define $P = \prod_{i=0}^{k-1} p_i$, $Q' = \prod_{i=\ell+1}^{\alpha\beta-1} q_i$, $\hat{Q} = PQ'$, $\{Q'_j = \prod_{i=j\alpha}^{(j+1)\alpha-1} q_i\}_{j \in [0,\texttt{dnum})}$, and $\hat{Q}_j = \prod_{i=0 \wedge i \neq j}^{\texttt{dnum}-1} Q'_i$. The decomposition algorithm (`Dcomp`), `ModUp`, and `ModDown` are shown in Algorithm 3, Algorithm 2, and Algorithm 4, respectively.

Please refer to Appendix D for more details.

Algorithm 1: $\texttt{Conv}_{\mathcal{S}_{\mathcal{C}_L} \cup \mathcal{S}_\mathcal{B} \to \mathcal{S}'_{\mathcal{C}_L} \cup \mathcal{S}'_\mathcal{B}}([a(X)]_{\mathcal{S}_{\mathcal{C}_L} \cup \mathcal{S}_\mathcal{B}})$ (Fast basis conversion)

1 **for** $q_i \in \mathcal{S}'_{\mathcal{C}_L} \cup \mathcal{S}'_\mathcal{B}$ // we use q_i to represent all the elements in $\mathcal{S}'_\mathcal{B}$
2 $\quad [\tilde{a}(X)]_{q_i} \leftarrow \left[\sum_{q_j \in \mathcal{S}_{\mathcal{C}_L}} [[a(X)]_{q_j} \cdot \hat{Q}_j''^{-1}]_{q_j} \cdot [\hat{Q}_j'']_{q_i} \right]_{q_i}$
3 $\quad [\bar{a}(X)]_{q_i} \leftarrow \left[\sum_{p_j \in \mathcal{S}_\mathcal{B}} [[a(X)]_{p_j} \cdot \hat{Q}_j'''^{-1}]_{p_j} \cdot [\hat{Q}_j''']_{q_i} \right]_{q_i}$
4 $\quad [\tilde{a}(X)]_{q_i} \leftarrow [\tilde{a}(X)]_{q_i} + [\bar{a}(X)]_{q_i}$
5 **end for**
6 **return** $[\tilde{a}(X)]_{\mathcal{S}'_\mathcal{B} \cup \mathcal{S}'_{\mathcal{C}_L}} = [\tilde{a}(X)]_{\mathcal{S}'_\mathcal{B}} || [\tilde{a}(X)]_{\mathcal{S}'_{\mathcal{C}_L}}$

Algorithm 2: $\texttt{ModUp}_{\mathcal{C}'_i \to \mathcal{D}_\beta}([a]_{\mathcal{C}'_i})$

1 $([a(X)]_{\mathcal{C}'_i}) \leftarrow \texttt{iNTT}([a]_{\mathcal{C}'_i})$
2 $(\tilde{a}^{(j)})_{j \in A_i} = (a^{(j)})_{j \in A_i}$
3 $[\tilde{a}(X)]_{\mathcal{D}_\beta - \mathcal{C}'_i} \leftarrow \texttt{Conv}_{\mathcal{C}'_i \to \mathcal{D}_\beta - \mathcal{C}'_i}\ ([a(X)]_{\mathcal{C}'_i})$
4 $(\tilde{a}^{(j)})_{j \in ([0,k+\alpha\beta-1]-A_i)} \leftarrow \texttt{NTT}([\tilde{a}(X)]_{\mathcal{D}_\beta - \mathcal{C}'_i})$
5 **return** $[\tilde{a}]_{\mathcal{D}_\beta} = (\tilde{a}^{(0)}, \cdots, \tilde{a}^{(k+\alpha\beta-1)})$

Algorithm 3: $\texttt{Dcomp}(d = (d^{(0)}, \cdots, d^{(\ell)}))$

1 $d^{(j)} \leftarrow 0\ \forall j \in [\ell+1, \alpha\beta-1]$
2 $d_j^{(i)} \leftarrow d^{(j\alpha+i)} \cdot [Q']_{q_{j\alpha+i}} \cdot [\hat{Q}_j^{\,-1}]_{q_{j\alpha+i}} \quad \forall i \in [0, \alpha-1], \forall j \in [0, \beta-1].$
3 $d_j \leftarrow (d_j^{(i)})_{i \in [0,\alpha-1]}\ \forall j \in [0, \beta-1]$
4 **return** $\vec{d} = (d_j)_{j \in [0,\beta-1]}$

Algorithm 4: Approximate Modulus Reduction

1 **procedure** $\texttt{ModDown}_{\mathcal{D}_\beta \to \mathcal{C}_\ell}(\tilde{b}^{(0)}, \tilde{b}^{(1)}, \ldots, \tilde{b}^{(k+\alpha\beta-1)})$
2 $\quad [\tilde{b}]_{\mathcal{D}_\beta - \mathcal{C}_\ell} \leftarrow (\tilde{b}^{(0)}, \ldots, \tilde{b}^{(k-1)}, \tilde{b}^{(k+\ell)}, \ldots, \tilde{b}^{(k+\alpha\beta-1)})$
3 $\quad [\tilde{b}(X)]_{\mathcal{D}_\beta - \mathcal{C}_\ell} \leftarrow \texttt{iNTT}([\tilde{b}]_{\mathcal{D}_\beta - \mathcal{C}_\ell})$
4 $\quad [\tilde{a}(X)]_{\mathcal{C}_\ell} \leftarrow \texttt{Conv}_{\mathcal{D}_\beta - \mathcal{C}_\ell \to \mathcal{C}_\ell}([\tilde{b}(X)]_{\mathcal{D}_\beta - \mathcal{C}_\ell})$
5 $\quad [\tilde{a}]_{\mathcal{C}_\ell} \leftarrow \texttt{NTT}([\tilde{a}(X)]_{\mathcal{C}_\ell})$
6 $\quad$ **for** $0 \le j \le \ell$ **do**
7 $\quad\quad b^{(j)} = [\hat{Q}^{-1}]_{q_j} \cdot \left(\tilde{b}^{(k+j)} - \tilde{a}^{(j)}\right) \pmod{q_j}$
8 $\quad$ **end**
9 $\quad$ **return** $(b^{(0)}, \ldots, b^{(\ell)})$
10 **end procedure**

2.4 Description of CKKS operations

We compactly describe CKKS [HK20] and its core operations. Let $Q_i = \prod_{j=0}^{i} q_j$. Q_L is termed the *ciphertext modulus*. CKKS encodes message $\vec{z}$, a vector of $N/2$ complex numbers, into a polynomial $m(X) \in \mathcal{R}_Q$, called *plaintext*. We refer to each position of a complex value in a message as a *slot*. The encoding step is as follows: (1) performing the inverse Discrete Fourier Transform (iDFT) on $\vec{z}$, (2) extracting the real and imaginary parts to concatenate the two, and (3) scaling up by multiplying with a scalar value Δ (called the *scaling factor*), followed by a rounding operation. A larger Δ increases the message precision but also requires larger primes $q_i \in \mathcal{C}_L$, lowering the security bit with others being equal. Typical values of Δ range from 2^{40} to 2^{50}.

Let a *secret key* be $s(X) \xleftarrow{\$} \chi_{\mathsf{key}}$ and a plaintext be $m(X)$, whose corresponding NTT representations are $s = [s]_{\mathcal{C}_L} \leftarrow \mathsf{NTT}([s(X)]_{\mathcal{C}_L})$ and $m = [m]_{\mathcal{C}_L} \leftarrow \mathsf{NTT}([m(X)]_{\mathcal{C}_L})$. Let an evaluation key be $\mathbf{evk} = (\mathbf{evk}_i)_{i \in [0,\mathtt{dnum})}$, where $\mathbf{evk}_i = (evk_i^{(j)} = (a_{evk_i}^{(j)}, b_{evk_i}^{(j)}))_{j \in [0,k+L]} \in \prod_{n=0}^{k-1}((\mathbb{Z}_{p_n}^*)^N)^2 \times \prod_{n=0}^{L}((\mathbb{Z}_{q_n}^*)^N)^2$ such that $(a_{evk_i}^{(j)}, b_{evk_i}^{(j)})$ are a pair of polynomials in the NTT domain.

We represent a level ℓ ciphertext $\mathbf{ct}$ as a pair of polynomials in the NTT domain $(a, b) \in (\prod_{j=0}^{\ell} (Z_{q_\ell}^*)^N)^2$ rather than the RNS domain $([a(X)]_{\mathcal{C}_\ell}, [b(X)]_{\mathcal{C}_\ell})$. CKKS encrypts

$m(X)$ as a level L ciphertext $\mathbf{ct} = (a, b) \leftarrow \texttt{RLWE}(s, m)$[1], where $(a \stackrel{\$}{\leftarrow} \prod_{j=0}^{L}(\mathbb{Z}_{q_j}^*)^N$, $b \leftarrow a \odot s + e + m)$ and $e \stackrel{\$}{\leftarrow} \chi_{\mathsf{err}}$. More information is provided in Subsection D.2.

An FHE-mult between two ciphertexts (HMULT) followed by *rescaling*, a function adjusting the scaling factor Δ of the message, reduces one level of the ciphertext. For an HE circuit reducing the input ciphertext level by d, we refer to d as the *multiplicative depth*.

We provide the details of HMULT and the other FHE operations in Algorithm 5 - 9. $\texttt{CMULT}(\mathbf{ct}, m)$ returns (ma,mb), where $(a, b) \leftarrow \mathbf{ct}$. HMULT (HADD) multiplies (adds) $\mathbf{ct}_0$ by (and) $\mathbf{ct}_1$. RESCALE performs the rescaling operation for $\mathbf{ct}$. $\texttt{HROTATE}(\mathbf{ct}, sn, ebk)$ rotates the message of $\mathbf{ct}$ according to the rotation index sn, which is translated into a Frobenius map in the corresponding plaintext: $m(X) \rightarrow m(X^{5^{sn}})$. The function $\texttt{FrobeniusMap}(m, sn)$ performs $m(X) \mapsto m(X^{5^{sn}})$ in the NTT domain [HK20]. For the given $\{m^{(i)} = (m_j^{(i)})_{j\in[0,N-1]}\}_{i\in[0,\ell]}$, it generates $\{m'^{(i)} = ((m^{(i)}_{\pi_n^{-1}(j)})_{j\in[0,N-1]}\}_{i\in[0,\ell]}$, where $\pi_n(x) = ([5^n(2x+1)]_{2N} - 1)/2$, which is a permutation. KeySwitch decomposes the input polynomial $[d]_{\mathcal{C}_\ell}$ into $[d_j]_{\mathcal{C}'_j}$, where $j \in [0, \beta)$, extends the moduli of the decomposed parts using ModUp, multiplies them by an evaluation key, and finally reduces the moduli to the original level ℓ using ModDown. Throughout the paper, we refer to step 3 in Algorithm 9 as the *Inner-product*.

Algorithm 5: $\texttt{HMULT}(\mathbf{ct}_0,\mathbf{ct}_1,\mathbf{evk})$

1. $\mathbf{ct}_0 \rightarrow (a_0, b_0)$,$\mathbf{ct}_1 \rightarrow (a_1, b_1)$
2. $d_2 \leftarrow (a_0 \odot a_1)$, $d_0 \leftarrow (b_0 \odot b_1)$
3. $d_1 \leftarrow (a_1 \odot b_0 + b_1 \odot a_0)$
4. $(c'_0, c'_1) \leftarrow \texttt{KeySwitch}(d_2, \mathbf{evk})$
5. **return** $\mathbf{ct}_{\mathsf{mult}} = (d_1 + c'_0,\ d_0 + c'_1)$

Algorithm 6: $\texttt{HROTATE}(\mathbf{ct},sn,\mathbf{evk})$

1. $\mathbf{ct} \rightarrow (a, b)$
2. $a' \leftarrow \texttt{FrobeniusMap}(a,sn)$
3. $b' \leftarrow \texttt{FrobeniusMap}(b,sn)$
4. $(a'', b'') \leftarrow \texttt{KeySwitch}(a',\mathbf{evk})$
5. **return** $\mathbf{ct}' = (a'', b' + b'')$

Algorithm 7: $\texttt{HADD}(\mathbf{ct}_0,\mathbf{ct}_1)$

1. $\mathbf{ct}_0 \rightarrow (a_0, b_0)$,$\mathbf{ct}_1 \rightarrow (a_1, b_1)$
2. $d_0 \leftarrow (a_0 + a_1)$
3. $d_1 \leftarrow (b_0 + b_1)$
4. **return** (d_0, d_1)

Algorithm 8: $\texttt{RESCALE}(\mathbf{ct})$

1. $\mathbf{ct} \rightarrow ([a]_{\mathcal{C}_\ell}, [b]_{\mathcal{C}_\ell})$
2. $a'^{(j)} \leftarrow [q_\ell^{-1}(a^{(j)} - \texttt{NTT}([\texttt{iNTT}(a^{(\ell)})]_{q_j}))]_{q_j, j\in[0,\ell-1]}$
3. $b'^{(j)} \leftarrow [q_\ell^{-1}(b^{(j)} - \texttt{NTT}([\texttt{iNTT}(b^{(\ell)})]_{q_j}))]_{q_j, j\in[0,\ell-1]}$
4. **return** $([a']_{\mathcal{C}_{\ell-1}}, [b']_{\mathcal{C}_{\ell-1}})$

Algorithm 9: $\texttt{KeySwitch}([d]_{\mathcal{C}_\ell},\mathbf{evk})$ performs key-switching over d

1. $\vec{d} \leftarrow \texttt{Dcomp}(d)$, $(d_j)_{j\in[0,\beta-1]} \leftarrow \vec{d}$
2. $[\tilde{d}_j]_{\mathcal{D}_\beta} = (\tilde{d}_j^{(0)}, \tilde{d}_j^{(1)}, \cdots, \tilde{d}_j^{(k+\alpha\beta-1)}) \leftarrow \texttt{ModUp}([d_j]_{\mathcal{C}'_j})$ for $j \in [0, \beta-1]$
3. $([c_0]_{\mathcal{D}_\beta}, [c_1]_{\mathcal{D}_\beta})$, where $(c_0^{(i)}, c_1^{(i)}) = \Sigma_{j=0}^{\beta-1} \tilde{d}_j^{(i)} \odot \mathbf{evk}_j^{(i)}$ for $i \in [0, k+\alpha\beta-1]$
4. $([c_0]_{\mathcal{C}_\ell}, [c_1]_{\mathcal{C}_\ell}) \leftarrow (\texttt{ModDown}([c_0]_{\mathcal{D}_\beta}), \texttt{ModDown}([c_1]_{\mathcal{D}_\beta}))$
5. **return** $([c_0]_{\mathcal{C}_\ell}, [c_1]_{\mathcal{C}_\ell})$

2.5 Bootstrapping in CKKS

Before the level of a ciphertext is depleted by consecutive operations, *bootstrapping* must be performed on the ciphertext to increase its level and to allow more FHE operations on the ciphertext. We briefly explain the CKKS bootstrapping algorithm here [CHK+18, HK20].
Modulus Raising: Let $\mathbf{ct}$ be a ciphertext given by encrypting the plaintext $m(X)$. Let the current modulus of $\mathbf{ct}$ be $q = q_0$ having a zero level. Our goal is to increase the modulus (or, level). First, the modulus of $\mathbf{ct}$ is increased to Q_L, the modulus of a freshly

[1] The process of encryption using a public key rather than a private key is different from this, but the form of the result is the same. However, there is a difference that the size of the included error increases.

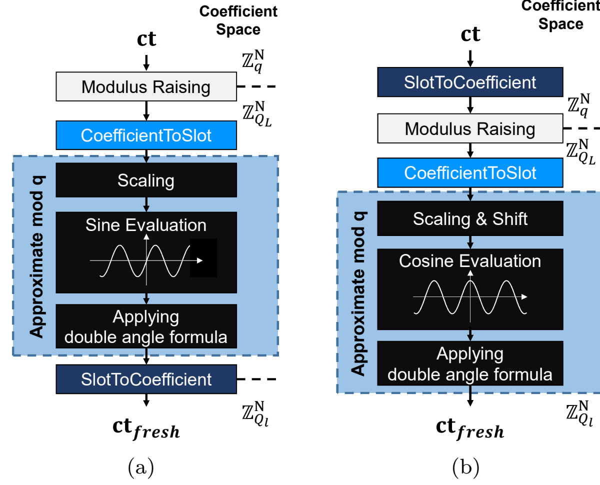


Figure 1: Flow diagram of (a) the original CKKS bootstrapping algorithm [CHK+18] and (b) ours adopting cosine evaluation in [HK20] and slim bootstrapping in [CH18].

encrypted ciphertext, producing $\mathbf{ct}'$. Although the level is increased, it adds the error polynomial $q \cdot I(X)$ to the plaintext: $Decrypt(\mathbf{ct}') = t(X) = m(X) + q \cdot I(X)$, where $I(X)$ is a polynomial whose coefficients are integers bounded to a small integer K, which is determined by the secret key distribution. To remove the error, we apply the approximate modulo operation in a homomorphic manner.

CoefficientToSlot: Let the plaintext $t(X) = t_0 + t_1X + t_2X^2 + \ldots + t_{N-1}X^{N-1}$ be the decryption of $\mathbf{ct}'$ in Modulus Raising and its decoding be $\vec{z}$. The goal of CoefficientToSlot is to compute $\mathbf{ct_1}$ and $\mathbf{ct_2}$, which contain messages $\vec{z}_1 = (t_0, t_1, \ldots, t_{N/2-1})$ and $\vec{z}_2 = (t_{N/2}, t_{N/2+1}, \ldots, t_{N-1})$, respectively. $\mathbf{ct}_1$ and $\mathbf{ct}_2$ are computed by evaluating the encoding circuit, *the linear transformation on* $\mathbf{ct}$:

$$\vec{z}_1 = 1/N \cdot (\overline{\mathbf{V}}^T \vec{z} + \mathbf{V}^T \overline{\vec{z}}),\ \vec{z}_2 = 1/N \cdot (\overline{\mathbf{W}}^T \vec{z} + \mathbf{W}^T \overline{\vec{z}})$$

where

$$\mathbf{V} = \begin{pmatrix} 1 & \omega_0 & \cdots & \omega_0^{\frac{N}{2}-1} \\ 1 & \omega_1 & \cdots & \omega_1^{\frac{N}{2}-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_{\frac{N}{2}-1} & \cdots & \omega_{\frac{N}{2}-1}^{\frac{N}{2}-1} \end{pmatrix} \text{ and } \mathbf{W} = \begin{pmatrix} \omega_0^{\frac{N}{2}} & \omega_0^{\frac{N}{2}+1} & \cdots & \omega_0^{N-1} \\ \omega_1^{\frac{N}{2}} & \omega_1^{\frac{N}{2}+1} & \cdots & \omega_1^{N-1} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_{\frac{N}{2}-1}^{\frac{N}{2}} & \omega_{\frac{N}{2}-1}^{\frac{N}{2}+1} & \cdots & \omega_{\frac{N}{2}-1}^{N-1} \end{pmatrix}.$$

Instead of performing two linear transformations, the relationships $i\mathbf{V} = \mathbf{W}$ and $\overline{\overline{\mathbf{V}}^T \vec{z}} = \mathbf{V}^T \overline{\vec{z}}$ are exploited, requiring only one linear transform to produce $\overline{\mathbf{V}}^T \vec{z}$; the following homomorphic conjugation and addition/subtraction compute the other terms.

Approximated modulo operation: For the two ciphertexts $\mathbf{ct}_1$ and $\mathbf{ct}_2$, we want to perform modular reduction on each element of their messages: $(t_i \mod q)$ for all i. However, because the modular reduction operation is not available in FHE, twofold approximation of the modulo operation is used instead. First, the modulo q operation is approximated as a scaled sine function, such as $f(t) = q/2\pi \sin(2\pi t/q)$ [CHK+18]. This exploits the facts that each value t_i in message $\vec{z}_1$ and $\vec{z}_2$ is distributed near $q \cdot I$ for an integer $I \in (-K, K)$ for a small K, and the scaled sine function resembles the modulo operation near $q \cdot I$. Second, as the sine function is also not available in FHE, it is approximated as a polynomial. We adopted a recent bootstrapping algorithm from [HK20]. First, it approximates not the sine but the cosine function by shifting $\mathbf{ct}_1$ and $\mathbf{ct}_2$. Also, it uses polynomial interpolation with the Chebyshev polynomial basis to approximate the

cosine function in addition to adopting the double-angle formula. We use one of their parameter sets for bootstrapping; we evaluate a 31-degree polynomial and then apply the double-angle formula three times, approximating $f(t) = \cos(24\pi t)$. With applying the cosine evaluation to $\mathbf{ct}_1$ and $\mathbf{ct}_2$, we obtain two output ciphertexts $\mathbf{ct}'_1$ and $\mathbf{ct}'_2$ whose messages are $\vec{z}'_1$ and $\vec{z}'_2$, respectively. Our cosine evaluation step has a multiplicative depth of 11.

SlotToCoefficient: This is the opposite of CoefficientToSlot. With $\mathbf{ct}'_1$ and $\mathbf{ct}'_2$, we compute an output ciphertext, $\mathbf{ct}_{fresh}$, which contains message $\vec{z}_{fresh}$, by evaluating a linear transformation, as shown below:

$$\vec{z}_{fresh} = \mathbf{V}\vec{z}'_1 + \mathbf{W}\vec{z}'_2 = \mathbf{V}(\vec{z}'_1 + i\vec{z}'_2)$$

Here, $\vec{z}_{fresh}$ is approximately equal to the message of $\mathbf{ct}$ before Modulus Raising.

We also adopt the additional technique of *slim bootstrapping* from [CH18], which reorders the bootstrapping process (see Figure 1) from (Modulus Raising, CoefficientToSlot, Cosine evaluation, SlotToCoefficient) to (SlotToCoefficient, Modulus Raising, CoefficientToSlot, Cosine evaluation). By postponing Modulus Raising, the computational cost of SlotToCoefficient decreases as the ciphertext level of the input to SlotToCoefficient decreases.

We also use efficient CoefficientToSlot and SlotToCoefficient proposed in [HHC19]. That study adopts the Cooley-Tukey FFT algorithm [CT65] for linear transform in CoefficientToSlot and SlotToCoefficient. The DFT algorithm can be expressed in the form of multiplication between DFT matrix $\mathbf{D}$ and input vector $\vec{v}$. The computational complexity of directly computing $\mathbf{D} \cdot \vec{v}$ is $\mathcal{O}(N^2)$. By adopting the Cooley-Tukey FFT algorithm, $\mathbf{D}$ can be decomposed into $\log_2 N$ *block-diagonal sparse matrices*, reducing the total complexity to $\mathcal{O}(N \log N)$.

For a given radix r and degree N, they decompose $\mathbf{V}_{rev}$, which is a row-permuted $\mathbf{V}$, into $\log_r(N/2)$ block-diagonal sparse matrices $\{\mathbf{V}_i^{(r)}\}_{1\leq i\leq \log_r(N/2)}$ such that $\mathbf{V}_{rev} = \mathbf{V}_1^{(r)}\mathbf{V}_2^{(r)} \ldots \mathbf{V}_{\log_r(N/2)}^{(r)}$. Each decomposed block-diagonal sparse matrix $\mathbf{V}_i^{(r)}$ is a matrix having $2r-1$ (when $i > 1$) or r (when $i = 1$) *diagonals*, with all other element values being zero. We refer to each diagonal of index i from matrix $\mathbf{M}$ as a length-$N/2$ vector, $diag_i(\mathbf{M})$ such that $diag_i(\mathbf{M}) = (\mathbf{M}(0,i), \mathbf{M}(1,i+1), \ldots, \mathbf{M}(N/2-1, N/2+i-1))$ where $\mathbf{M}(i,j)$ refers to the (i,j) element of $\mathbf{M}$. The same decomposition technique applies to CoefficientToSlot as well. For more details, see Appendix B.

An important characteristic of the $(2r-1)$ (or, r) diagonals of each decomposed block-diagonal sparse matrix is that *their indices form an arithmetic progression.* Using this property, [HHC19] adopts the Baby-step Giant-step algorithm, as in [HS15, CH18, CHK+18], reducing the number of FHE rotations required during each multiplication between a block-diagonal sparse matrix and a vector that is encrypted as a ciphertext.

Baby-step Giant-step algorithm: Each multiplication between a block-diagonal sparse matrix and the input vector is performed in a homomorphic-friendly way, exploiting the Baby-step Giant-step algorithm (BSGS) [HS15, CH18, CHK+18]. BSGS computes the matrix-vector multiplication via the summation of the products of plaintexts, each being an encoded diagonal, and the shifted input vectors, each being the message of a correspondingly rotated ciphertext.

Let a predetermined block-diagonal sparse matrix with n diagonals be $\mathbf{M}$ and the input vector be $\vec{v}$, the message of the ciphertext $\mathbf{ct}$. Let $rot_i(\vec{v})$ be an input vector shifted by i, the message of `HROTATE(ct, i, evk)`. For l and t such that $l \cdot t = n$, BSGS computes $\mathbf{M} \cdot \vec{v}$ as shown below, where the common difference of the arithmetic progression is one:

$$\begin{aligned}\mathbf{M}\cdot\vec{v} &= \sum_{i=0}^{n-1} diag_i(\mathbf{M})\odot rot_i(\vec{v}) = \sum_{i=0}^{l-1}\sum_{j=0}^{t-1} diag_{ti+j}(\mathbf{M})\odot rot_{ti+j}(\vec{v}) \\ &= \sum_{i=0}^{l-1} rot_{ti}(\sum_{j=0}^{t-1} rot_{-ti}(diag_{ti+j}(\mathbf{M}))\odot rot_j(\vec{v}))\end{aligned} \tag{1}$$

Given a radix r, by choosing l and t near $\sqrt{n}$, the number of rotations required for each SlotToCoefficient and CoefficientToSlot is reduced from $\mathcal{O}(r\log_r n)$ to $\mathcal{O}(\sqrt{r}\log_r n)$, as each block-diagonal sparse matrix has $2r-1$ (or, r) diagonals. We can choose any l and t with flexibility by adding zero diagonals [HHC19].

3 CKKS: Baseline implementation

3.1 Basic HE operations

We provide here the pertinent details of the baseline CPU and GPU implementation of CKKS. For readers unfamiliar with contemporary GPU architectures, we recommend reading Appendix A. There are four groups of functions that compose operations in CKKS: (1) element-wise RNS operations between polynomials, such as multiplication in the NTT domain, addition, and subtraction; (2) NTT and iNTT; (3) fast basis conversion (`Conv`) used in `ModUp` and `ModDown` [BEHZ16a, CHK+19]; and (4) Inner-product in key-switching. Throughout this paper, we use double-word (i.e., 64-bit) moduli such that $\{q_i\}_{i\in[1,L]}$ are 52-bit and $\{q_0, p_0, ..., p_k\}$ are 62-bit. Also, we layout input and output data contiguously in memory in a degree-first manner (i.e., a chunk of N residues of $a^{(i)}$ are continuous for each i) so that the data become NTT-friendly [CLP17, HS14, CHK+19].

RNS operation: We refer to an *RNS operation* as a binary operation that takes residues as input and performs an element-wise operation on them, such as multiplication, addition, and subtraction. In our CPU implementation, each CPU thread takes two vectors with N residues (i.e., $a^{(i)}, b^{(i)} \in \mathcal{R}_{q_j}$) at a time and performs N RNS operations, as in [CLP17]. Then, the thread (or another thread in a multi-threaded, multi-core environment) takes another pair of two vectors with N residues, until $(\ell+1)\times N$ RNS operations are complete by all threads. In contrast, our GPU implementation follows an earlier method in [BPA+19]; we launch $(\ell+1)\times N$ threads for a single GPU kernel. Letting each GPU thread perform a single RNS operation, we exploit the massive thread-level parallelism available from modern GPUs, which can run hundreds of thousands of threads concurrently.

NTT and iNTT: For NTT and iNTT, in both CPU and GPU implementation, we exploit the David Harvey's NTT algorithm [Har14] along with the in-place Cooley-Tukey algorithm, which is commonly used in other works [CLP17, Sho16, HS14].

For the CPU implementation of NTT, we use the same approach as in the RNS operation, where each thread takes N residues (i.e., $a^{(i)}$ for a given i) at a time, and perform N-point NTT. For the GPU implementation, we use the hierarchical NTT implementation [KJPA20], which heavily exploits shared memory in GPUs while adopting an earlier approach in [GLD+08]. Specifically, for every (i)NTT with N residues, we use 8 per-thread (i)NTT kernels, as described in [KJPA20], where each thread in a kernel loads eight residues into the registers at a time. We launch kernels each performing radix-256 or radix-512 (i)NTT, where radix-k divides an N-point transformation into k interleaved N/k-point transformations. It uses shared memory as storage for the temporal output of each (i)NTT stage so that the N residues are loaded from and stored to global memory only once per kernel [GLD+08]. We launch two sequential GPU kernels for practical values of N ranging from 2^{16} to 2^{17}. See Appendix C for more details.

ModUp and ModDown: First, we explain the implementation of fast basis conversion. Given an input polynomial $a(X)$ and RNS bases $\mathcal{S}_1 = \mathcal{S}_{\mathcal{C}_L} \cup \mathcal{S}_{\mathcal{B}}$ and $\mathcal{S}_2 = \mathcal{S}'_{\mathcal{C}_L} \cup \mathcal{S}'_{\mathcal{B}}$,

Algorithm 10: Baseline implementation of Inner-product in key-switching

```
1  evk_i := (a_{evk_i}, b_{evk_i}), i = 0, 1, ..., β − 1
2  d := {d_2^(0), d_2^(1), ..., d_2^(β−1)}
3  procedure Inner-product(accum, evk, d)
4      accum = d_2^(0) · evk_0                          // partial sums
5      for i in [1, β) do
6          accum += d_2^(i) · evk_i
7      end
8      b'_accum = BarrettModReduction(b_accum)
9      a'_accum = BarrettModReduction(a_accum)
10     return (a'_accum, b'_accum)
11 end procedure
```

a fast basis conversion $\mathtt{Conv}_{\mathcal{S}_1\rightarrow\mathcal{S}_2}([a(X)]_{\mathcal{S}_1})$ consists of two steps. Step 1 multiplies each input residue by $\hat{Q}''^{-1}_j \mod q_j$ (or $\hat{Q}'''^{-1}_j \mod p_j$) for the corresponding j. Then, step 2 computes each output residue via a multiply-accumulate (MAC) operation on $|\mathcal{S}_1|$ input residues each multiplied by ($\hat{Q}''_j \mod q_j(p_j)$) or ($\hat{Q}'''_j \mod q_j(p_j)$) for the corresponding q_j (or p_j) $\in \mathcal{S}_1$, followed by a modular reduction with p_i (or q_i) $\in \mathcal{S}_2$ (see `Conv` in Subsection 2.3).

In the CPU implementation of fast basis conversion, each thread takes N residues that are coefficients of $[a(X)]_{q_j (\text{or } p_j)}$, where q_j (or p_j) $\in \mathcal{S}_1$ serve as inputs, and performs the step 1. After all threads complete the first step, step 2 begins, where each thread computes N output residues; the inner-most loop, which performs MAC followed by modular reduction, computes one output residue at a time.

The GPU implementation of the fast basis conversion consists of two kernels (see Appendix C for details). The first kernel (called the *scaling* kernel) for step 1 launches $|\mathcal{S}_1| \times N$ threads where each thread multiplies an input residue by a corresponding constant $\hat{Q}''^{-1}_j(\hat{Q}'''^{-1}_j) \mod q_j\ (p_j)$. The second kernel for step 2 launches $|\mathcal{S}_2| \times N$ threads where a thread takes $|\mathcal{S}_1|$ residues of a coefficient as inputs to compute one output. Throughout the accumulation, the partial sum resides in the registers held by the threads. When the accumulation process is completed, the thread performs modular reduction on the result. A prior work [BPA+19] implements fast basis conversion on a GPU but performs modular reduction right before the partial sum overflows; in contrast, we accumulate the partial sum into three double-word values (which are stored in the register file).

Both `ModUp` and `ModDown` exploit fast basis conversion, performing `iNTT` on the inputs of fast basis conversion and `NTT` on the outputs. The difference between them lies in the RNS operations coming after `NTT`. `ModUp` concatenates the output of `NTT` to its input with the original basis to extend the basis. `ModDown` (see Algorithm 4) subtracts the original input from the output of NTT and scales by $\hat{Q}^{-1} \mod q_j$ (line 7). Because the subtraction and scaling after NTT are element-wise RNS operations, our CPU and GPU implementations use an identical approach when executing them as explained above.

Inner-product in key-switching: The baseline implementation of Inner-product in key-switching is shown in Algorithm 10. It includes the lazy modular reduction technique adopted in FHE libraries [CLP17, Cry20]. This technique performs wide MAC operations (i.e., the partial sum being 128-bit) instead of doing modular reduction β times.

In our CPU implementation, a thread loads three vectors of N residues, one from $d_2^{(i)}$ and two from $\mathbf{evk}_i$ for $i \in [0, \beta)$ and then performs multiplication (line 4) or MAC (line 6) in an element-wise manner. This is repeated β times, followed by Barrett's modular reduction [Bar87] (lines 8 and 9). Our baseline GPU implementation launches three types of kernels: one for multiplication (line 4), another for MAC (line 6), and the third for

Algorithm 11: Evaluation of BSGS in Equation 1

```
1  ct :=(input ciphertext)
2  out :=(zero-initialized output ciphertext)
3  ptxt := {m_{i,j}}_{i∈[0,l),j∈[0,t)}          // precomputed plaintexts
4  procedure
5      for j in [1,t) do
6          ct_j = HROTATE(ct, j)
7      end
8      for i in [0,l) do
9          accum = CMULT(ct, m_{i,0})
10         for j in [1,t) do
11             temp = CMULT(ct_j, m_{i,j})
12             accum = HADD(temp, accum)
13         end
14         temp = HROTATE(accum, ti)
15         out = HADD(accum, out)
16     end
17     return out
18 end procedure
```

modular reduction (lines 8 and 9). Each of the three launches $(\alpha\beta + k)N$ threads. In the first and the second kernel, each thread takes three residues as inputs, one from $d_2^{(i)}$ and two from $\mathbf{evk}_i$. Then the thread multiplies the first input by the others, accumulates them into two 128-bit partial sums, and stores them as in the CPU implementation case. The last kernel reduces the 128-bit partial sums to 64-bits using Barrett's modular reduction.

3.2 Baby-step Giant-step

Let a matrix $\mathbf{M}$ be predetermined with l diagonals whose indices form arithmetic progression, and let vector $\vec{v}$ be encrypted as a ciphertext $\mathbf{ct}$. In CKKS, the evaluation of BSGS in Equation 1, where $rot_{-ti}(diag_{ti+j}(\mathbf{M}))$ is encoded as a plaintext $m_{i,j}$ for each $0 \le i < l, 0 \le j < t$, is done as shown in Algorithm 11. This requires a distinct rotation key for each rotation index j, but we skip the notation here for simplicity.

We describe the implementation of the HE operations required in BSGS. `CMULT` is performed as two element-wise RNS operations as described in the previous section. `HROTATE` is mostly similar to `HMULT` for its key-switching, except that it includes a Frobenius map (see Section 2), which is a permutation in the NTT domain. For the implementation of the permutation with index n, where the $\{j\}_{[0,N)}$-th residue of a given input $\{a^{(i)}\}_{[0,\ell]}$ is moved into the $\pi_n(j)$-th position, each CPU thread executes in-place permutation on N residues, whereas the GPU performs out-of-place permutation by launching $(\ell + 1) \times N$ threads such that the $(i \times N + j)$-th thread takes the j-th residue of $a^{(i)}$.

Hoisting: Halevi et al. [HS18] suggested reducing the computation cost of BSGS by applying *hoisting*. Hoisting reduces the required number of `ModUp` operations, when multiple `HROTATE` operations with different rotation indices are performed on the same ciphertext. Specifically, it restructures the `HROTATE` algorithm to perform `ModUp` first, before the Frobenius map. Below is the computation order in `HROTATE` with hoisting:

1. (`PrecomputeModUp`) Perform iNTT, fast basis conversion, and NTT on a of a ciphertext $\mathbf{ct} = (a, b)$.
2. (`FastRotate`) Perform the Frobenius map with rotation index i. Skip `ModUp`. Perform Inner-product, `ModDown`, and remaining functions required for `HROTATE`.

The modified `HROTATE` consists of two steps. When one needs to rotate a single ciphertext with different rotation indices, we perform `ModUp` only once on a ciphertext, after which we perform multiple `FastRotate` instead of `HROTATE`. The error bound of `FastRotate` remains identical to that of `HROTATE`; the error bound is proportional to the infinity norm $\|\tilde{a}\|_\infty$, where $\tilde{a}$ is the reconstructed output of `ModUp` [CHK+19], and hoisting preserves the infinity norm (i.e., it changes only the order of the coefficients).

By applying the hoisting, one `PrecomputeModUp` is required immediately before the line 5 of Algorithm 11, whereas `HROTATE` in line 6 changes to `FastRotate`, reducing the required number of `ModUp` operations from $(t-1)+(l-1)$ to $1+(l-1)$. We evaluate the performance improvement of bootstrapping with hoisting in Section 6.

4 Bottleneck Analysis of FHE-Mult

In this section, we show that the major FHE operations in CKKS are primarily limited by the global memory bandwidth of GPUs.

4.1 Function-level Operation Complexity and Memory Access Analysis

First, we show the operational complexity and the number of global memory accesses of each function during a multiplication of CKKS (`HMULT`). We evaluate the operational complexity in terms of the number of integer multiplications and additions, as in [TLW19]. First, we count the number of modular multiplications (*ModMuls*) as well as the other operations and then convert the ModMuls into the number of multiplications and additions. We count one Barrett's ModMul [Bar87] (Shoup's ModMul [Sho16]) as four (three) multiplications and three (two) additions, using the methodologies explained in [TLW19]. For simplicity, we assume that the two input ciphertexts have the same level ℓ, with $\beta = (\ell+1)/\alpha$ without the ceiling function.

Prior work [HK20] showed the operation complexity of `HMULT` in CKKS for various values of `dnum` in terms of the number of ModMuls and showed that it is practical to choose a moderate `dnum`; however, they did not consider the high memory bandwidth requirement, which directly affects into the performance of `HMULT`, especially on GPUs. We assume that each modulus $\{q_i\}_{0\le i\le L}$ and $\{p_j\}_{0\le j<k}$ is of the double-word type (8 bytes). We count the global memory accesses in unit of eight bytes. Also, we consider subtraction as signed addition.

Tensor-product: We exploit the Karatsuba algorithm [KO62], which is commonly used in many HE libraries [HS14, Cry20, CLP17]. It computes $a_1 \odot a_0, b_1 \odot b_0, (a_0+a_1) \odot (b_0+b_1)$ first and then computes (d_2, d_1, d_0) (see Subsection 2.4) from them, reducing the number of polynomial multiplications from 4 to 3 at cost of three additions, which is inexpensive. This results in $3(\ell+1)N$ ModMuls, $4(\ell+1)N$ additions in total. For the memory accesses, four input polynomials are loaded and three output polynomials are stored such that $4(\ell+1)N$ reads and $3(\ell+1)N$ writes take place.

RNS-Decomposition: There are $\alpha\beta N = (\ell+1)N$ ModMuls and $2(\ell+1)N$ memory accesses in total for reads and writes (see `Dcomp` in Subsection 2.3).

NTT & iNTT: For a single execution of (i)NTT on the N residues of each $\{a^{(i)}\}_{i\in[0,\ell]}$, there are $\log N$ stages, resulting in $(N \log N)/2$ ModMuls and $N \log N$ additions. The number of global memory accesses can be as high as $2N \log N$ for the loading and storing the input and output, respectively, and N for the loading twiddle factors.

However, there are two more factors that affect the total number of memory accesses: the cache memory size of the processor and Shoup's ModMul [Sho16]. If the cache is large enough to accommodate the length-N input of (i)NTT, it does not have to access the global memory at each stage, reducing the memory accesses by up to a factor of $1/\log N$. Modern server-class CPUs have last-level cache memory of dozens of megabytes [AFK+19],

Table 1: Operation complexity and memory accesses required in `HMULT` with ciphertexts having level $l < L$ and degree N. We converted the number of ModMuls into multiplications (muls) and additions (adds) to count total integer operations. $\alpha = (L+1)/\texttt{dnum}$ and $\beta = (\ell+1)/\alpha$.

	# of ModMuls	**# of muls and adds**	**# of total integer ops**	**Memory accesses (each 8B)**
Tensor-product	$3(\ell+1)N$	$4(\ell+1)N$	$25(\ell+1)N$	$7(\ell+1)N$
RNS-decomposition	$(\ell+1)N$	-	$7(\ell+1)N$	$2(\ell+1)N$
NTT	$\beta(\alpha\beta+k)N(\log N)/2$	$\beta(\alpha\beta+k)N(\log N)$	$7\beta(\alpha\beta+k)N(logN)/2$	$\beta(\alpha\beta+k)(2N+N\log N)$
iNTT	$(2\alpha\beta+k)N(\log N)/2$	$(2\alpha\beta+k)N(\log N)$	$7(2\alpha\beta+k)N(\log N)/2$	$(2\alpha\beta+k)(2N+N\log N)$
Inner-product	$2(\alpha\beta+k)N$	$(4\beta-2)(\alpha\beta+k)N$	$(4\beta+12)(\alpha\beta+k)N$	$(11\beta-4)(\alpha\beta+k)N$
Conv (ModUp)	$\beta(\alpha\beta+k-\alpha)N$	$2\alpha\beta(\alpha\beta+k-\alpha)N$	$(7\beta+2\alpha\beta)(\alpha\beta+k-\alpha)N$	$(\alpha\beta+\alpha\beta^2+k\beta)N$
Conv (ModDown)	$2\alpha\beta N$	$4k\alpha\beta N$	$(4k+14)\alpha\beta N$	$2(\alpha\beta+k)N$

easily achieving such a reduction in memory accesses. In contrast, GPUs have a much smaller last-level cache of a few megabytes [NVI17] and can scarcely accommodate the input/output of (i)NTT. Therefore, the number of memory accesses required on a GPU to load an input polynomial of (i)NTT is between $2N\log N$ and $2N$.

Our baseline (i)NTT implementation [KJPA20] launches two GPU kernels, each of which performs radix-$\sqrt{N}$ (i)NTT, resulting in $2N\log N/\log\sqrt{N}$ accesses for the input and output. Rather than Barrett's algorithm [Bar87], we adopt Shoup's ModMul, as implemented in [KJPA20], which is commonly used for (i)NTT to reduce the operational complexity of ModMuls [CLP17, HS14]. Using Shoup's method adds extra N memory accesses as it demands a precomputed value for each ModMul. iNTT is performed at the front of `ModUp` and `ModDown`, each being done β times for αN residues and once for $(\alpha\beta+k)N$ residues (see Subsection 2.3). NTT is performed on the output of the basis conversion in `ModUp` and `ModDown`, each done β times for $(\alpha\beta+k-\alpha)N$ residues and once for $\alpha\beta N$ residues.

ModUp: Because we count the operations in (i)NTT separately, we consider only fast basis conversion in `ModUp` and `ModDown` here. With the lazy modular reduction technique, one needs to perform modular reduction only once after computing a sum of product, as stated in Section 3. Therefore, $\alpha\beta(\alpha\beta+k-\alpha)N$ multiplications, $\alpha\beta(\alpha\beta+k-\alpha)N$ additions, and $\beta(\alpha\beta+k-\alpha)N$ modular reductions are required in total. The number of memory accesses is $\alpha\beta N$ for reads and $\beta(\alpha\beta+k)N$ for writes in total.

Inner-product: As in `ModUp`, lazy modular reduction is applied to Inner-product in key-switching, resulting in $2\beta(\alpha\beta+k)N$ multiplications, $2(\beta-1)(\alpha\beta+k)N$ additions, and $2(\alpha\beta+k)N$ modular reductions. The required memory accesses are $2\beta(\alpha\beta+k)N$ for loading $\{\mathbf{evk}_i\}_{0\le i<\beta}$, $\beta(\alpha\beta+k)N$ for loading $\{d_{2,i}\}_{0\le i<\beta}$, and finally, $4(\beta-1)(\alpha\beta+k)N$ and $4\beta(\alpha\beta+k)N$ for loading and storing **accum** in Algorithm 10, respectively.

ModDown: There are $k(\alpha\beta)N$ multiplications, $k(\alpha\beta)N$ additions, and $(\alpha\beta)N$ modular reductions for each of the two polynomials (a'_{accum} and b'_{accum}), which are the output of **Inner-product**. Meanwhile, there exist kN memory accesses for the loading of the input and $(\alpha\beta+k)N$ for the storing of the output for each of the two polynomials.

We summarize the overall operational complexity and number of memory accesses required in Table 1.

For given values of L, ℓ, and N, the value of `dnum` that minimizes the total number of modular multiplications and plain multiplications is $(L+1)\sqrt{8\ell+7\log N+32}$ $/((\ell+1)\sqrt{7\log N+22})$. The global memory access count is minimized when `dnum` $= ((L+1)\sqrt{logN})/((\ell+1)\sqrt{logN+14})$. Prior work [HK20] chooses `dnum` that minimizes the number of ModMuls. However, this could be sub-optimal if an FHE operation is mostly bottlenecked by the global memory bandwidth, which is especially true in GPUs compared to CPUs. Figure 2(a) shows the `HMULT` time with various `dnum` values on our baseline CKKS implementation, both with CPUs and GPUs. As stated above, because modern server-class CPUs have a last-level cache of several dozens of megabytes that can accommodate one or two polynomials in FHE operations, the last-level cache absorbs

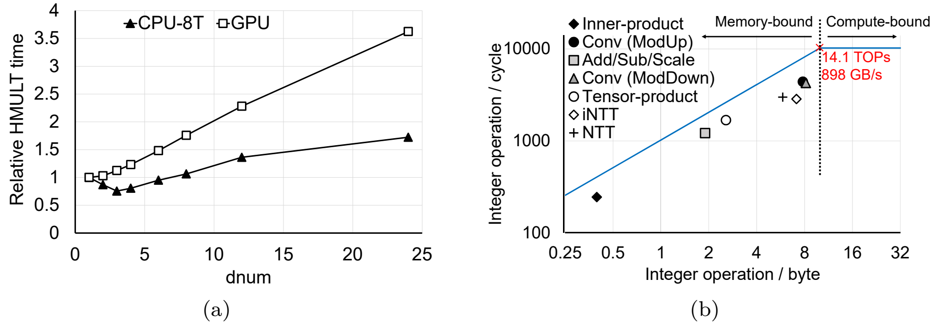


Figure 2: (a) HMULT time with 8-threaded CPU (CPU-8T) and our baseline GPU implementation when $(N, L) = (65536, 24)$. (b) A roofline model of multiplication in CKKS with our baseline GPU implementation when $(N, L, \texttt{dnum}) = (2^{16}, 45, 45)$. The number of integer instructions is measured with NVIDIA Nsight compute [NVI20].

most of the DRAM traffic. This makes the execution time of HMULT directly proportional to the number of arithmetic operations. In contrast, GPUs have smaller caches and the execution time is for this reason determined instead by the global memory accesses, becoming susceptible to the memory bandwidth and making it preferable to use a smaller dnum for fewer memory accesses.

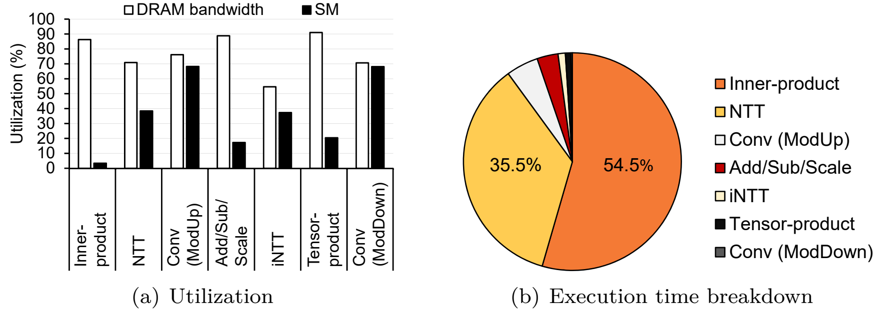
4.2 Memory-bandwidth Bottleneck of FHE Operations on GPUs

A single ModMul operation is translated into different instructions depending on the type of machine. To understand the actual operational characteristics of HMULT on GPUs, we provide a roofline plot [HP17] of CKKS multiplication in our baseline GPU implementation in Figure 2(b). We measured the integer operation throughput and DRAM access counts using a profiling tool provided by NVIDIA Nsight Compute [NVI20].

Most operations in HMULT are not compute-bound but are instead global-memory-bandwidth bound. Among these operations, Inner-product has the lowest operation intensity, severely bounded by the global memory bandwidth. The operational intensity of Inner-product is even lower than that of the addition operations because the baseline Inner-product implementation accumulates the i-th product $d_2^{(i)} \cdot \mathbf{evk}_i$ into **accum** (lines 4-7 in Algorithm 10), whose elements are 128 bits (quad-word) long, incurring numerous memory accesses.

We observe that Inner-product in key-switching is a major bottleneck when attempting to accelerate HMULT because it has an extremely low operational intensity despite its large number of memory accesses, especially with a large dnum. Figure 3 presents the execution time, global memory (DRAM) bandwidth, and instruction throughput (SM utilization) of HMULT with our baseline GPU implementation. For most functions, these DRAM bandwidth utilization rates are higher than those of SM utilization, in good agreement with the low operational intensity shown in the roofline plot (Figure 2)(b). Also, although NTT performs most of the modular multiplications required in HMULT, Inner-product dominates by 54.5% the overall execution time. Because Inner-product is the most memory-intensive operation in HMULT, it utilizes SM in GPU at only 3.4%. The second-most time-consuming function, NTT, also exhibits high global memory utilization (71.0%) and low SM utilization (38.4%) rates.

One of the factors contributing to the large number of memory accesses in Inner-product is the use of a large dnum, as the sizes of the evaluation keys are proportional to dnum ($\{\mathbf{swk}_i\}_{0 \le i < \texttt{dnum}}$). Throughout this paper, we adopt the generalized key switching strategy in [HK20], which prefers the use of smaller dnum values, reducing the sizes of the keys and thus amortizing the global memory bandwidth bottleneck of Inner-product when run on a GPU. Moreover, we apply memory-centric optimizations that are commonly applicable



(a) Utilization (b) Execution time breakdown

Figure 3: Execution time, DRAM utilization, and SM utilization of a single FHE-mult on our baseline GPU implementation of CKKS with (N, L, $\texttt{dnum}$) = (2^{16}, 45, 45).

to GPUs, such as operation fusion (kernel fusion [QRHT19]) in both an intra- and an inter-FHE manner, which significantly reduces the bandwidth requirement.

5 Memory-centric optimizations for FHE on GPUs

Kernel fusion is a technique that fuses multiple GPU kernels into one kernel. Typical GPU kernels operate in three steps: they load the input data from DRAM into shared memory or registers, compute the data, and store the output data. Kernel fusion combines multiple kernels, and by reusing data in the register file or shared memory, this strategy reduces the number of global memory accesses between kernels. By judiciously applying various kernel fusion methods both in an intra- and inter-FHE-operation manner, we significantly reduce the number of memory accesses required for memory-intensive FHE operations.

5.1 Intra-FHE-operation Fusion

There are a number of opportunities for kernel fusion inside a single FHE operation. More specifically, we show that the major FHE operations in CKKS such as multiplication, rotation, and rescaling have ample opportunities to adopting kernel fusion to reduce the number of memory accesses. Below, we describe how we exploit the kernel fusion technique to those FHE operations.

Fusing the scaling kernel in ModUp with batched iNTT: We suggest and apply two optimizations to `ModUp`. First, we batch β iNTT kernels in `ModUp`s that are invoked β times per `KeySwitch`. Each execution of the fast basis conversion in `ModUp` follows $\texttt{iNTT}([a]_{\mathcal{C}'_i})$, as described in Subsection 2.3. When `dnum` is large enough, the input size of each iNTT ($LN/\texttt{dnum} = \alpha N$) decreases; one cannot maximally exploit thread-level parallelism in GPUs because the throughput of an iNTT kernel decreases due to the kernel call overhead [BVMA18, KJPA20]. Instead of launching numerous small iNTT kernels, we fuse them into a single, large iNTT kernel that takes $(\alpha\beta)N$ input residues, and launch the large kernel once.

Second, we fuse the scaling kernel of `Conv` in `ModUp` (which multiplies either $[\hat{Q}_j^{\prime\prime -1}]_{q_j}$ or $[\hat{Q}_j^{\prime\prime\prime -1}]_{p_j}$; see Subsection 2.3), with the preceding batched iNTT kernel. Because the scaling kernel has low arithmetic intensity and becomes memory-bound, fusing it with its preceding batched iNTT kernel removes most of the required memory accesses. Specifically, because we perform 8 per-thread iNTT for the batched iNTT kernel, each thread holds eight double-word output elements in its registers, before storing them into the global memory. We multiply each of the eight output elements with its corresponding $[\hat{Q}_j^{\prime\prime -1}]_{q_j}$ or $[\hat{Q}_j^{\prime\prime\prime -1}]_{p_j}$, right before storing them into global memory. This eliminates $2\alpha\beta N$ memory accesses in total.

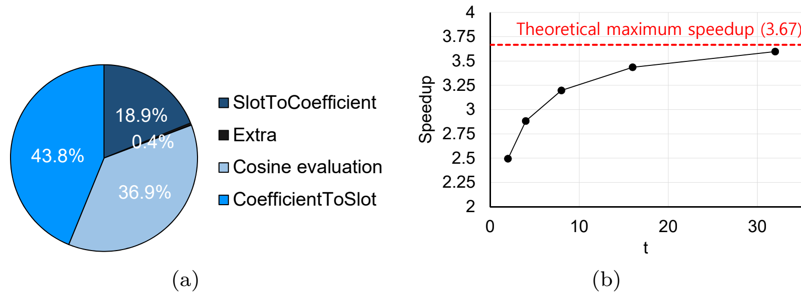


Figure 4: (a) Execution time breakdown of a single run of bootstrapping on GPU, (b) speedup of line 9-13 in Algorithm 11 with batching plaintext-ciphertext MAC for various t. The red line represents the maximum speedup, 11/3. (N, L, $\texttt{dnum}$) = (2^{17}, 29, 3).

Fusing Inner-product in key-switching: Inner-product in key-switching (see Algorithm 10) requires $(11\beta - 4)(\alpha\beta + k)N$ memory accesses in total (see Table 1). Most required global memory accesses come from reading and writing **accum** multiple times. Because each element of **accum** is 128 bits long, these operations account for most of the memory accesses.

To reduce the number of memory accesses, we fuse the three types of kernels in Algorithm 10, which are described in Subsection 3.1, into a single kernel that performs all of the multiplication, mult-and-add, and reduction operations. In the fused kernel, each thread holds the partial sum in its registers until the last mult-and-add operation is complete. This simple but effective approach reduces the number of total memory accesses required from $(11\beta - 4)(\alpha\beta + k)N$ to $(3\beta + 2)(\alpha\beta + k)N$.

Fusing element-wise operations in ModDown with NTT: We fuse two element-wise operations in `ModDown`, subtraction and scaling with $[\hat{Q}^{-1}]_{q_j}$ for each j (lines 6-8 in Algorithm 4), with a preceding NTT (line 5). Prior to the fusion step, we call one kernel for subtraction and one for scaling. Similar to the kernel fusion we apply to `ModUp`, these two operations are applied immediately before storing the NTT's output into global memory, removing $2(\ell+1)N$ global memory accesses during the subtraction and $2(\ell+1)N$ during the scaling steps overall.

This fusion strategy in `ModDown` is also applicable to the rescaling operation, as `RESCALE` can be understood as `ModDown` that drops a prime (i.e., $\texttt{ModDown}_{\mathcal{C}_\ell \to \mathcal{C}_{\ell-1}}$) on both instances of input $([a]_{\mathcal{C}_\ell}, [b]_{\mathcal{C}_\ell}) \leftarrow \mathbf{ct}$. Therefore, we also apply this fusion step when rescaling and evaluate its performance after fusion in Section 6.

5.2 Inter-FHE-operation Fusion

Most of the bootstrapping time is spent evaluating linear transformation. Therefore, we suggest optimizations applicable to linear transformation, i.e., batching plaintext-ciphertext multiply-accumulate (MAC) operations.

Breakdown on bootstrapping: Figure 4(a) shows the execution time breakdown of bootstrapping with one of the representative parameter sets in Section 6, after applying all intra-FHE-operation fusion steps in the previous section. Approximately 62.7% of the bootstrapping time is spent evaluating linear transforms, SlotToCoefficient and CoefficientToSlot, which are respectively homomorphic decoding and encoding processes. We focus on these two functions and accelerate their core algorithm, BSGS (see Equation 1).

Batching plaintext-ciphertext multiply-accumulate (MAC): As described in Subsection 2.5, the core operation of SlotToCoefficient and CoefficientToSlot is a series of multiplications between block-diagonal sparse matrices, whose diagonals are encoded into plaintexts, and a vector, which is a ciphertext, using the BSGS algorithm (Equation 1).

Our key observation is that the inner-most sum of BSGS (lines 10-13 in Algorithm 11) exhibits a severe memory bandwidth bottleneck on a GPU, similar to Inner-product (lines

5-7 in Algorithm 10). For a given ciphertext of level ℓ, the number of required memory accesses of the innermost sum of Algorithm 11 is $5t(\ell+1)N$ for t `CMULT` and $6(t-1)(\ell+1)N$ for $t-1$ `HADD`. We fuse the kernels for `CMULT` and `HADD` in lines 9-13 of Algorithm 11 into a single kernel, reducing the total memory accesses of $(11t-6)(\ell+1)N$ to $(3t-1)(\ell+1)N$. We refer to this fusion as *batching plaintext-ciphertext MAC*.

As `CMULT` and `HADD` in Algorithm 11 are memory-bound, the reduction of the number of memory accesses is translated into a speedup of as many as asymptotically 11/3 times with batching plaintext-ciphertext MAC. We benchmark the speedup of line 9-13 in Algorithm 11 with our GPU implementation for various values of t in Figure 4(b). Even with $t=2$, we obtain a 2.66× speedup. On the other hand, the overall execution time of BSGS is as follows. Let $f_{speedup}(t) = (11t-6)/(3t-1)$. Then, the execution time of Algorithm 11 becomes $(l+t-2)$ (`HROTATE` time) + $\{l \cdot t$ (`CMULT` time) + $(t-1)(l-1)$ (`HADD` time)$\}/f_{speedup}(t)$.

We can model the speedup of each execution of BSGS in CoefficientToSlot and SlotToCoefficient. For a given radix r, because each block-diagonal sparse matrix (except for the first one, which is initially multiplied with the input first) has $2r-1$ diagonals, we should select proper values of l and t in Equation 1 such that $l \cdot t > 2r-1$, with both l and t near $\sqrt{2r-1}$. Following [HHC19], we select a proper radix r first. For $N = 2^{16}$, we set r to 2^5, resulting in three matrix-vector multiplications, each with radix 2^5 (because $N/2 = (2^5)^3$). For $N = 2^{17}$, as $\log r = 5$ does not divide $\log(N/2)$ without a remainder, we perform three matrix-vector multiplications, each with radix 2^5, 2^5, and 2^6. Thus, we can determine the total time of CoefficientToSlot (the same applies to SlotToCoefficient) when $N = 2^{16}$ as follows: (total time of each baby-step giant-step in SlotToCoefficient) = 14 (`HROTATE` time) + $\{64$ (`CMULT` time) + 49 (`HADD` time)$\}$ $/f_{speedup}(8)$, where $f_{speedup}(8) = 3.56$.

6 Evaluation

We present the performance improvement in FHE operations in CKKS with our inter- and intra-FHE-optimizations, the effect of generalized key-switching, and hoisting [2]. All experiments were performed on a single NVIDIA Tesla V100 [NVI17] GPU with CUDA 11.2 and an Intel Xeon Gold 6234 @3.3GHz CPU with eight cores [Int20]. We also put all the evaluation keys and constants into the global memory for both the CPU and GPU implementations. We used all parameters that meet the stipulation of security bit $\lambda > 80$, as calculated from the LWE estimator [APS15]. Our parameter sets used for evaluating the performance of bootstrapping add 2^{-19} mean errors to the input message after bootstrapping; these values are small enough so as not to hinder most applications [HK20, CCS19, HHC19, CHK+18].

Performance of individual FHE operations: Table 2 summarizes the latency of the FHE operations on a single-threaded CPU, our GPU implementation, and PrivFT [BHM+20], the state-of-the-art implementation of CKKS [CHK+19] on a GPU. Because PrivFT is closed-source, we compared the latency of each operation provided by the corresponding paper using the same GPU. Also, we compared the performance on the parameter set with the largest level in [BHM+20] considering the practical use of FHE. This parameter set with a large level represents parameter sets suitable for applications that do not require bootstrapping.

First, by applying all of the intra-FHE-operation fusion techniques, our optimized GPU implementation beats the baseline CPU implementation by 152×, 153×, 229×, and 135× in `HMULT`, `HROTATE`, `RESCALE`, and `HADD`, respectively. Second, compared to PrivFT [BHM+20], even without the optimizations, our baseline implementation outperforms PrivFT by 1.67×

[2] The GPU implementations of the inter- and intra-FHE-optimizations are available in https://github.com/scale-snu/ckks-gpu-core.

Table 2: Execution time of FHE operations in our single-threaded CPU implementation, GPU implementation without (baseline) and with (**fused**, **fused**$_L$, and **fused**$_H$) our optimizations, and PrivFT [BHM+20].

	Execution time (ms)						Speedup
	CPU	GPU					
	1 thread	base	**fused**	**fused**$_L$	**fused**$_H$	PrivFT [BHM+20]	
N	2^{16}	2^{16}	2^{16}	2^{16}	2^{17}	2^{16}	
$\log PQ$	2366	2366	**2366**	**2364**	3220	2360*	
$\log Q$	2305	2305	2305	1693	2305	2300	
L	44	44	**44**	32	**44**	44	
`dnum`	45	45	45	3	3	45	
λ	98	98	98	100	128	98	
`HMULT`	2644.8	33.51	17.4	2.96	**7.96**	55.884	**7.02**×
`HROTATE`	2578.9	32.93	16.83	2.55	6.6	-	-
`RESCALE`	145.4	0.846	0.635	0.49	1.20	1.632	1.36×
`HADD`	28.12	0.208	0.208	0.162	0.378	0.188	0.50×
`CMULT`	26.22	0.177	0.177	0.135	0.318	0.170	0.54×

* We estimate $\log P$ as 60 since the paper does not provide the value.

Table 3: Bootstrapping time of our baseline implementation and ones that incrementally adopt MF, IF, and MDF together (Intra-FHE-fusion), batching plaintext-ciphertext MAC (Batching), and hoisting (Hoisting).

Parameter set	Execution time (ms)					Speedup (vs. CPU 1 thread)
(N, Level, `dnum`) (logPQ, λ)	CPU 1 thread	GPU Baseline	Intra-FHE fusion	Batching	Hoisting	
(2^{16}, 34, 5) (2222, 106)	79444	428.94	377.78	351.09	328.25	242×
(2^{17}, 29, 3) (2150, 173)	135400	719.87	623.92	568.2	526.96	257×

in `HMULT`. Our memory-centric optimizations are effective such that we realize a speedup of 3.21× in `HMULT` (**fused**), compared to that of PrivFT.

We also demonstrated the impact of choosing a proper value of `dnum` in the GPU implementation. Because using a smaller `dnum` decreases the multiplicative level with a fixed $\log PQ$ and security bit, we show two parameter sets having either the same $\log PQ$ with fewer levels (**fused**$_L$), or the same level with higher $\log PQ$ and N values to ensure high security of 128 bits (**fused**$_H$). Comparing the result of **fused**$_H$ with PrivFT, which does not exploit generalized key-switching, we improved the performance by 7.02× in `HMULT`. In the **fused**$_H$ case, `HADD`, `RESCALE`, and `CMULT` become slower because the ciphertext size is larger than in **fused**; however, the execution time of `HMULT` and `HROTATE` generally dominates by one order of magnitude. Compared to the method that uses maximum `dnum` (**fused**), **fused**$_L$ obtains speedups of 5.88× and 6.6× in `HMULT` and `HROTATE`, respectively. Our results thus demonstrate the importance of memory-centric optimizations and the proper selection of `dnum` when accelerating CKKS.

Performance of bootstrapping: We evaluate the performance of bootstrapping with our intra- and inter-FHE-operation fusion strategy with hoisting [HS18] in Table 3. For this evaluation, we select two parameter sets with small `dnum` and moderate levels, which represent bootstrap-friendly parameter sets suited for applications demanding bootstrapping. First, our baseline GPU implementation already provides up to a 185× speedup compared to implementation on a CPU. Using a small `dnum` value increased the performance gap

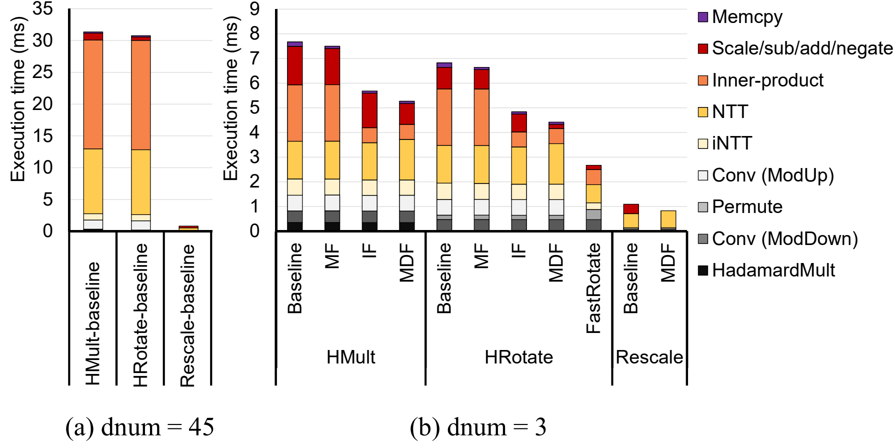


Figure 5: Performance of each single FHE operation without fusion (Baseline) and incrementally adopting fusion. FastRotate represents the implementation that applies hoisting after MDF. (N, $logQ$, L) = (2^{16}, 2305, 44).

between the CPU and GPU implementations by relieving the memory bandwidth bottleneck on the GPU. Second, intra-FHE-operation fusions lead to a speedup of up to 1.15× compared to the Baseline case. Moreover, batching plaintext-ciphertext MAC and hoisting provide an additional speedup of up to 257× in total compared to the single-thread CPU implementation. In both parameter sets, we found similar speedups on each optimization.

Speedup breakdown: Using a small `dnum` significantly reduces the Inner-product time by a factor of 7.48 and 7.50 in our baseline GPU implementation of `HMULT` and `HROTATE`, respectively (see Figure 5(a) and (b)). As the number of memory accesses required by Inner-product is reduced to become proportional to `dnum`, the portion of the Inner-product time changes from 54.6% (55.8%) when `dnum`= 45 to 29.8% (33.5%) when `dnum`= 3 in `HMULT` (`HROTATE`).

After reducing memory accesses using a small `dnum` value, we analyze the effect of each intra-FHE-operation fusion (Figure 5(b)). We refer to the three intra-FHE-operation fusions listed in Subsection 5.1 as ModUp Fusion (*MF*), Inner-product Fusion (*IF*), and ModDown Fusion (*MDF*). Figure 5(b) shows the execution time breakdown when we applied MF, IF, and MDF incrementally. MF increases the overall performance of `HMULT` and `HROTATE` by 1.02× and 1.03×, respectively. The speedup is relatively small, as the size of `iNTT` in `ModUp` is small, taking up a small portion in the overall execution time. On top of MF, IF significantly decreases the execution time of Inner-product by reducing the number of memory accesses by up to a factor of 11/3, resulting in a 3.75× (3.75×) speedup in Inner-product and a 1.35× (1.41×) speedup in total for `HMULT` (`HROTATE`). Lastly, MDF increases the overall performance by 1.46×, 1.54×, and 1.32× correspondingly for `HMULT`, `HROTATE`, and `RESCALE` compared to the baseline.

We also evaluate the effectiveness of hoisting [HS18] in `HROTATE`, shown as a distinct column in Figure 5(b). With hoisting, `FastRotate` outperforms `HROTATE` with MDF by 1.65×, as it does not require ModUp or multiple NTT/iNTT executions. The execution time for permutation (Frobenius map in the NTT domain) increases by 2.32× as the size of the permutation becomes larger by precomputing ModUp; however, the performance benefit outweighs the cost as permutation only accounted for 4% in `HROTATE` with MDF.

Performance of Baby-step Giant-step: Figure 6(a) shows the performance improvement in BSGS with batching plaintext-ciphertext MAC and hoisting with various values of t. Batching plaintext-ciphertext MAC improves the performance up to 1.25×. Moreover, applying hoisting results in a speedup of up to 1.45×, while providing no benefit when $t = 2$ as we perform only one `FastRotate` and thus cannot save any `ModUp`.

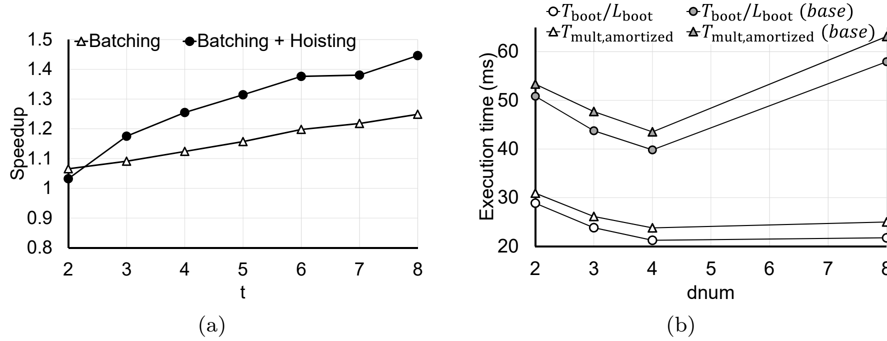


Figure 6: (a) Speedup of evaluation of BSGS (Algorithm 11) with batching plaintext-ciphertext MAC (Batching) and hoisting over t. (b) Amortized FHE-mult time ($T_{\text{mult,Amortized}}$) for a given $\log PQ \approx 2450$ and N $= 2^{16}$ (97-bit security).

Amortized FHE-mult time: We propose a new metric for performance that reflects the performance of actual applications requiring bootstrapping: *amortized FHE-mult time*. It considers both the bootstrapping cost and the number of multiplications between two consecutive bootstrapping operations. Let one perform consecutive HMULT until the level of a ciphertext is depleted, perform bootstrapping, and then repeat this process. We define the amortized FHE-mult time as $T_{\text{mult,amortized}} = T_{\text{mult}} + \frac{T_{\text{boot}}}{L_{\text{boot}}}$, where T_{mult} is the average time of HMULT between two bootstrapping operations, T_{boot} is the bootstrapping time, and L_{boot} is the number of HMULT operations between bootstrappings.

Figure 6(b) shows the amortized FHE-mult time of our best-performing and baseline GPU implementation with various values of dnum for a given security bit. In both implementations, the amortized time becomes minimal when dnum $= 4$ (24.35 ms and 43.5 ms for the optimized and baseline cases, respectively). However, our optimized implementation greatly reduces the performance gap between the optimal dnum and the large value (dnum $= 8$), as it significantly amortizes the memory bandwidth pressure by applying memory-centric optimizations. This makes high dnum values more attractive for those who want more security with the parameter sets that consider bootstrapping.

Our optimized implementation of CKKS outperforms the baseline by 1.68×, 1.78×, 1.79×, and 2.46× with dnum values of 2, 3, 4, and 8, respectively. By adopting generalized key-switching as well as memory-centric optimizations, *the amortized FHE-mult time of our optimized GPU implementation even outperforms the single multiplication time of PrivFT [BHM+20], which does not support bootstrapping.*

Performance of Logistic Regression: *Our optimized GPU implementation of CKKS achieves a similar level of performance improvement in complex applications requiring bootstrapping.* As a target application, we evaluated the execution time required to train a logistic regression model. We used a methodology from [HHCP19], which trains a binary classification model with non-RNS CKKS [CKKS17]. Moreover, we used the same dataset (the subset of MNIST labeled as 3 and 8), learning rate, weight update algorithm, and the circuit that approximates a Sigmoid function as a cubic polynomial.

In [HHCP19], the authors exploited the gradient descent algorithm to find the weight vector $\mathbf{w}$ that minimizes the negative log-likelihood function $NL(\mathbf{w}) = -1/m \cdot \log P(\mathbf{w})$, where m is the batch size, $P(\mathbf{w}) = \Pi_{i=1}^{m} p_{\mathbf{w}}(\mathbf{x}_i)^{y_i} \cdot (1 - p_{\mathbf{w}}(\mathbf{x}_i))^{1-y_i}$ and $p_{\mathbf{w}}(\mathbf{x}_i) = \mathcal{S}(\mathbf{IP}(\mathbf{w}, (1, \mathbf{x}_i)))$. Here $(1, \mathbf{x}_i)$ is a concatenation of 1 and vector $\mathbf{x}_i$, $\mathbf{IP}(\mathbf{w}, (1, \mathbf{x}_i))$ is the inner product between the two vectors, and $\mathcal{S}(x)$ is a Sigmoid function, which is approximated as a cubic polynomial [HHCP18].

Given learning rate α and gradient $\Delta_w NL(\mathbf{w})$, training is an iterative process that

Table 4: Performance of training a binary classification model with 8-threaded CPU implementation (CPU-8T) and our optimized GPU implementation with parameter set $(N, \Delta, \log PQ, L, \texttt{dnum}) = (2^{17}, 2^{51}, 2395, 35, 3)$.

	Accuracy (%)	Execution time per mini-batch (ms)		
		Bootstrapping	Others	Total
CPU-8T	96.4	15 284.7	15 760.3	31 045.0
GPU	96.4	328.7	446.3	775.0
Speedup		46.5×	35.3×	40.0×

updates the weight $\mathbf{w}$ to $\mathbf{w}'$ such that

$$\mathbf{w}' = \mathbf{w} - \alpha \cdot \Delta_w NL(\mathbf{w}),\ \Delta_w NL(\mathbf{w}) = -1/m \cdot \sum_{i=1}^{m} \mathcal{S}(-\mathbf{IP}(\mathbf{z}_i, \mathbf{w})) \cdot \mathbf{z}_i$$

where $\mathbf{z}_i = y_i' \cdot (1, \mathbf{x}_i)$, and $y_i' = 2y_i - 1 \in \{-1, 1\}$. We packed the training data into ciphertexts such that each sample of training data is packed to a ciphertext, with each ciphertext holding $(N/2)/f'$ training samples, where f' is the smallest power of two, which is greater than the number of features f. As in [HHCP19], we compress each patch of 2×2 pixels in the MNIST dataset [Lec98] into their arithmetic means, resulting in $f = (28/2) \times (28/2) = 196$.

We apply our implementation of CKKS to a circuit [HHCP19] that exploits non-RNS CKKS, resulting in five multiplicative depths per training iteration. We use a parameter set of $N = 2^{17}$, $\Delta = 2^{51}$, $\log PQ = 2395$, $L = 35$, and $\texttt{dnum}= 3$. Because the level in our parameter set is 35, each bootstrapping operation consumes 16 levels and leaves 19 levels, requiring a bootstrapping for every 3 iterations.

Table 4 shows the execution time of our 8-threaded CPU and the best-performing GPU implementation for a single iteration of training the binary classification model. We showed the amortized time per iteration (i.e., the total learning time divided by the number of iterations). We achieve 96.4% accuracy and 0.99 AUROC (area under the receiver operating characteristics) with 30 iterations, identical to the outcome in [HHCP19]. Our GPU implementation outperforms the CPU implementation by 40.0× overall, resulting in a sub-second iteration per mini-batch, whereas the earlier work [HHCP19] reported four minutes of execution time per iteration with an 8-threaded CPU with non-RNS CKKS implementation [Cry20].

7 Related work

To the best of our knowledge, PrivFT [BHM+20] is the only method that accelerates the RNS-variant of CKKS using a GPU, exploiting the libraries from their previous works [BPA+19, BVMA18] that implemented the RNS-variants [BEHZ16a, HPS19a] of the Brakerski-Fan-Vercauteren (BFV) scheme [Bra12, FV12]. A prior work [JLK+21] accelerates CKKS on GPUs, but they did not target the RNS-variant. All of them did not implement bootstrapping.

An open-source library called cuFHE [Ver18] implemented a TFHE scheme [CGGI16, CGGI17], including the bootstrapping of TFHE, which is the fastest bootstrapping method, though it accommodates at most up to several bit plaintext per ciphertext. We conducted benchmarking on the GPU we used throughout the evaluation and obtained a bootstrapping time of 0.5 ms per binary gate in cuFHE. Our bootstrapping implementation, as shown in Table 3, exhibits a per-slot bootstrapping time of 8 us, resulting in a 62.5× speedup.
NTT vs. DGT: Recent works [BVMA18, ANA21] suggest adopting Discrete Gaussian Transform (DGT), which exploits Gaussian primes and transforms the N-size datapath in NTT into $N/2$-size datapath. DGT saves $N/2$ twiddle factors, which can amortize

the bandwidth requirement in GPUs. However, it requires an extra *twisting* step (an element-wise multiplication) prior to transformation, trading the savings in twiddle factors with $N/2$ additional twisting factors.

Applicability to other schemes: Our memory-centric optimizations are partially applicable to the BFV [BPA+19] and TFHE [CGGI20] schemes, which include operations with low arithmetic intensities. Both schemes include dot product between a polynomial and a key-switching key during key-switching, whose arithmetic intensity is low enough to benefit from the kernel fusion in Inner-product. For BFV, the fusion in `ModUp` and `ModDown` is available when the same RNS decomposition methods are used [HPS19b, BEHZ16b] as in [HK20]. TFHE has an operation called *Cmux* [CGGI20], which consists of Hadamard multiplications, permutations, and additions between ciphertexts. Because Cmux is consecutively and repeatedly used (e.g., in *BlindRotate* [CGGI20]), we believe plaintext-ciphertext MAD batching will be effective. A comprehensive performance characterization of HE operations in BFV and TFHE, which is required for understanding the bandwidth requirements of the corresponding GPU implementations, will be our future work.

8 Conclusion

In this paper, we demonstrated the first GPU implementation of the bootstrapping of CKKS. We showed that the global memory bandwidth bottleneck is the key challenge in accelerating RLWE-based FHE operations with GPUs. Also, we found that the decomposition number (`dnum`) in a parameter set significantly impacts the performance; raising `dnum` significantly increases both the number of global memory accesses and the capacity required for Inner-product during key-switching. Based on the observation of the high memory bandwidth requirement, we devised memory-centric optimizations for our GPU implementation, in this case kernel fusion and a proper choice of `dnum`. Also, by applying our batching plaintext-ciphertext MAC in bootstrapping, we realized a per-slot bootstrapping time of 8 us with 19-bit precision, which corresponds to a 257× speedup over the single-threaded CPU implementation. Finally, we demonstrated the effectiveness of our solutions by training a logistic regression model, which obtains a speedup of 40.0× with our single GPU implementation compared to the 8-threaded CPU implementation.

Acknowledgement

This work was supported in part by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2020-0-00840, 40%) and National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (No. 2020R1A2C2010601, 30%), (No. 2019R1A2C4069769, 30%). Younho Lee is the corresponding author of this paper.

References

[AB75] Ramesh C. Agarwal and C. Sidney Burrus. Number Theoretic Transforms to Implement Fast Digital Convolution. *Proceedings of the IEEE*, 63(4):550–560, April 1975.

[AFK+19] Mohamed Arafa, Bahaa Fahim, Sailesh Kottapalli, Akhilesh Kumar, Lily Pao Looi, Sreenivas Mandava, Andy Rudoff, Ian M. Steiner, Bob Valentine, Geetha Vedaraman, and Sujal Vora. Cascade lake: Next generation intel xeon scalable processor. *IEEE Micro*, 39(2):29–36, 2019.

[ANA21] Pedro G. M. R. Alves, Jheyne Nayara Ortiz, and Diego F. Aranha. Faster homomorphic encryption over gpgpus via hierarchical dgt. In *International Conference on Financial Cryptography and Data Security*, 2021.

[APS15] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *J. Math. Cryptol.*, 9(3):169–203, 2015.

[Bar87] Paul Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *CRYPTO'86*, volume 263 of *LNCS*, pages 311–323. Springer, Heidelberg, August 1987.

[BEHZ16a] Jean-Claude Bajard, Julien Eynard, M. Anwar Hasan, and Vincent Zucca. A full RNS variant of FV like somewhat homomorphic encryption schemes. In *SAC 2016*, volume 10532 of *LNCS*, pages 423–442. Springer, Heidelberg, August 2016.

[BEHZ16b] Jean-Claude Bajard, Julien Eynard, M Anwar Hasan, and Vincent Zucca. A full rns variant of fv like somewhat homomorphic encryption schemes. In *International Conference on Selected Areas in Cryptography*, pages 423–442. Springer, 2016.

[BHM+20] Ahmad Al Badawi, Louie Hoang, Chan Fook Mun, Kim Laine, and Khin Mi Mi Aung. Privft: Private and fast text classification with homomorphic encryption. *IEEE Access*, 8:226544–226556, 2020.

[BHS19] Song Bian, Masayuki Hiromoto, and Takashi Sato. Hardware-accelerated secured naïve bayesian filter based on partially homomorphic encryption. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, 102-A(2):430–439, 2019.

[BPA+19] Ahmad Al Badawi, Yuriy Polyakov, Khin Mi Mi Aung, Bharadwaj Veeravalli, and Kurt Rohloff. Implementation and performance evaluation of RNS variants of the BFV homomorphic encryption scheme. *IEEE Transactions on Emerging Topics in Computing*, 2019.

[Bra12] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In *CRYPTO 2012*, volume 7417 of *LNCS*, pages 868–886. Springer, Heidelberg, August 2012.

[BVMA18] Ahmad Al Badawi, Bharadwaj Veeravalli, Chan Fook Mun, and Khin Mi Mi Aung. High-performance FV somewhat homomorphic encryption on GPUs: An implementation using CUDA. *IACR TCHES*, 2018(2):70–95, 2018. https://tches.iacr.org/index.php/TCHES/article/view/875.

[CCF+67] William T. Cochran, James W. Cooley, David L. Favin, Howard D. Helms, Reginald A. Kaenel, William W. Lang, George C. Maling, David E. Nelson, Charles M. Rader, and Peter D. Welch. What is the Fast Fourier Transform? *Proceedings of the IEEE*, 55(10):1664–1674, Oct 1967.

[CCS19] Hao Chen, Ilaria Chillotti, and Yongsoo Song. Improved bootstrapping for approximate homomorphic encryption. In *EUROCRYPT 2019, Part II*, volume 11477 of *LNCS*, pages 34–54. Springer, Heidelberg, May 2019.

[CGGI16] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In *ASIACRYPT 2016, Part I*, volume 10031 of *LNCS*, pages 3–33. Springer, Heidelberg, December 2016.

[CGGI17] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster packed homomorphic operations and efficient circuit bootstrapping for TFHE. In *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, pages 377–408. Springer, Heidelberg, December 2017.

[CGGI20] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, January 2020.

[CH18] Hao Chen and Kyoohyung Han. Homomorphic lower digits removal and improved FHE bootstrapping. In *EUROCRYPT 2018, Part I*, volume 10820 of *LNCS*, pages 315–337. Springer, Heidelberg, April / May 2018.

[CHK+18] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. Bootstrapping for approximate homomorphic encryption. In *EUROCRYPT 2018, Part I*, volume 10820 of *LNCS*, pages 360–384. Springer, Heidelberg, April / May 2018.

[CHK+19] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. A full RNS variant of approximate homomorphic encryption. In *SAC 2018*, volume 11349 of *LNCS*, pages 347–368. Springer, Heidelberg, August 2019.

[CKKS17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. Homomorphic encryption for arithmetic of approximate numbers. In *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, pages 409–437. Springer, Heidelberg, December 2017.

[CLP17] Hao Chen, Kim Laine, and Rachel Player. Simple encrypted arithmetic library - SEAL v2.1. In *FC 2017 Workshops*, volume 10323 of *LNCS*, pages 3–18. Springer, Heidelberg, April 2017.

[Cra19] Jack Lik Hon Crawford. *Fully Homomorphic Encryption Applications: The Strive Towards Practicality.* PhD thesis, Queen Mary University of London, 2019.

[Cry20] CryptoLab Inc. HEAAN, 2020. https://github.com/snucrypto/HEAAN.

[CT65] James W. Cooley and John W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, 19(90):297–301, 1965.

[FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012. https://eprint.iacr.org/2012/144.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *41st ACM STOC*, pages 169–178. ACM Press, May / June 2009.

[GHS12] Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic evaluation of the AES circuit. In *CRYPTO 2012*, volume 7417 of *LNCS*, pages 850–867. Springer, Heidelberg, August 2012.

[GLD+08] Naga K. Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith, and John Manferdelli. High performance discrete fourier transforms on graphics processors. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–12, 2008.

[Har14] David Harvey. Faster arithmetic for number-theoretic transforms. *J. Symb. Comput.*, 60:113–119, 2014.

[HHC19] Kyoohyung Han, Minki Hhan, and Jung Hee Cheon. Improved homomorphic discrete fourier transforms and FHE bootstrapping. *IEEE Access*, 7:57361–57370, 2019.

[HHCP18] Kyoohyung Han, Seungwan Hong, Jung Hee Cheon, and Daejun Park. Efficient logistic regression on large encrypted data. Cryptology ePrint Archive, Report 2018/662, 2018. https://eprint.iacr.org/2018/662.

[HHCP19] Kyoohyung Han, Seungwan Hong, Jung Hee Cheon, and Daejun Park. Logistic regression on homomorphic encrypted data at scale. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01):9466–9471, Jul. 2019.

[HK20] Kyoohyung Han and Dohyeong Ki. Better bootstrapping for approximate homomorphic encryption. In *CT-RSA 2020*, volume 12006 of *LNCS*, pages 364–390. Springer, Heidelberg, February 2020.

[HLF+19] Kai Huang, Ximeng Liu, Shaojing Fu, Deke Guo, and Ming Xu. A lightweight privacy-preserving CNN feature extraction framework for mobile sensing. *IEEE Transactions on Dependable and Secure Computing*, 2019.

[HP17] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 6th edition, 2017.

[HPS19a] Shai Halevi, Yuriy Polyakov, and Victor Shoup. An improved RNS variant of the BFV homomorphic encryption scheme. In *CT-RSA 2019*, volume 11405 of *LNCS*, pages 83–105. Springer, Heidelberg, March 2019.

[HPS19b] Shai Halevi, Yuriy Polyakov, and Victor Shoup. An improved rns variant of the bfv homomorphic encryption scheme. In *Cryptographers' Track at the RSA Conference*, pages 83–105. Springer, 2019.

[HS14] Shai Halevi and Victor Shoup. Algorithms in HElib. In *CRYPTO 2014, Part I*, volume 8616 of *LNCS*, pages 554–571. Springer, Heidelberg, August 2014.

[HS15] Shai Halevi and Victor Shoup. Bootstrapping for HElib. In *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 641–670. Springer, Heidelberg, April 2015.

[HS18] Shai Halevi and Victor Shoup. Faster homomorphic linear transformations in HElib. In *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 93–120. Springer, Heidelberg, August 2018.

[Int20] Intel Corporation. Second Generation Intel Xeon Scalable Processors. Technical Report 336198-002, 2020. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/2nd-gen-xeon-scalable-processors-brief-Feb-2020-2.pdf.

[JHK+20] Xiaoqian Jiang, Arif O. Harmanci, Miran Kim, Haixu Tang, XiaoFeng Wang, Tsung-Ting Kuo, and Lucila Ohno-Machado. IDASH PRIVACY & SECURITY WORKSHOP 2020 - secure genome analysis competition, 2020. http://www.humangenomeprivacy.org/2020/.

[JLK+21] Wonkyung Jung, Eojin Lee, Sangpyo Kim, Jongmin Kim, Namhoon Kim, Keewoo Lee, Chohong Min, Jung Hee Cheon, and Jung Ho Ahn. Accelerating fully homomorphic encryption through architecture-centric analysis and optimization. *IEEE Access*, 2021.

[KJPA20] Sangpyo Kim, Wonkyung Jung, Jaiyoung Park, and Jung Ho Ahn. Accelerating number theoretic transformations for bootstrappable homomorphic encryption on gpus. In *IEEE International Symposium on Workload Characterization, IISWC 2020, Beijing, China, October 27-30, 2020*, pages 264–275. IEEE, 2020.

[KJT+20] Tsung-Ting Kuo, Xiaoqian Jiang, Haixu Tang, XiaoFeng Wang, Tyler Bath, Diyue Bu, Lei Wang, Arif Harmanci, Shaojie Zhang, Degui Zhi, Heidi J. Sofia, and Lucila Ohno-Machado. iDASH Secure Genome Analysis Competition 2018: Blockchain Genomic Data Access Logging, Homomorphic Encryption on GWAS, and DNA Segment Searching. *BMC Medical Genomics*, 13(7), 2020.

[KO62] Anatolii Alekseevich Karatsuba and Yu P Ofman. Multiplication of many-digital numbers by automatic computers. In *Doklady Akademii Nauk*, volume 145, pages 293–294. Russian Academy of Sciences, 1962.

[Lec98] Yann Lecun. The MNIST Database of Handwritten Digits, 1998. http://yann.lecun.com/exdb/mnist.

[NVI17] NVIDIA Corporation. NVIDIA Tesla V100 GPU Architecture. Technical Report WP-08608-001_v1.1, 2017. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[NVI20] NVIDIA Corporation. Nsight Compute, 2020. https://docs.nvidia.com/nsight-compute/NsightCompute/index.html (visited on Jan 20th, 2021).

[NVI21] NVIDIA Corporation. NVIDIA Ampere GA102 GPU Architecture. Technical report, 2021. https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf.

[QRHT19] Bo Qiao, Oliver Reiche, Frank Hannig, and Jürgen Teich. From loop fusion to kernel fusion: A domain-specific approach to locality optimization. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2019, Washington, DC, USA, February 16-20, 2019*, pages 242–253. IEEE, 2019.

[Sho16] Victor Shoup. NTL: A library for doing number theory, 2016. https://libntl.org.

[TLW19] Jing Tian, Jun Lin, and Zhongfeng Wang. Ultra-fast modular multiplication implementation for isogeny-based post-quantum cryptography. In *2019 IEEE International Workshop on Signal Processing Systems, SiPS 2019, Nanjing, China, October 20-23, 2019*, pages 97–102. IEEE, 2019.

[Ver18] Vernam Group. cuFHE, 2018. https://github.com/vernamlab/cuFHE (visited on Jan 20th, 2021).

[WTW+18] XiaoFeng Wang, Haixu Tang, Shuang Wang, Xiaoqian Jiang, Wenhao Wang, Diyue Bu, Lei Wang, Yicheng Jiang, and Chenghong Wang. iDASH secure genome analysis competition 2017. *BMC Medical Genomics*, 11(4), 2018.

[YZLT19] Fan Yin, Yandong Zheng, Rongxing Lu, and Xiaohu Tang. Achieving efficient and privacy-preserving multi-keyword conjunctive query over cloud. *IEEE Access*, 7:165862–165872, 2019.

[ZLX+20] Chengliang Zhang, Suyi Li, Junzhe Xia, Wei Wang, Feng Yan, and Yang Liu. Batchcrypt: Efficient homomorphic encryption for cross-silo federated learning. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pages 493–506. USENIX Association, 2020.

A An Overview of GPU Architecture

We briefly explain the architecture of modern GPUs. Typical GPUs exploit massive thread-level parallelism (TLP) by exploiting many scalar, in-order processors that run concurrently. A GPU consists of dozens of hardware units called Streaming Multiprocessors (*SM*), each of which can run thousands of threads concurrently. Threads in a GPU are grouped into a unit called *thread block*. Threads in a thread block share the resource assigned to the thread block, such as register file and a user-configurable scratchpad memory, called *shared memory*, typically sized several dozens of KBs [NVI17, NVI21].

Threads in a thread block are again grouped into a unit of 32 threads, called *warp*. All the threads in a warp execute an instruction at a time in a lockstep manner. An SM can hold multiple thread blocks at a time. Each SM holds multiple *warp schedulers*, each selecting one or more warps ready for issuing an instruction at the cycle. The latency of an instruction executed by a warp is hidden by other warps selected by the warp schedulers. Thus, GPU is well suited to well-parallelizable programs and can run a massive number of threads in a throughput-oriented manner. A group of thread blocks is grouped into a unit called *grid*. The number of thread blocks in a grid and the number of threads in a thread block is specified by the execution unit of GPU called *kernel*, a user-specified function called by CPU.

Our target GPU architecture is NVIDIA Tesla V100 GPU [NVI17], which we use for all the experiments in this paper. It features 80 SMs that can run up to 163,840 threads concurrently. Compared to a modern server-class CPU that typically has a last-level cache sized dozens of MBs, a GPU has a smaller cache but provides higher main memory (global memory) bandwidth. We evaluated the performance of our CPU implementation of CKKS with Intel Xeon Gold 6234 [Int20] that has 24.75 MBs of last-level cache per socket, which is large enough to accommodate one or two polynomials with typical HE parameters. In contrast, Tesla V100 has a last-level cache of 6 MBs but is equipped with a large global memory bandwidth of 900 GB/s, being suitable for throughput-oriented programs.

B Matrix Decomposition in CoefficientToSlot/SlotToCoefficient

The matrix decomposition of $\mathbf{V}$ in Subsection 2.5 with Cooley-Tukey algorithm [CT65] in the previous work [HHC19] is as follows. For the bit-reversing (permutation) matrix $\mathbf{R}$, let $\mathbf{V}_{rev} = \mathbf{VR}$. With a radix of 2, the following equations hold:

$$\mathbf{V}_{rev} = \prod_{i=1}^{log_2(N/2)} \mathbf{V}_i^{(2)},$$

$$\mathbf{V}_i^{(2)} = \begin{bmatrix} \begin{bmatrix} \mathbf{I}_{\frac{N}{2^i}} & \mathbf{W}_{\frac{N}{2^i}} \\ \mathbf{I}_{\frac{N}{2^i}} & -\mathbf{W}_{\frac{N}{2^i}} \end{bmatrix} & 0 & \cdots & 0 \\ 0 & \begin{bmatrix} \mathbf{I}_{\frac{N}{2^i}} & \mathbf{W}_{\frac{N}{2^i}} \\ \mathbf{I}_{\frac{N}{2^i}} & -\mathbf{W}_{\frac{N}{2^i}} \end{bmatrix} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \begin{bmatrix} \mathbf{I}_{\frac{N}{2^i}} & \mathbf{W}_{\frac{N}{2^i}} \\ \mathbf{I}_{\frac{N}{2^i}} & -\mathbf{W}_{\frac{N}{2^i}} \end{bmatrix} \end{bmatrix}$$

where $\mathbf{I}_n$ is an $n \times n$ identity matrix and $\mathbf{W}_n$ is an $n \times n$ diagonal matrix whose (i,i)-th element is $\omega_{4n}^{5^i}$ for $0 \leq i < n$. With the decomposition, each $\mathbf{V}_i^{(2)}$ has two $(i = 1)$ or three $(i \neq 1)$ diagonals. A decomposition with a higher radix with $r > 2$ is obtained by

multiplying the matrices together:

$$\prod_{i=1}^{log_2(N/2)} \mathbf{V}_i^{(2)} = \prod_{i=1}^{log_r(N/2)} (\prod_{j=1}^{log_2(r)} \mathbf{V}_{ri+j}^{(2)}) = \prod_{i=1}^{log_r(N/2)} \mathbf{V}_i^{(r)}$$

where each $\mathbf{V}_i^{(r)}$ has r (when $i = 1$) or $2r - 1$ ($i \neq 1$) diagonals. For the complete proof, please refer to [HHC19].

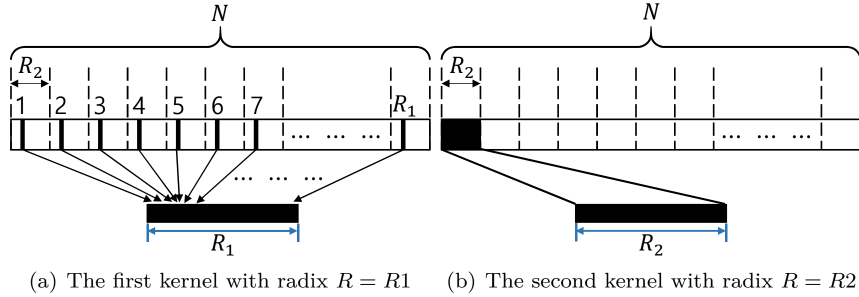
Then, SlotToCoeff and CoeffToSlot directly use the bit-reversed matrix $\mathbf{V}_{rev}$ for their evaluations. CoeffToSlot is modified to output bit-reversed results $\vec{t_1}$ and $\vec{t_2}$:

$$\vec{t_1} = \mathbf{R}\vec{z_1} = \frac{1}{N}(\overline{\mathbf{V}_{rev}}^T \vec{z} + \mathbf{V}_{rev}\overline{\vec{z}})$$
$$\vec{t_2} = \mathbf{R}\vec{z_2} = \frac{1}{N}(-i\overline{\mathbf{V}_{rev}}^T \vec{z} + i\mathbf{V}_{rev}\overline{\vec{z}}).$$

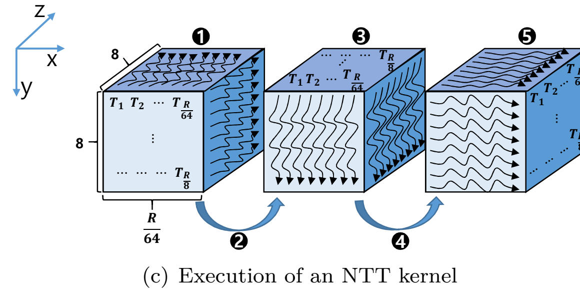
In the same way, SlotToCoeff takes bit-reversed inputs but outputs a non-bit-reversed result because the inputs are reversed again: $\vec{z} = \mathbf{V}_{rev}(\vec{t_1} + i\vec{t_2})$.

C GPU implementation of NTT and fast basis conversion

We describe 8 per-thread NTT implementation [KJPA20] that we used throughout this paper. Because iNTT is symmetric with NTT, we only show the case of NTT. Figure 7(a) and (b) show the data access pattern of the two kernels composing an NTT execution. In the two kernels, a thread block loads $R1$ or $R2$ residues that are selected from N residues corresponding to a prime. The residues loaded to the register file can be considered as a 3-D data cube (Figure 7(c)), where an index of the residues, idx, maps to $(x, y, z) = (idx\%\frac{R}{64}, (idx\%\frac{R}{8})/\frac{R}{64}, idx/\frac{R}{8})$ so that the x dimension comes first.



(a) The first kernel with radix $R = R1$ (b) The second kernel with radix $R = R2$



(c) Execution of an NTT kernel

Figure 7: Data access pattern of a thread block in (a) the first kernel and (b) the second kernel of N-size 8 per-thread NTT implementation with GPU [KJPA20], where $N = R_1 \cdot R_2$. (c) The thread block, which consists of $R/8$ threads, performs radix-R NTT on the loaded data cube, where a thread performs an NTT with a size of 8 three times (❶, ❸ , ❺). Also, synchronization of a thread block for storing the output to shared memory is shown (❷, ❹).

Algorithm 12: Implementation of fast basis conversion in GPU

```
1  $\mathcal{S} := \{q_i\}_{i\in[0,l)}$
2  $\mathcal{S}' := \{q'_i\}_{i\in[0,l')}$ $s.t.$ $\mathcal{S}\cap\mathcal{S}' = \emptyset$
3  $\hat{q}_i = (\prod_{j=0}^{l-1} q_j)/q_i$,
4  in := $[a(X)]_{\mathcal{S}}$ $s.t.$ $a(X) = \sum_{i=0}^{N-1} a_i X^i$
5  out := $[a'(X)]_{\mathcal{S}'} = \texttt{Conv}_{\mathcal{S}\to\mathcal{S}'}([a(X)]_{\mathcal{S}})$ $s.t.$ $a'(X) = \sum_{i=0}^{N-1} a'_i X^i$
6  procedure FastBasisConversionStep1                    // launch $lN$ threads
7      idx = blockDim.x * blockIdx.x + threadIdx.x
8      $i$ = idx / $N$
9      $j$ = idx % $N$
10     $[a_j]_{q_i} = \texttt{ShoupModMul}_{q_i}([a_j]_{q_i}, [\hat{q}_i^{-1}]_{q_i})$
11 end procedure
12 procedure FastBasisConversionStep2                    // launch $l'N$ threads
13     idx = blockDim.x * blockIdx.x + threadIdx.x
14     $i$ = idx / $N$
15     $j$ = idx % $N$
16     accum = $[a_j]_{q_0} \cdot [\hat{q}_0]_{q'_i}$
17     for $m$ in $[1, l)$ do
18         accum += $[a_j]_{q_m} \cdot [\hat{q}_m]_{q'_i}$
19     end
20     $[a'_j]_{q'_i} = \texttt{BarrettModReduction}_{q'_i}$(accum)
21 end procedure
```

Then the thread block, which consists of $T_{\frac{R}{8}}$ threads, performs NTT along the z axis first (using the twiddle factors inside the global memory), stores the outputs to shared memory, synchronizes, loads the stored data in shared memory to the register file, and repeats for the y and x axes. Because each thread performs up to 8-point NTT, we call this as 8 per-thread NTT.

Bank conflicts in shared memory occurs with the naïve implementation. To reduce the bank conflicts, we add appropriate offsets to the base address of the shared memory space allocated for each kernel (i.e., padding to the shared memory) as in [GLD+08].

Algorithm 12 shows the implementation of the fast basis conversion on GPU. It consists of two kernels, where each index of data that a thread processes (**idx**) is characterized by the index of the corresponding thread block (blockIdx.x), dimension of the block (blockDim.x), and thread index inside the warp (threadIdx.x).

We recommend more interested readers to check out the code in https://github.com/scale-snu/ckks-gpu-core.

D Details on CKKS subroutines

This section provides supplementary explanations for the subroutines referred in the main body of the paper to enhance the understanding of the CKKS method, with the required mathematical background.

D.1 Notation in HE

Table 5 summarizes the symbols and their descriptions used throughout this paper.

Table 5: Notation for HE used in this paper.

$i, j, \log_2 N$	$\in \mathbb{N}$	q_i, p_i	Prime numbers
$\mathcal{R}$	$\mathbb{Z}[X]/(X^N+1)$	$\mathcal{R}_{q_i}$	$\mathbb{Z}_{q_i}[X]/(X^N+1)$
$\mathcal{B}$	$\{p_0,\ldots,p_{k-1}\}$	$\mathcal{C}_i$	$\{q_0,\ldots,q_i\}$
$\mathcal{D}_i$	$(\mathcal{B}\bigcup(\bigcup_{0\le j<i}\mathcal{C}'_j)$	$m(X)$	Coefficient-wise representation of $\sum_{i=0}^{N-1} m_i X^i \in R$
$[s(X)]_{q_j}$	Coefficient-wise $mod\ q_j$	$[a]_{q_j}$	$a\ mod\ q_j$
$\odot$	Hadamard multiplication	$[\![s(X)]\!]_{\mathcal{C}_i}$	$([s(X)]_{q_0},\cdots,[s(X)]_{q_i})$
Q_i	$\prod_{j=0}^{i} q_j$	L	Maximum level of ciphertext
$\hat{Q}_j$	$\prod_{i=0\wedge i\neq j}^{\mathtt{dnum}-1} Q'_i$	ℓ	Current level of ciphertext
$\hat{q}_j$	$\prod_{i=0\wedge i\neq j}^{L} q_j$	χ_{key}	Secret key distribution [CKKS17]
P	$\prod_{i=0}^{k-1} p_i$	ω_{q_i}	Primitive Nth root of unity in R_{q_i}
α	$(L+1)/\ \mathtt{dnum}$	β	$\lceil(\ell+1)/\alpha\rceil$
$\{Q'_j\}_{j\in[0,\mathtt{dnum}]}$	$\{\prod_{i=j\alpha}^{(j+1)\alpha-1} q_i\}_{j\in[0,\mathtt{dnum})}$	$[a,b]$	$\{n\in\mathbb{Z} \| a\le n\le b\}$
$\mathcal{C}'_i$	$\{q_{i\alpha},\ldots,q_{i\alpha+\alpha-1}\}$.	**ct**	Ciphertext whose form is given as $(a,b)\in(\prod_i (Z^*_{q_i})^N)^2$
Q'	$\prod_{i=\ell+1}^{\alpha\beta-1} q_i$	$\hat{Q}$	$P\cdot Q'$
$\mathcal{S}_Y,\mathcal{S}'_Y$	subsets of Y s.t. $\mathcal{S}_Y\cap\mathcal{S}'_Y=\phi$, where $Y\in\{\mathcal{B},\mathcal{C}_L\}$	$\hat{Q}'''_j$	$\prod_{q_i\in\mathcal{S}_{\mathcal{C}_L}} q_i\times\prod_{p_i\in\mathcal{S}_{\mathcal{B}}\wedge i\neq j} p_i$
A_i	$[i\alpha, i\alpha+\alpha-1]$	$\hat{Q}''_j$	$\prod_{q_i\in\mathcal{S}_{\mathcal{C}_L}\wedge i\neq j} q_i\times\prod_{p_i\in\mathcal{S}_{\mathcal{B}}} p_i$
$\pi_n(a)$	A permutation on $[0,N-1]$. $a\mapsto([\![5^n(2a+1)]\!]_{2N}-1)/2$. Please refer to D.3 for more info.		
$\mathtt{dnum}$	RNS-decomposition number [HK20] Please refer to D.7 for more information.		
slot	Position of a plaintext in a ciphertext that contains a vector of plaintexts		
χ_{err}	Error distribution used for encryption and key generation [HK20].		
k	Number of prime moduli used for `ModUp`/`ModDown`.		
$\mathtt{NTT}([\![s(X)]\!]_{q_i})$	NTT op. returning $s^{(i)}=\left([s(\omega^0_{q_i})]_{q_i},\ \cdots,\ [s(\omega^{N-1}_{q_i})]_{q_i}\right)\in(\mathbb{Z}^*_{q_i})^N$		
$\mathtt{NTT}([\![s(X)]\!]_{\mathcal{C}_i})$	Executes $(\mathtt{NTT}([\![s(X)]\!]_{q_i}))_{q_i\in\mathcal{C}_i}$ then returns $s=[s]_{\mathcal{C}_i}=(s^{(0)},\cdots,s^{(i)})$		
$\mathtt{iNTT}([\![s]\!]_{\mathcal{C}_i})$	Inverse NTT op. returning $[s(X)]_{\mathcal{C}_i}=([s(X)]_{q_j})_{j\in[0,i-1]}$		
$a\xleftarrow{\$}S$	a is (uniformly) sampled from the distribution (or a set) S.		
evk	Key-switching key of the form $(\mathbf{evk}_0,\ldots,\mathbf{evk}_{\mathtt{dnum}-1})$,where $\mathbf{evk}_i=(evk_i^{(j)})_{j\in[0,k+L]}\in\prod_{n=0}^{k-1}((\mathbb{Z}^*_{p_n})^N)^2\times\prod_{n=0}^{L}((\mathbb{Z}^*_{q_n})^N)^2$		

D.2 $\mathsf{RLWE}_{\mathcal{C}_i}(s,m)$

CKKS uses RLWE instances for key generation and encryption. For example, to encrypt a plaintext polynomial $m(X)\in\mathcal{R}_Q$, CKKS first generates $(a(X),b(X))$, which is a pair of elements in $\mathcal{R}_Q$ where $a(X)\xleftarrow{\$}\mathcal{R}_Q$, $b(X)\leftarrow a(X)\cdot s(X)+e(X)$, and $e(X)\xleftarrow{\$}\chi_{\mathsf{err}}$. After generating the pair, by adding $m(X)$ to $b(X)$ we complete the encryption. Here, $e(X)\in\mathcal{R}_Q$, and since the absolute value of a coefficient of $e(X)$ is about several tens

at maximum with very high probability [CKKS17], the bit lengths of $e(X)$'s coefficients are very short. $s(X) \in \mathcal{R}_Q$ is the secret key, and only with this, $m(X)+e(X)$ can be recovered through $(a(X), b(X)) \cdot (-s(X), 1)$.

We can use CRT to convert an element in $\mathcal{R}_{Q_\ell}$ into one in $\prod_{j=0}^{\ell} \mathcal{R}_{q_j}$. By applying NTT further, finally it can be expressed as an element in $\prod_{j=0}^{\ell} (\mathbb{Z}_{q_j}^*)^N$. That is, the following relations are preserved through CRT and NTT:

$$a(X) \in \mathcal{R}_{Q_\ell} \underset{\text{iCRT}}{\overset{\text{CRT}}{\rightleftarrows}} (a_{q_j}(X))_{j\in[0,\ell]} \in \prod_{j=0}^{\ell} \mathcal{R}_{q_j} \underset{\texttt{iNTT}}{\overset{\texttt{NTT}}{\rightleftarrows}} (a^{(j)})_{j\in[0,\ell]} \in \prod_{j=0}^{\ell} (\mathbb{Z}_{q_j}^*)^N$$

The algorithm $\mathsf{RLWE}_{\mathcal{C}_i}(s,m)$ in the main body of the text generates an RLWE instance in NTT domain $(a,b) \in (\prod_{j=0}^{\ell}(\mathbb{Z}_{q_j}^*)^N)^2$ with s and m, each of which is a secret key $s(X)$ and message $m(X)$ in NTT domain, respectively; the output (a,b) can be converted from and to $(a(X), b(X))$ in $(\mathcal{R}_{Q_\ell})^2$.

D.3 `FrobeniusMap`(a,n)

This algorithm performs Frobenius map function in NTT domain for `HROTATE`. In CKKS, a vector $\vec{z}$ of complex numbers of dimension $M \le N/2$ can be converted into a plaintext polynomial $m(X) \in \mathcal{R}_{Q_\ell}$

$$\vec{z} \in \mathbb{C}^M \xrightarrow{\tau^{-1}} (m_{\mathbb{Q}}(X)) \in \mathbb{Q}(X)/(X^N+1) \xrightarrow{\lfloor \Delta \cdot \rceil} m(X) \in \mathcal{R}_{Q_\ell} \quad (2)$$

where $\tau : z_i \mapsto m_{\mathbb{Q}}(\zeta_i)$ and $\vec{z} = (z_i)_{i\in[0,N-1]}$, $\zeta_i = \zeta^{5^i}$, $\zeta = exp(-2\pi i/4M)$, and $\lfloor \Delta \cdot \rceil : a \mapsto \lfloor \Delta \cdot a \rceil$. In this case, $\zeta_M = \zeta_0 = \zeta$. We define the result obtained by substituting ζ_i for X in $m_{\mathbb{Q}}(X)$ as the value of the i-th slot.

Now we consider the slot rotation operation. For example, assuming that the rotation by n slots is performed in $m_{\mathbb{Q}}(X)$, the resultant polynomial $m'_{\mathbb{Q}}(X) \in \mathbb{Q}[X]/(X^N+1)$ should preserve $m_{\mathbb{Q}}(\zeta_i) = m'_{\mathbb{Q}}(\zeta_{i+n})$. That is, the value in the $(i+n)$-th slot of $m'_{\mathbb{Q}}(X)$ should be the same as the one in the i-th slot of $m_{\mathbb{Q}}(X)$. Because $\zeta_{i+n} = \zeta^{5^{i+n}} = \zeta_i^{5^n}$, we need to compute $m'_{\mathbb{Q}}(X) = m_{\mathbb{Q}}(X^{5^{-n}})$ from $m_{\mathbb{Q}}(X)$.

The above relation holds on ciphertexts. For example, to rotate the hidden message in a ciphertext $(a(X), b(X) = a(X)\cdot s(X) + m(X) + e(X)) \in \mathcal{R}_{Q_\ell}^2$ by n slots, we need to derive $(a(X^{5^{-n}}), b(X^{5^{-n}}) = a(X^{5^{-n}}) \cdot s(X^{5^{-n}}) + m(X^{5^{-n}}) + e(X^{5^{-n}}))$ from $(a(X), b(X))$.

Let us consider $a(X)$ (because the same applies to $b(X)$). As described in the previous section, we can convert $a(X)$ to $(a^{(j)})_{j\in[0,\ell]} \in \prod_{j=0}^{\ell}(\mathbb{Z}_{q_j}^*)^N$ through CRT and NTT. Let's focus on one element $a^{(i)} = \overbrace{([a(\omega_0)]_{q_i}, \cdots, [a(\omega_{N-1})]_{q_i})}^{(A)} \in (\mathbb{Z}_{q_i}^*)^N$ where ω_j is $\omega_{q_i}^j$ in the main body of the text ($j \in [0, N-1]$). In order to rotate the hidden message by n slot, we need to calculate $\overbrace{([a(\omega_0^{5^{-n}})]_{q_i}, \cdots, [a(\omega_{N-1}^{5^{-n}})]_{q_i})}^{(B)}$, which is the result of applying NTT to $[a(X^{5^{-n}})]_{q_i}$. Fortunately, Equation (A) above is a permutation of Equation (B) (i.e., (B)=$\overbrace{([a(\omega_{\pi_n^{-1}(0)})]_{q_i}, \cdots, [a(\omega_{\pi_n^{-1}(N-1)})]_{q_i})}^{(C)}$). Here, $\pi_n : [0, N-1] \to [0, N-1]$ is a permutation that satisfies $\omega_j^{5^n} = \omega_{\pi_n(j)}$.

Now let's discuss `FrobeniusMap`(a,n). $a = (a^{(j)})_{j\in[0,N-1]} \in (\mathbb{Z}_{q_j}^*)^N$. It is a subroutine to calculate the rotation result by n-slots for all $a^{(j)}$s in a by the way described in Equation (C).

D.4 $\texttt{Conv}_{\mathcal{S}_{\mathcal{C}_L}\cup\mathcal{S}_{\mathcal{B}}\to\mathcal{S}'_{\mathcal{C}_L}\cup\mathcal{S}'_{\mathcal{B}}}([a(X)]_{\mathcal{S}_{\mathcal{C}_L}\cup\mathcal{S}_{\mathcal{B}}})$

This subroutine refers to the fast basis conversion algorithm [CHK+19, BEHZ16a]. Given the result of modular operations with the prime numbers in $\mathcal{S}_{\mathcal{C}_L}\cup\mathcal{S}_{\mathcal{B}}$ for a polynomial $a(X)$, the purpose of the algorithm is to produce the result of modular operations on the coefficients of $a(X)$ with the different prime numbers in $\mathcal{S}'_{\mathcal{C}_L}\cup\mathcal{S}'_{\mathcal{B}}$.

More precisely, given $[a(X)]_{\mathcal{S}_{\mathcal{C}_L}\cup\mathcal{S}_{\mathcal{B}}}$ as an element in $\prod_{q_i\in\mathcal{S}_{\mathcal{C}_L}}\mathcal{R}_{q_i}\times\prod_{p_j\in\mathcal{S}_{\mathcal{B}}}\mathcal{R}_{p_j}$, the algorithm converts it into an element $[a(X)]_{\mathcal{S}'_{\mathcal{C}_L}\cup\mathcal{S}'_{\mathcal{B}}}$ in $\prod_{q_i\in\mathcal{S}'_{\mathcal{C}_L}}\mathcal{R}_{q_i}\times\prod_{p_j\in\mathcal{S}'_{\mathcal{B}}}\mathcal{R}_{p_j}$ with the prime numbers given in $\mathcal{S}'_{\mathcal{C}_L}\cup\mathcal{S}'_{\mathcal{B}}$.

To obtain the result above, one can simply use iCRT for the coefficient of $[a(X)]_{\mathcal{S}_{\mathcal{C}_L}\cup\mathcal{S}_{\mathcal{B}}}$ and then perform the modular reduction operation on the coefficient using each of the prime modulus values to represent the target rings in the result. However, if the number of target rings is large, many long-precision operations should be conducted, which exhibits huge computation cost. For example, if iCRT is performed to $(a_{q_j}(X))_{q_j\in\mathcal{C}_\ell}$ which is an element of $\mathcal{R}_{\mathcal{C}_\ell}$, the result, $A(X)\in\mathcal{R}$, can be calculated with the following formula:

$$A(X)\leftarrow(\sum_{j=0}^{\ell}(((\hat{q}_j^{-1} \bmod q_j)\cdot[a(X)]_{q_j}) \bmod q_j)\cdot(\hat{q}_j)) \bmod Q_\ell, \tag{3}$$

where $\hat{q}_j=\prod_{i\in[0,\ell]\wedge i\neq j}q_i$. The bit length of $\hat{q}_j$ is much larger than that of the primes that are single- or double-word, increasing the computation cost.

The fast basis conversion algorithm directly retrieves $[a(X)]_{\mathcal{S}'_{\mathcal{C}_L}\cup\mathcal{S}'_{\mathcal{B}}}$ by computing the following equation for each $q'_i\in\mathcal{S}'_{\mathcal{C}_L}\cup\mathcal{S}'_{\mathcal{B}}$:

$$(\sum_{j=0}^{\ell}(((\hat{q}_j^{-1} \bmod q_j)\cdot[a(X)]_{q_j}) \bmod q_j)\cdot(\hat{q}_j \bmod q'_i)) \bmod q'_i. \tag{4}$$

The algorithm uses the fact that only the prime numbers in a specific set are used to represent polynomial rings used in CKKS. The prime numbers used are all elements of $\mathcal{C}_L\cup\mathcal{B}$. Thus, if we pre-compute $[\hat{Q}_j''^{-1}]_{q_j},[\hat{Q}_j'']_{q_j}$ for all $q_j\in\mathcal{C}_L$, and $[\hat{Q}_j'''^{-1}]_{p_j},[\hat{Q}_j''']_{p_j}$ for all $p_j\in\mathcal{B}$ in advance, we obtain the result with much less computation cost compared to using iCRT. Line 2-4 of Algorithm 1 of the main body of this paper are the main computation steps for the above formula. Because the bit lengths of $[\hat{Q}_j'']_{q_i}$ and $[\hat{Q}_j''']_{q_i}$ are at most that of a single prime number, they can be performed efficiently.

In Equation 4, '$\bmod p_i$' operation is performed without performing the last '$\bmod Q_\ell$' in Equation 3. Thus, there is a difference in the result compared to executing modular reduction by p_i to the result of running Equation 4. However, since the calculation result is returned to $\mathcal{R}_{Q_\ell}$ through `modDown` in the future, the error can be ignored.

D.5 $\texttt{ModUp}_{\mathcal{C}'_i\to\mathcal{D}_\beta}([a]_{\mathcal{C}'_i})$

This algorithm takes $[a]_{\mathcal{C}'_i}\in\prod_{j=i\cdot\alpha}^{i\cdot\alpha+\alpha-1}(\mathbb{Z}^*_{q_j})^N$ as input and changes its basis to $\mathcal{D}_\beta$ including the existing basis $\mathcal{C}'_i$ using the fast basis conversion: $[a]_{\mathcal{D}_\beta}\in\prod_{j=0}^{i\cdot\alpha+\alpha-1}(\mathbb{Z}^*_{q_j})^N\times\prod_{j=0}^{k-1}(\mathbb{Z}^*_{p_j})^N$. The input $[a]_{\mathcal{C}'_i}$ is one of the decomposed part of $[a]_{C_\ell}$ after the RNS decomposition that will be described later.

Specifically, in the basis conversion, $\texttt{Conv}_{\mathcal{C}'_i\to\mathcal{D}_\beta-\mathcal{C}'_i}([a]_{\mathcal{C}'_i})$ is performed and the result is concatenated with the input $[a]_{\mathcal{C}'_i}$ to finally obtain the result. Since the input of `Conv` operates only on a polynomial element of coefficient-wise representation, `iNTT` is performed to convert $[a]_{\mathcal{C}'_i}$ into $[a(X)]_{\mathcal{C}'_i}\in\prod_{q_j\in\mathcal{C}'_i}\mathcal{R}_{q_j}$ before executing `Conv`.

D.6 $\texttt{ModDown}_{\mathcal{D}_\beta \to \mathcal{C}_\ell}(\tilde{b}^{(0)}, \tilde{b}^{(1)}, \cdots, \tilde{b}^{(k+\alpha\beta-1)})$

Suppose $P = \prod_{r_i \in \mathcal{D}_\beta - \mathcal{C}_\ell} r_i$ and $Q = \prod_{r_i \in \mathcal{C}_\ell} r_i$. In this paper, the elements of $\mathcal{D}_\beta - \mathcal{C}_\ell$ are expressed in the form of either p_i or q_j, However, we use r_i instead to simplify the expression.

If iNTT and iCRT are executed for each coefficient of the input $\tilde{b}^{(0)}, \tilde{b}^{(1)}, \cdots, \tilde{b}^{(k+\alpha\beta-1)}$, it can be expressed as $[\tilde{b}(X)]_{PQ} \in \mathcal{R}_{PQ}$. ModDown basically does the following equations but in RNS domain:

$$\begin{aligned} t(X) &\leftarrow [\tilde{b}(X)]_{PQ} - [\tilde{b}(X)]_{PQ} \bmod P \\ b(X) &\leftarrow t(X)/P \end{aligned}$$

ModDown performs the above formula on RNS with $([\tilde{b}(X)]_{r_i})_{r_i \in \mathcal{D}_\beta} \in \prod_{r_i \in \mathcal{D}_\beta} \mathcal{R}_{r_i}$, which is approximately $[\tilde{b}(X)]_{PQ}$ after applying CRT.

For $r_i \in \mathcal{D}_\beta - \mathcal{C}_\ell$, $([\tilde{b}(X)]_{r_i} - [\tilde{b}(X)]_{r_i} \bmod P) \bmod r_i$ equals to zero. Therefore, the output becomes $([b(X)]_{r_i})_{r_i \in \mathcal{C}_\ell}$ where

$$[b(X)]_{r_i} = [P^{-1}(\overbrace{([\tilde{b}(X)]_{PQ} \mod r_i}^{\text{(D)}} - \overbrace{([\tilde{b}(X)]_{PQ} \mod P) \mod r_i)}^{\text{(E)}}]_{r_i}.$$

(D) is the same as the provided input $[\tilde{b}(X)]_{r_i}$. (E) should be calculated using the fast basis conversion $(\texttt{Conv}_{\mathcal{D}_\beta - \mathcal{C}_\ell \to \mathcal{C}_\ell}(\cdot))$. The evaluation of (E) is $[b'(X)]_{r_i \in \mathcal{C}_\ell} = \texttt{Conv}_{\mathcal{D}_\beta - \mathcal{C}_\ell \to \mathcal{C}_\ell}(([\tilde{b}(X)]_{r_i})_{r_i \in \mathcal{D}_\beta - \mathcal{C}_\ell})$. In conclusion, the following formula can be derived from the above:

$$\overbrace{[b(X)]_{r_i} = [P^{-1}([\tilde{b}(X)]_{r_i} - [b'(X)]_{r_i})]_{r_i}}^{\text{(F)}} \quad (5)$$

ModDown first converts the inputs $(\tilde{b}^{(0)}, \tilde{b}^{(1)}, \cdots, \tilde{b}^{(k+\alpha\beta-1)})$ to $([\tilde{b}(X)]_{r_i})_{r_i \in \mathcal{D}_\beta}$ using iNTT and evaluates $([\tilde{b}(X)]_{r_i})_{r_i \in \mathcal{D}_\beta}$. Finally, it performs subtractions and multiplication with P^{-1} to obtain the desired result.

Please be aware that in the description of ModDown in Algorithm 4, NTT is executed before executing (F). Since Equation (F) still holds although we replace X to ω_j $(j \in [0, N-1])$, it is possible to execute (F) after applying NTT; thus, we obtain $b^{(i)} = ([b(\omega_{q_i}^{(j)})])_{j \in [0, N-1]})$ for every $i \in [0, \ell]$.

D.7 RNS Decomposition ($\texttt{Dcomp}(d = (d^{(0)}, \cdots, d^{(\ell)}))$)

Recall $Q_\ell = \prod_{i=0}^{\ell} q_i$. Suppose there are two elements A and B in $\mathbb{Z}_{Q_\ell}^*$. We can calculate Z on the equation below.

$$Z = \sum_{i=0}^{\ell}((A \cdot \hat{q}_i^{-1}) \bmod q_i) \cdot (\hat{q}_i \cdot B)) \bmod Q_\ell \quad (6)$$

We can see that $Z \bmod q_i = A \cdot B \bmod q_i$ is satisfied for all q_i $(i \in [0, L])$. Therefore, $Z \bmod Q_\ell = AB \bmod Q_\ell$ by CRT.

The above relationship holds true also for $A(X), B(X) \in \mathcal{R}_{Q_\ell}$ and $Z(X) \in \mathcal{R}_{Q_\ell}$, which are corresponding to A, B, and Z, respectively. Since we are using an RNS-variant of CKKS, suppose that $(A(X) \cdot \hat{q}_i^{-1}) \bmod q_i) \cdot (\hat{q}_i \cdot B(X))$ is performed in $\prod_{j=0}^{\ell} \mathcal{R}_{q_j}$ after applying CRT on $A(X)$ and $B(X)$. Then, the term $\hat{q}_i \cdot B(X) \bmod q_j$ becomes all zero for all j where $j \neq i$. Eventually, the following relationship is satisfied:

$$\begin{aligned}&\textstyle\sum_{i=0}^{\ell}((A(X)\cdot\hat{q}_i^{-1}) \bmod q_i)\cdot(\hat{q}_i\cdot B(X))) \bmod Q_\ell\\ &\xrightarrow{CRT} (((A(X)\cdot\hat{q}_i^{-1}) \bmod q_i)\cdot(\hat{q}_i\cdot B(X)) \bmod q_i)_{i\in[0,\ell]},\end{aligned} \tag{7}$$

where $\xrightarrow{CRT}$ refers to applying CRT.

This method can be applied to the multiplication on ciphertexts to increase the maximum level for a fresh ciphertext on the same parameter set compared to where the method is not applied. We call the above technique as RNS-decomposition.

Instead of performing decomposition on each of individual prime numbers $\{q_i\}_{i\in[0,\ell]}$, the above method can be applied per group of primes, after grouping L prime numbers used in the scheme. Han et al. [HK20] define the parameter α as the number of prime numbers in a group, which can be calculated by $(L+1)/\mathsf{dnum}$, where dnum, the decomposition number, is an arbitrarily chosen parameter; dnum is normally selected from the set of the divisors of $L+1$. Then, the above Equation 7 changes to work with groups of prime numbers as below.

$$\begin{aligned}&\textstyle\sum_{i=0}^{\beta-1}((A(X)\cdot\hat{Q}_i^{-1}) \bmod Q_i)\cdot(\hat{Q}_i\cdot B(X))) \bmod Q_{\alpha\beta}\\ &\xrightarrow{CRT} (\overbrace{((A(X)\cdot\hat{Q}_i^{-1}) \bmod q_{i\alpha+j})}^{(G)}\cdot(\hat{Q}_i\cdot B(X)) \bmod q_{i\alpha+j})_{i\in[0,\beta-1],j\in[0,\alpha-1]},\end{aligned} \tag{8}$$

where $\alpha,\beta\in\mathbb{Z},\beta=\lceil\frac{\ell+1}{\alpha}\rceil$, and $\hat{Q}_i=\prod_{j=\alpha\cdot i}^{\alpha\cdot i+\alpha-1}q_j$ $(i\in[0,\beta-1])$. We have to set $A(X) \bmod q_{\ell+1}=\cdots=A(X) \bmod q_{i\cdot\alpha+\alpha-1}=0$ such that Equation 8 also holds true when $i\cdot\alpha<\ell<i\cdot\alpha+\alpha-1$.

Now we provide the description on $\mathtt{Dcomp}(d=(d^{(0)},\cdots,d^{(\ell)}))$ in Section 2. It executes the process (G) in Equation 8 above in NTT domain. We omitted the description on other parameters such as $\alpha,\mathsf{dnum},\ell,L$ in the description of $\mathtt{Dcomp}$.