Ciphertext-Ciphertext Matrix Multiplication: Fast for Large Matrices

Jai Hyun Park

CryptoLab Inc., Lyon, France

Abstract. Matrix multiplication of two encrypted matrices (CC-MM) is a key challenge for privacy-preserving machine learning applications. As modern machine learning models focus on scalability, fast CC-MM on large datasets is increasingly in demand.
In this work, we present a CC-MM algorithm for large matrices. The algorithm consists of plaintext matrix multiplications (PP-MM) and ciphertext matrix transpose algorithms (C-MT). We propose a fast C-MT algorithm, which is computationally inexpensive compared to PP-MM. By leveraging high-performance BLAS libraries to optimize PP-MM, we implement large-scale CC-MM with substantial performance improvements. Furthermore, we propose lightweight algorithms, significantly reducing the key size from 1 960 MB to 1.57 MB for CC-MM with comparable efficiency.
In a single-thread implementation, the C-MT algorithm takes 0.76 seconds to transpose a 2 048 $\times$ 2 048 encrypted matrix. The CC-MM algorithm requires 85.2 seconds to multiply two 4 096 $\times$ 4 096 encrypted matrices. For large matrices, our algorithm outperforms the state-of-the-art CC-MM method from Jiang-Kim-Lauter-Song [CCS'18] by a factor of over 800.

1 Introduction

Ciphertext-ciphertext matrix multiplication (CC-MM) takes as input two bundles of ciphertext(s) encrypting two input matrices and outputs ciphertext(s) encrypting the product matrix. CC-MM plays a central role in privacy-preserving machine learning (PPML) when a server trains or performs inference on machine learning models using encrypted data from the client. For example, during privacy-preserving training and inference of large language models (LLM) in [PZM$^+$24,HLC$^+$22,ZLY$^+$23], CC-MM takes an important role.

As mentioned in [BCH$^+$24], homomorphic multiplication with large matrices appears in various steps during PPML. The authors pointed out that the matrix dimension often ranges up to 16 384 (in GPT-3.5) and 18432 (in PaLM 540B) for privacy-preserving inference of LLMs. As we consider privacy-preserving training and inference of such LLMs, fast CC-MM for large-dimensional matrices is necessary.

However, the existing CC-MM algorithms are much slower than plaintext-plaintext matrix multiplication (PP-MM) or plaintext-ciphertext matrix multiplication (PC-MM). To take a notable example from [JKLS18], it takes 0.6

seconds to multiply two square matrices of dimension 64. To multiply square matrices of a larger dimension, 4 096, for instance, [JKLS18] would require more than 19 hours, even with the Strassen algorithm [Str69]. We note that most of the current CC-MM implementations are based on [JKLS18] or its variants.

As a reference point, for plaintext-ciphertext matrix multiplication (PC-MM), a recent work [BCH$^+$24] significantly accelerated it to take 17.1 seconds to multiply square matrices of dimension 4 096 in a single thread CPU. For plaintext-plaintext matrix multiplication (PP-MM), by utilizing highly optimized linear algebra libraries (BLAS libraries), it takes 1.5 seconds for square matrices of dimension 4 096. The inefficiency of CC-MM has been one of the main bottlenecks for practical privacy-preserving training and inference of large neural networks, including LLMs such as GPT [RNSS18], BERT [DCLT18] and LLaMA [TLI$^+$23].

The difficulty of CC-MM stems from the complicated structure of ciphertexts. State-of-the-art CC-MM algorithms rely on fully homomorphic encryption (FHE) schemes based on the ring learning-with-error problem [SSTX09,LPR10] (RLWE). RLWE-based FHE schemes [BGV14,Bra12,FV12,CKKS17] encrypt a vector in a ciphertext, requiring "key switching" operations to arrange the vector. Key switching operations during CC-MM introduce computational overhead and disrupt the memory access pattern, significantly degrading the efficiency. Consequently, CC-MM has been significantly slower than optimized PP-MM implementations, such as those in BLAS libraries. This inefficiency becomes even more severe for large matrices, highlighting the need for more efficient CC-MM algorithms for large matrices.

1.1 Contributions

Our main result is a fast CC-MM algorithm for large matrices. The algorithm focuses on matrices whose dimensions are at least the RLWE ring degree. The concrete efficiency is supported by experiments. We also provide a variant of the CC-MM algorithm with a small size of keys and comparable efficiency.

Our CC-MM algorithm consists of reducing CC-MM to four modular PP-MMs, where PP-MMs have the same dimension as the given CC-MM. From the reduction, we take full advantage of the high efficiency of existing PP-MM libraries (BLAS libraries). Our strategy is inspired by prior reductions from PC-MM to PP-MM introduced in [BCH$^+$24]. This prior work significantly improved the efficiency of PC-MM, but we note that the reduction from CC-MM to PP-MM has remained an open question. While a pre-FHE scheme [GHV10] provides a hint, it supports only a single CC-MM, making further multiplications difficult. In this work, we present a reduction from CC-MM in FHE settings to PP-MM and demonstrate its effectiveness in accelerating CC-MM.

As a main tool for our CC-MM algorithm, we propose a fast ciphertext matrix transpose (C-MT) algorithm. We devise a new C-MT algorithm with a divide-and-conquer approach. Our algorithm uses $\tilde{O}(N^2)$ bit operations. We note that C-MT is independent of CC-MM and has broader applications. For example, during PPML scenarios, one often needs to convert the client-wise encrypted ciphertexts into feature-wise encrypted ciphertexts and vice versa. Our C-MT algorithm enables this conversion efficiently.

In addition, we provide lightweight variants of CC-MM and C-MT algorithms, which use fewer evaluation keys with comparable efficiency to the prior algorithms. These lightweight algorithms address the large key size of our CC-MM and C-MT algorithms. For instance, for the CC-MM algorithm, the lightweight modification reduces the key size from 1 960 MB to 1.57 MB.

We implemented our algorithms in the HEaaN library [HEa22]. Our implementation takes 85.2 seconds to multiply two 4 096 $\times$ 4 096 encrypted matrices using CKKS ciphertext of degree 2^{12} in a single thread. For ciphertext matrix transpose, it takes 0.76 seconds for a square matrix of dimension 2 048. For lightweight CC-MM, our implementation uses 1.57 MB of keys, and for lightweight C-MT, it uses 0.246 MB, taking 672 seconds for square matrices of dimension 8 192 and 4.92 seconds for a square matrix of dimension 4 096, respectively. Although we focus on the CKKS scheme, we note that our algorithms are also applicable to BGV and FV schemes.

1.2 Technical overview

We construct a reduction from CC-MM to PP-MM, which enables performing CC-MM by taking full advantage of highly optimized BLAS libraries such as OpenBLAS [oBlv], LAPACK [ABB+99], and FLINT [tea24]. This strategy improves the practical latency significantly. It is inspired by [BCH+24], which reduces PC-MM to PP-MM, achieving a significant speed up for PC-MM.

In this overview, we first explain our new C-MT algorithm. Then, we introduce the matrix form of encryptions, which represents RLWE-based ciphertexts encrypting either the rows or the columns of a matrix. Using the representation and C-MT algorithm, we describe our reduction from CC-MM to PC-MM and introduce our fast CC-MM algorithm. Finally, we explain the lightweight variants of our algorithms to reduce the key size.

Ciphertext matrix transpose. Before constructing the reduction from CC-MM to PP-MM, we introduce the main tool, ciphertext matrix transpose (C-MT). C-MT takes as input ciphertexts encrypting a matrix row-by-row (resp. column-by-column), and returns ciphertexts encrypting the same matrix column-by-column (resp. row-by-row). In this work, we focus on large matrices where each ciphertext encrypts only one row (or column). While we extensively utilize it for CC-MM, we also note that C-MT is an interesting problem beyond its application as a tool for CC-MM.

We start from the well-known observation on ring $\mathcal{R}_Q = \mathbb{Z}_Q[X]/(X^N + 1)$ that

$$N \cdot m_j = \sum_{\sigma \in \mathrm{Gal}(\mathcal{R}/\mathbb{Z})} \sigma(X^{-j} \cdot m)$$

for each $j = 0, 1, \cdots, N-1$, where $m(X) = \sum_{i=0}^{N-1} m_i X^i$ is an element in $\mathcal{R}_Q$, and $\mathrm{Gal}(\mathcal{R}/\mathbb{Z})$ is the group of automorphisms of $\mathcal{R}$ induced by Galois automorphisms in $\mathbb{Q}[X]/(X^N+1)$ that fixes $\mathbb{Q}$.

For given N plaintexts $\{m_i = \sum_j M_{i,j} X^j\}_{0 \le i < N}$ in $\mathcal{R}_Q$ that each stores one row of an $N \times N$ matrix $\mathbf{M}$, the plaintexts $\{m'_j\}_{0 \le j < N}$ that store the transpose matrix $\mathbf{M}^t$ is:

$$m'_j = \sum_{i=0}^{N-1} M_{i,j} X^i = \sum_{i=0}^{N-1} \left(N^{-1} \cdot \sum_{\sigma \in \mathrm{Gal}(\mathcal{R}/\mathbb{Z})} \sigma(X^{-j} \cdot m_i) \right) \cdot X^i$$
$$= N^{-1} \cdot \sum_{\sigma \in \mathrm{Gal}(\mathcal{R}/\mathbb{Z})} \sigma \left(\sum_{i=0}^{N-1} m_i \cdot \sigma^{-1}(X^i) \right) \sigma(X^{-j})$$

for each $j = 0, 1, \cdots N-1$. The goal of C-MT is to obtain $\{m'_j\}_{0 \le j < N}$ from $\{m_i\}_{0 \le i < N}$ in the encrypted state.

We proceed in three steps:

1. Computing $\{\widetilde{m}_\sigma = \sum_i m_i \cdot \sigma^{-1}(X^i)\}_{\sigma \in \mathrm{Gal}(\mathcal{R}/\mathbb{Z})}$ from $\{m_i\}_{0 \le i < N}$,
2. Computing $\{\bar{m}_\sigma = \sigma(\widetilde{m}_\sigma)\}_{\sigma \in \mathrm{Gal}(\mathcal{R}/\mathbb{Z})}$ using automorphisms, and
3. Computing $\{m'_j = \sum_\sigma \bar{m}_\sigma \cdot \sigma(X^{-j})\}_{0 \le j < N}$ from $\{\bar{m}_\sigma\}_{\sigma \in \mathrm{Gal}(\mathcal{R}/\mathbb{Z})}$.

We compute the second step in the encrypted state using N key switching operations, which costs $\tilde{O}(N^2)$ bit operations. For steps 1 and 3, we devise a fast divide-and-conquer algorithm, reducing the cost to $O(N^2 \log N)$. Putting it together, the overall cost of our C-MT algorithm is $\tilde{O}(N^2)$. We refer to Section 3 for details.

Matrix form of encryptions. We start from the matrix form of RLWE-based ciphertexts, which is also introduced in [BCH+24]. The N RLWE-based ciphertexts $(a_i, b_i)_{0 \le i < N}$ encrypts each row of a matrix $\mathbf{M}$ if and only if, over $\mathcal{R}_Q = \mathbb{Z}_Q[X]/(X^N+1)$:

$$\forall i,\ a_i \cdot \mathtt{sk} + b_i \approx \sum_j M_{i,j} X^j,$$

where $\mathtt{sk}$ is the shared secret key. As described in [BCH+24], we can rewrite the above equation in matrix form:

$$\mathbf{A} \cdot \mathtt{Toep}(\mathtt{sk}) + \mathbf{B} \ \approx\ \mathbf{M}, \tag{1}$$

where each row of $\mathbf{A}$ (resp. $\mathbf{B}$) corresponds to the a part (resp. b part) of each ciphertext, and where $\mathtt{Toep}(\mathtt{sk})$ is the Toeplitz matrix of $\mathtt{sk} = \sum_i s_i X^i \in \mathcal{R}$:

$$\mathtt{Toep}(\mathtt{sk}) = \begin{bmatrix} s_0 & s_1 & \cdots & s_{N-1} \\ -s_{N-1} & s_0 & \cdots & s_{N-2} \\ \cdots & \cdots & \ddots & \cdots \\ -s_1 & -s_2 & \cdots & s_0 \end{bmatrix}.$$

We note that each ciphertext encrypts a row of $\mathbf{M}$. We call Eq. (1) as the *matrix form* of row-wise encryptions.

In the same way, we define a matrix form of column-wise encryptions, $(a_j, b_j)_{0\le j<N}$ encrypting each column of a matrix $\mathbf{M}$, as follows.

$$\mathtt{Toep}(\widetilde{\mathtt{sk}}) \cdot \underline{\mathbf{A}} + \underline{\mathbf{B}} \approx \mathbf{M},$$

where each column of $\underline{\mathbf{A}}$ (resp. $\underline{\mathbf{B}}$) corresponds to the a part (resp. b part) of each ciphertext, and $\widetilde{\mathtt{sk}}$ is $\mathtt{sk}(X^{-1})$ in $\mathcal{R}$.

Ciphertext-ciphertext matrix multiplication. We finally explain the reduction from CC-MM to PP-MMs. The basic idea is to multiply two RLWE-based encryptions in matrix forms (Eq. (1)) while preserving the Toeplitz-related structure using C-MT. Assume that we are given two bundles of ciphertexts that encrypt each row of matrices $\mathbf{M}$ and $\mathbf{M}'$, respectively. In matrix form, we are given matrices $\mathbf{A}$, $\mathbf{B}$, $\mathbf{A}'$ and $\mathbf{B}'$ such that in modulo q:

$$\mathbf{A} \cdot \mathtt{Toep}(\mathtt{sk}) + \mathbf{B} \approx \mathbf{M} \quad \text{and} \quad \mathbf{A}' \cdot \mathtt{Toep}(\mathtt{sk}) + \mathbf{B}' \approx \mathbf{M}'$$

Using C-MT algorithm, we transpose the row-wise encryption $(\mathbf{A}, \mathbf{B})$ of $\mathbf{M}$. In matrix form, the C-MT algorithm outputs the column-wise encryption $(\underline{\mathbf{A}}, \underline{\mathbf{B}})$ of $\mathbf{M}$ that satisfies

$$\mathtt{Toep}(\widetilde{\mathtt{sk}}) \cdot \underline{\mathbf{A}} + \underline{\mathbf{B}} \approx \mathbf{M} \quad \text{and} \quad \mathbf{A}' \cdot \mathtt{Toep}(\mathtt{sk}) + \mathbf{B}' \approx \mathbf{M}'.$$

We multiply the above two matrix forms above to obtain

$$\begin{aligned}\mathbf{M}\mathbf{M}' &\approx (\mathtt{Toep}(\widetilde{\mathtt{sk}}) \cdot \underline{\mathbf{A}} + \underline{\mathbf{B}}) \cdot (\mathbf{A}' \cdot \mathtt{Toep}(\mathtt{sk}) + \mathbf{B}') \\ &= \mathtt{Toep}(\widetilde{\mathtt{sk}}) \cdot \mathbf{C}_{0,0} \cdot \mathtt{Toep}(\mathtt{sk}) + \mathtt{Toep}(\widetilde{\mathtt{sk}}) \cdot \mathbf{C}_{0,1} + \mathbf{C}_{1,0} \cdot \mathtt{Toep}(\mathtt{sk}) + \mathbf{C}_{1,1},\end{aligned}$$

where $\mathbf{C}_{0,0} = \underline{\mathbf{A}}\mathbf{A}', \mathbf{C}_{0,1} = \underline{\mathbf{A}}\mathbf{B}', \mathbf{C}_{1,0} = \underline{\mathbf{B}}\mathbf{A}'$, and $\mathbf{C}_{1,1} = \underline{\mathbf{B}}\mathbf{B}'$. We note that the Toeplitz matrices are preserved.

We consider $(\mathbf{C}_{0,0}, \mathbf{O})$ as a column-wise encryption of $\mathtt{Toep}(\widetilde{\mathtt{sk}}) \cdot \mathbf{C}_{0,0}$, and apply C-MT algorithm to it. Then, we obtain a row-wise encryption $(\mathbf{D}_0, \mathbf{D}_1)$ of $\mathtt{Toep}(\widetilde{\mathtt{sk}}) \cdot \mathbf{C}_{0,0}$. We rewrite it in matrix form:

$$\mathtt{Toep}(\widetilde{\mathtt{sk}}) \cdot \mathbf{C}_{0,0} \approx \mathbf{D}_0 \cdot \mathtt{Toep}(\mathtt{sk}) + \mathbf{D}_1.$$

Similarly, we obtain $(\mathbf{D}_2, \mathbf{D}_3)$ such that

$$\mathtt{Toep}(\widetilde{\mathtt{sk}}) \cdot \mathbf{C}_{0,1} \approx \mathbf{D}_2 \cdot \mathtt{Toep}(\mathtt{sk}) + \mathbf{D}_3$$

by applying C-MT to $(\mathbf{C}_{0,1}, \mathbf{O})$. Putting it together, we have

$$\mathbf{M}\mathbf{M}' \approx \mathbf{D}_0 \cdot \mathtt{Toep}(\mathtt{sk}^2) + (\mathbf{D}_1 + \mathbf{D}_2 + \mathbf{C}_{1,0}) \cdot \mathtt{Toep}(\mathtt{sk}) + (\mathbf{D}_3 + \mathbf{C}_{1,1}).$$

After the row-wise relinearization, which is key switchings from $\mathtt{sk}^2$ to $\mathtt{sk}$, we finally obtain a matrix equation

$$\mathbf{A}''\mathtt{Toep}(\mathtt{sk}) + \mathbf{B}'' \approx \mathbf{M}\mathbf{M}'.$$

This is a matrix form of row-wise encryption of the product matrix $\mathbf{MM}'$. This completes CC-MM.

The above protocol reduces CC-MM to four PP-MMs and three C-MTs. The cost of PP-MM is $O(N^\omega)$ where ω is a constant set to 3 in most practical implementations and is at least 2 in theory. Meanwhile, the cost of C-MT is $\tilde{O}(N^2)$, and it is asymptotically negligible compared to PP-MMs.

Lightweight algorithms with small key sizes. There is a potential concern about the large key size of our CC-MM and C-MT algorithms. The previous C-MT algorithm requires N evaluation keys for each of the N homomorphic automorphisms. It turns out to be evaluation keys of size $\tilde{\Omega}(N^2)$, which might be problematic as N is usually large.

To this end, we suggest a lightweight C-MT algorithm that uses only three evaluation keys. The basic idea is to repeatedly update and use a single evaluation key for all homomorphic automorphisms. We need one evaluation key for all homomorphic automorphisms and two other keys to "update" the evaluation key. This idea is motivated by the hierarchical key management system in [LLKN23]. While the update procedure requires additional computation, the asymptotic complexity is the same as the original algorithm.

We also introduce a lightweight CC-MM algorithm. The algorithm follows directly from the lightweight C-MT algorithm, requiring 4 evaluation keys, 3 of which are for the C-MT algorithm and 1 for relinearization.

1.3 Related works

Comparison to [JKLS18]. The seminal work [JKLS18] presents a CC-MM algorithm adopted by most of the current implementations of CC-MM, often with modest modifications [HCJG24,MMJG24,ZL23,HHW+21,GQH+24]. However, it does not reduce CC-MM to PP-MM, and cannot adopt highly optimized BLAS libraries. As a consequence, although [JKLS18] achieves an appropriate bit complexity, its practical performance is several orders of magnitude slower than PP-MM. While it performs reasonably well for small matrices, it becomes impractical as the matrix size increases. On the other hand, we reduce CC-MM to PP-MM, fully leveraging the high efficiency of BLAS libraries, resulting in a significant speedup.

We note that [JKLS18] focuses on relatively small matrices, and our algorithms are derived for large matrices. As recent machine learning models are becoming larger, scalable CC-MM is desirable, particularly in PPML applications based on FHE. Our CC-MM algorithm is particularly beneficial for PPML on large models.

Other approaches of CC-MM. Several other works on CC-MM algorithms exist, including [ZLW23,CYW+24]. However, previous works do not reduce CC-MM to PP-MM, resulting in significantly slower performance in practical implementations. In contrast, our CC-MM algorithm is compatible with conventional FHE settings and fully benefits from the highly optimized BLAS libraries.

In [ZLW23], the authors introduce a CC-MM algorithm based on cyclotomic fields of composite order, defined as the product of three pairwise coprime integers. However, the constraint on the choice of cyclotomic fields restricts the FHE parameters, particularly the ciphertext space, and is generally incompatible with conventional FHE parameters.

Another work, [CYW+24], proposes using bicyclic encoding for CC-MM between coprime-dimensional matrices. However, this algorithm has the drawback of being restricted to specific matrix shapes and generates slots filled by unusable data after CC-MM. Consequently, computationally expensive preprocessing or postprocessing steps may be required to continue further computations.

BGN-type cryptosystem [GHV10] is a pre-FHE scheme that supports only a single matrix-matrix multiplication. Its multiplication procedure is similar to ours, except that we leverage Toeplitz matrices as we rely on the RLWE problem. For iterative multiplications, GHV-type multiplication faces a fundamental challenge, as its ciphertext structure changes after each multiplication. We address this issue using our new C-MT algorithm. With a well-designed use of C-MT, we propose CC-MM algorithms with a consistent input and output format. Our CC-MM algorithm can be used iteratively without being restricted to quadratic functions, unlike [GHV10].

2 Preliminary

Vectors are denoted in bold and lower-case letters, and matrices are indicated with bold and upper-case letters. Vectors are column vectors unless explicitly stated. $[n]$ denotes the set $\{0, 1, \cdots, n-1\}$ for each positive integer n. For a power-of-two integer N and $Q \geq 2$, the ring $\mathcal{R}$ is $\mathbb{Z}[X]/(X^N+1)$ and the ring $\mathcal{R}_Q$ is $\mathcal{R}/Q\mathcal{R}$. Through the paper, N refers to the ring degree of underlying ring $\mathcal{R}$ or $\mathcal{R}_Q$. We denote $a(X) \in \mathcal{R}$ as a, omitting the symbol X unless necessary.

The Toeplitz matrix $\texttt{Toep}(m)$ is

$$\texttt{Toep}(m) = \begin{bmatrix} m_0 & m_1 & \cdots & m_{N-1} \\ -m_{N-1} & m_0 & \cdots & m_{N-2} \\ \vdots & \vdots & \ddots & \vdots \\ -m_1 & -m_2 & \cdots & m_0 \end{bmatrix}$$

for each ring element $m = \sum_i m_i X^i$.

2.1 Ring operations and Toeplitz matrix

Let $c(X) = \sum_{k=0}^{N-1} c_k X^k$ be the product of two ring elements $a(X) = \sum_{i=0}^{N-1} a_i X^i$ and $b(X) = \sum_{j=0}^{N-1} b_j X^j$ in $\mathcal{R}$. Concretely,

$$c_k = \sum_{i=0}^{k} a_i b_{k-i} - \sum_{i=k+1}^{N-1} a_i b_{k-i+N} \tag{2}$$

for each $k \in [N]$, as $X^N = -1$ in $\mathcal{R}$. Using this equation, by checking each entry, we can verify that the relation $c(X) = a(X) \cdot b(X)$ in $\mathcal{R}$ is equivalent to the following matrix equation:

$$\begin{bmatrix} a_0 \; a_1 \cdots a_{N-1} \end{bmatrix} \texttt{Toep}(b) = \begin{bmatrix} c_0 \; c_1 \cdots c_{N-1} \end{bmatrix},$$

where $\texttt{Toep}(b)$ is as defined above. Moreover, by stacking the matrix equation vertically, we can verify that the relation $\{c_i(X) = a_i(X) \cdot b(X)\}_{i \in [n]}$ is equivalent to

$$\begin{bmatrix} \boldsymbol{a_0}^t \\ \boldsymbol{a_1}^t \\ \vdots \\ \boldsymbol{a_{n-1}}^t \end{bmatrix} \texttt{Toep}(b) = \begin{bmatrix} \boldsymbol{c_0}^t \\ \boldsymbol{c_1}^t \\ \vdots \\ \boldsymbol{c_{n-1}}^t \end{bmatrix}, \tag{3}$$

where the vector $\boldsymbol{a_i}$ ($\boldsymbol{c_i}$ resp.) consists of the coefficients of $a_i(X)$ ($c_i(X)$ resp.) for each $i \in [n]$. Each relation $a_i(X) \cdot b(X) = c_i(X)$ is related to each i th *row* of Eq. (3).

There is another direction to bridge the ring multiplication with the Toeplitz matrix. From using Eq. (2), we check each entry to verify that the relation $c(X) = a(X) \cdot b(X)$ in ring $\mathcal{R}$ is equivalent to

$$\texttt{Toep}(\tilde{a}) \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{N-1} \end{bmatrix} = \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{N-1} \end{bmatrix},$$

where $\tilde{a}(X) \in \mathcal{R}$ is $a(X^{2N-1}) = a_0 - \sum_{i=1}^{N-1} a_{N-i} X^i$. By stacking the above equation horizontally, we can observe that the relation $\{c_j(X) = a(X) \cdot b_j(X)\}_j$ is equivalent to

$$\texttt{Toep}(\tilde{a}) \begin{bmatrix} \boldsymbol{b_0} \; \boldsymbol{b_1} \cdots \boldsymbol{b_{n-1}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{c_0} \; \boldsymbol{c_1} \cdots \boldsymbol{c_{n-1}} \end{bmatrix}, \tag{4}$$

where the vector $\boldsymbol{b_j}$ ($\boldsymbol{c_j}$ resp.) consists of the coefficients of $b_j(X)$ ($c_j(X)$ resp.) for each $j \in [n]$. We note that each relation $a(X) \cdot b_j(X) = c_j(X)$ is related to each j th *column* of Eq. (4).

All the above discussions can be generalized to $\mathcal{R}_Q$ with the same arguments in modulo Q.

2.2 Ring automorphisms and trace

The ring $\mathcal{R}$ is the extension ring of $\mathbb{Z}$. We introduce Galois group $\mathrm{Gal}(\mathcal{R}/\mathbb{Z})$, which is a group of automorphisms of $\mathcal{R}$ induced by Galois automorphisms in $\mathbb{Q}[X]/(X^N + 1)$ that fixes $\mathbb{Q}$. In particular,

$$\sigma(r_1 + r_2) = \sigma(r_1) + \sigma(r_2), \;\; \sigma(r_1 r_2) = \sigma(r_1)\sigma(r_2), \;\; \sigma(n) = n$$

for all $\sigma \in \mathrm{Gal}(\mathcal{R}/\mathbb{Z})$, $r_1, r_2 \in \mathcal{R}$, and $n \in \mathbb{Z}$.

For the ring $\mathcal{R} = \mathbb{Z}[X]/(X^N + 1)$, it is known that

$$\mathrm{Gal}(\mathcal{R}/\mathbb{Z}) = \left\{\sigma_t : X \mapsto X^{2t+1} \mid t \in [N]\right\},$$

and is generated by two generators, $X \mapsto X^5$ and $X \mapsto X^{-1}$.

The trace of a ring element $r \in \mathcal{R}$ is

$$\mathtt{Tr}(r) = \sum_{\sigma \in \mathrm{Gal}(\mathcal{R}/\mathbb{Z})} \sigma(r).$$

When we represent the ring elements r as a polynomial $\sum_i r_i X^i$, then it is known that

$$\mathtt{Tr}(r) = N \cdot r_0.$$

Following the above discussion in modulo Q, all the above discussion can be extended to $\mathcal{R}_Q$.

2.3 Homomorphic encryption and the CKKS scheme

Among various existing HE schemes [CKKS17,BGV14,Bra12,FV12,CGGI17,DM15], we mainly focus on the CKKS scheme. CKKS [CKKS17] HE scheme supports arithmetic over real numbers. It relies on the ring-learning-with error (RLWE) problem [SSTX09,LPR10] over the ring $\mathcal{R} = \mathbb{Z}[X]/(X^N + 1)$. Our algorithm is applicable to other existing RLWE-based schemes, such as BGV [BGV14] and FV [Bra12,FV12] schemes. The RLWE-based HE schemes have structures similar to those of the CKKS scheme except for the encoding structure.

Encodings, decoding, encryption, and decryption. Conventionally, CKKS uses *slot-encodings.* The slot-encoding and slot-decodings are maps between the message space $\mathbb{C}^{N/2}$ and the plaintext space $\mathcal{R}_Q$. To decode a plaintext $m(X) \in \mathcal{R}$, we first embed $m(X)$ into $\mathbb{C}[X]/(X^N + 1)$, and the decoded message is $\{\frac{1}{\Delta} m(\zeta_j)\}_{j \in [N/2]}$, where Δ is a scale factor and $\zeta_j = e^{2i\pi 5^j/2N}$ for each $j \in [N/2]$. The encoding is the inverse of the decoding. The slot-encoding supports slot-wise operations with SIMD property. For most of the homomorphic computations, slot-encoding is desirable.

Another encoding type is *coefficient-encoding.* The coefficient-encoding and coefficient-decoding are maps between $\mathbb{R}^N$ and the plaintext space $\mathcal{R}_Q$. The coefficient-encoding of a real vector $\{m_0, \cdots, m_{N-1}\}$ with a scale factor Δ is a ring element $\left\lfloor \Delta \left(\sum_{i=0}^{N-1} m_i X^i\right) \right\rceil$. In general, the coefficient encodings are not preferable for homomorphic computation since they are incompatible with ring multiplication. Nevertheless, coefficient-encoding is used for several FHE algorithms, such as ring packing [BCK+23,MS18,CDKS21], bootstrapping [CHK+18], and PC-MM [BCH+24]. Our CC-MM algorithm also operate in the coefficient-encoding.

A CKKS ciphertext of an encoded message $m \in \mathcal{R}_Q$ encrypted by a secret key $\mathtt{sk} \in \mathcal{R}$ is a pair of ring elements $(a, b) \in \mathcal{R}_Q^2$ such that $a \cdot \mathtt{sk} + b = m + e$ in

$\mathcal{R}_Q$ where e is a small error from the $\mathcal{R}_Q$. To decrypt a CKKS ciphertext (a, b), we decode the ring element $a\mathtt{sk} + b$, where $\mathtt{sk}$ is the secret key.

The secret key $\mathtt{sk}$ is usually a sparse ternary ring element in $\mathcal{R}$. Its coefficients are from $\{-1, 0, 1\}$, and there are only $\mathtt{hwt}$ non-zero elements among N coefficients.

Homomorphic operations. The CKKS scheme supports the following homomorphic operations. The operations below are compatible with both coefficient-encoding and slot-encoding.

- $\mathtt{Add}$. For given ciphertexts $\mathtt{ct}_0, \mathtt{ct}_1 \in \mathcal{R}_Q^2$ encrypting $m_0, m_1 \in \mathcal{R}_Q$ repectively, it returns a ciphertext $\mathtt{ct}_{\mathtt{Add}}$ that encrypts $m_0 + m_1$.
- $\mathtt{Rescale}$. For a given ciphertext $\mathtt{ct} \in \mathcal{R}_{Q_1}^2$ encrypting $m \in \mathcal{R}_{Q_1}$, it returns $\mathtt{ct}' \in \mathcal{R}_{Q_0}^2$ encrypting $\lfloor Q_0 m / Q_1 \rceil$.
  By selecting $Q_1/Q_0 \approx \Delta$, the $\mathtt{Rescale}$ procedure can manage scaled errors and scale factors after $\mathtt{PtMult}$ and $\mathtt{Mult}$.
- $\mathtt{PtMult}$. For a given ciphertext $\mathtt{ct} \in \mathcal{R}_Q^2$ encrypting m and a plaintext $\mu \in \mathcal{R}_Q$, it returns a ciphertext $\mathtt{ct}_{\mathtt{PtMult}}$ that encrypts μm.
  Note that the error inside $\mathtt{ct}$ is also multiplied by μ, which can be managed by a $\mathtt{Rescale}$ procedure after. Also, the scale factor of $\mathtt{ct}_{\mathtt{PtMult}}$ is the product of the scale factors of m and μ, which also can be managed by the $\mathtt{Rescale}$ procedure.
- $\mathtt{KeySwitch}$. For a given ciphertext $\mathtt{ct} \in \mathcal{R}_Q^2$ encrypted by a secret key $\mathtt{sk}$, it returns a ciphertext $\mathtt{ct}' \in \mathcal{R}_Q^2$ encrypted by $\mathtt{sk}'$.
  The $\mathtt{KeySwitch}$ procedure requires a *switching key* from $\mathtt{sk}$ to $\mathtt{sk}'$, which belongs to $\mathcal{R}_{PQ}^2$ where P is an auxiliary modulus for key switching.
  The auxiliary modulus and gadget decomposition are used for managing the error during $\mathtt{KeySwitch}$. The rank of gadget decomposition is called $\mathtt{dnum}$, and we refer to [HK20] for the details.
- $\mathtt{Mult}$. For given ciphertexts $\mathtt{ct}_0, \mathtt{ct}_1 \in \mathcal{R}_Q^2$ encrypting $m_0, m_1 \in \mathcal{R}_Q$ repectively, it returns a ciphertext $\mathtt{ct}_{\mathtt{Mult}}$ that encrypts $m_0 m_1$.
  It requires a relinearization key, a switching key from $\mathtt{sk}^2$ to $\mathtt{sk}$. Like $\mathtt{PtMult}$, the error and scale factor becomes larger, which can be managed by the $\mathtt{Rescale}$ procedure afterward.
  In the case of slot-encodings, it provides slot-wise multiplication over complex numbers.
- $\mathtt{Auto}$. For a given ciphertext $\mathtt{ct} \in \mathcal{R}_Q^2$ encrypting $m \in \mathcal{R}_Q$ and an automorphism $\sigma \in \mathrm{Gal}(\mathcal{R}/\mathbb{Z})$, it returns a ciphertext $\mathtt{ct}_\sigma$ which encrypts $\sigma(m)$.
  When $\sigma(X) = X^{2j+1}$ is an automorphism in $\sigma \in \mathrm{Gal}(\mathcal{R}/\mathbb{Z})$, we denote $\mathtt{ct}_\sigma$, the output of homomorphic automorphism on a ciphertext $\mathtt{ct}$, as

  $$\mathtt{Auto}(\mathtt{ct}; 2j + 1).$$

  The homomorphic automorphism requires an automorphism key, a switching key from $\sigma(\mathtt{sk})$ to $\mathtt{sk}$.

For slot-encoding ciphertexts, the CKKS algorithm provides more native operations by using the homomorphic structure of the slot-encodings.

- `Rotate`. For a given ciphertext $\mathtt{ct} \in \mathcal{R}_Q^2$ encrypting the complex vector $\{m_0, \cdots, m_{N/2-1}\}$ and a integer n, it returns a ciphertext encrypting $\mathtt{ct}' \in \mathcal{R}_Q^2$ which encrypts $\{m_r, m_{r+1} \cdots, m_{N/2-1}, m_0, \cdots, m_{r-1}\}$.
  It requires an automorphism key for $X \mapsto X^{5^r}$.
- `Conj`. For a given integer n and a ciphertext $\mathtt{ct} \in \mathcal{R}_Q^2$ encrypting a complex vector $\{m_0, \cdots, m_{N/2-1}\}$, it returns a ciphertext encrypting $\mathtt{ct}' \in \mathcal{R}_Q^2$ which encrypts the conjugate vector $\{\overline{m}_0, \cdots, \overline{m}_{N/2-1}\}$.
  It requires an automorphism key for $X \mapsto X^{-1}$.

We remark that the operations related to `KeySwitch` (i.e., `Mult`, `Auto`, `Rotate`, `Conj`) require $O(N \log N)$ operations in $\mathbb{Z}_Q$.

Modulus and bootstrapping. We remark that `PtMult` and `Mult` require `Rescale` to manage the scale factor and error, and the `Rescale` procedure consumes the *modulus* of the ciphertext space. Once the modulus becomes too low, we should refresh the modulus by using *bootstrapping* procedure in order to use CKKS as a fully FHE scheme.

However, bootstrapping is much heavier than the other homomorphic operations. In practical applications, the number of bootstrapping significantly affects the timing. Consequently, it is desirable to devise homomorphic algorithms that consume less moduli (i.e., have smaller multiplicative depth).

The current CKKS bootstrapping algorithms generally follow either of two approaches: S2C-first bootstrapping and C2S-first bootstrapping. For most applications, S2C-first bootstrapping is faster. We remark that during S2C-first bootstrapping, the ciphertexts use coefficient-encoding at the lowest modulus. We note that our algorithms operate in coefficient-encoding, allowing us to use the smallest parameters during FHE computation while leveraging the fast S2C-first bootstrapping.

Most practical CKKS implementation adopts RNS (residual number system) CKKS [BEHZ16,CHK$^+$19]. For the sake of simplicity, in the RNS system, we consider *modulus level*, which indicates the remaining number of times we can rescale it. For example, the fresh ciphertext would have a modulus level of 12, and each multiplication consumes one modulus level. Once the level becomes too low, we perform bootstrapping to recover the modulus level back to 12.

3 Ciphertext Matrix Transpose

We propose a fast ciphertext matrix transposition (C-MT) algorithm. Our algorithm converts N ciphertexts that encrypt each row of a given matrix to N ciphertexts that encrypt each column. To be precise, for a given $N \times N$ matrix $\mathbf{M}$, our C-MT algorithm takes as inputs N ciphertexts $(a_i, b_i)_{i \in [N]}$ such that

$$a_i \cdot \mathtt{sk} + b_i \;\approx\; \sum_j \mathbf{M}_{i,j} X^j$$

for all $i \in [N]$, and returns N ciphertexts $(a'_j, b'_j)_{j\in[N]}$ such that

$$a'_j \cdot \mathtt{sk} + b'_j \;\approx\; \sum_i \mathbf{M}_{i,j} X^i$$

for all $j \in [N]$, where $\mathtt{sk}$ is the secret key. While our algorithm focuses on $N \times N$ matrices, it can be extended to larger matrices by transposing each $N \times N$ submatrix blocks individually.

Before delving into details, we introduce an interesting application of C-MT. Consider FHE scenarios with multiple parties, such as multi-party HE [MTPBH21] and proxy re-encryption [GSB$^+$23]. Each client party encrypts its data with multiple features in a ciphertext, sends ciphertext(s) to the computing server, and the server computes tasks over the aggregated ciphertexts. During the computation, the server often needs to convert the client-wise encrypted ciphertexts into feature-wise encrypted ciphertexts and vice versa. This problem is the same problem with C-MT, and we can directly apply the C-MT algorithm to these scenarios. Figure 1 illustrates how to use C-MT algorithms for the multi-party settings.

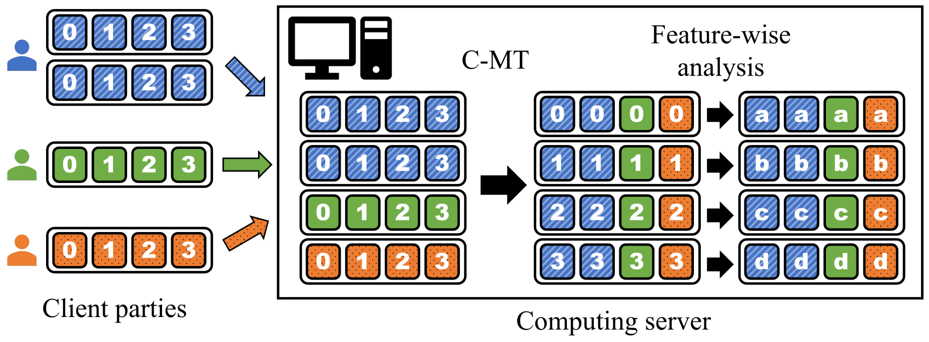

Fig. 1. Visualization of a C-MT application. Another C-MT can convert the feature-wise encryptions to client-wise encryptions before sending the results back to each client.

In this section, we propose a C-MT algorithm with $\tilde{O}(N^2)$ operations in $\mathbb{Z}_q$. We note that a transpose requires at least $\Omega(N^2)$ operations to read and write the data. Also, we point out that our algorithm does not consume any multicative level. Furthermore, we transpose the data in coefficient-encoding ciphertexts, enabling us to perform it at the lowest modulus with the fastest bootstrapping algorithms.

The proposed transpose algorithm can be generalized for other formats, such as MLWE [LS15,BGV14] and shared-a RLWE [PW08,BCH$^+$24,GKPV10,BLP$^+$13], to transpose encrypted data in those formats for matrices of various dimensions. However, while the tranpose for RLWE format is useful for CC-MM as we will describe in Section 4, transposes for other formats might not be directly useful for efficient CC-MM.

3.1 High-level description

We first describe the motivation of our algorithm with clear ring elements. Our algorithm starts from the observation that the trace of a ring element $m(X) = \sum_i m_i X^i \in \mathcal{R}_Q$ is the constant term m_0 of ring elements. Formally,

$$N \cdot m_0 = \mathtt{Tr}(m(X)) = \sum_{t=0}^{N-1} m(X^{2t+1}).$$

Similarly, for each $j \in [N]$, once we take the trace of $X^{-j} \cdot m(X)$, we obtain m_j:

$$N \cdot m_j = \mathtt{Tr}(X^{-j} \cdot m(X)) = \sum_{t=0}^{N-1} X^{-j(2t+1)} \cdot m(X^{2t+1}).$$

To utilize it for C-MT, assume that we are given N ring elements, $m_0, \cdots, m_{N-1}$ in $\mathcal{R}_Q$ such that

$$m_i(X) = \sum_j m_{i,j} X^j \in \mathcal{R}_Q.$$

In C-MT, we aim to obtain m'_j such that

$$m'_j(X) = \sum_i m_{i,j} X^i \in \mathcal{R}_Q$$

for each $j \in [N]$. The previous observation implies that

$$\begin{aligned} N \cdot m'_j(X) &= \sum_i \mathtt{Tr}(X^{-j} \cdot m_i) \cdot X^i = \sum_i \sum_t X^{-j(2t+1)+i} \cdot m_i(X^{2t+1}) \\ &= \sum_t \left(\sum_i X^i \cdot m_i(X^{2t+1}) \right) X^{-j(2t+1)} \quad \forall j \in [N]. \end{aligned}$$

Importantly, the term $\sum_i X^i \cdot m_i(X^{2t+1})$ is independent from j for all $t \in [N]$.

Based on the above discussion, in a nutshell, our proposed C-MT algorithm is as follows:

1. Compute $\{\widetilde{m}_t = \sum_i m_i(X) \cdot X^{i \cdot (2t+1)^{-1}}\}_{t \in [N]}$ from $\{m_i\}_{i \in [N]}$,
2. Compute $\{\bar{m}_t = \widetilde{m}_t(X^{2t+1})\}_{t \in [N]}$ from $\{\widetilde{m}_t\}_{t \in [N]}$, and
3. Compute $\{m'_j = \sum_t \bar{m}_t(X) \cdot X^{-j(2t+1)}\}_{j \in [N]}$ from $\{\bar{m}_t\}_{t \in [N]}$.

We can homomorphically compute all the above steps in the encrypted state by using key switching and arithemetic operations of FHE schemes. The homomorphic algorithm contains only N key switchings (step 2), which is desirable. However, it requires N^2 ring additions during the first and third steps, which uses $O(N^3)$ operations in $\mathbb{Z}_Q$.

To this end, we devise TWEAK algorithm that can compute the first and third step, using $\tilde{O}(N^2)$ operations instead of $\Omega(N^3)$. The observation is that in the above steps, all ring additions involved are structured and have the form

$$\sum_i m_i \cdot X^{2ij} \quad \forall j \in [N].$$

Our *TWEAK* algorithm (Algorithm 1) efficiently computes the structured additions with such a form.

In Section 3.2, we explain the TWEAK algorithm, and in Section 3.3, we describe our full C-MT algorithm.

3.2 TWEAK algorithm

Our TWEAK algorithm takes as an inputs n ciphertexts, $\{\mathtt{ct}_i\}_{i\in[n]}$, and returns n ciphertexts:

$$\left\{ \sum_i X^{2ij\frac{N}{n}} \cdot \mathtt{ct}_i \right\}_{j\in[n]}.$$

TWEAK algorithm computes the above n^2 ring additions over $\mathcal{R}_Q$ (which usually requires $\Omega(n^2N)$ additions over $\mathbb{Z}_Q$) with $\tilde{O}(nN)$ additions in $\mathbb{Z}_Q$. While we utilize it for C-MT in this paper, TWEAK algorithm might have many applications related to homomorphic computations with structured circuits. Algorithm 1 describes the algorithm.

Algorithm 1 TWEAK

```
Require: A power-of-two integer $n$, and $n$ ciphertexts $\mathtt{ct}$.
Ensure: Ciphertexts $\mathtt{ct}'$ such that $\mathtt{ct}'_j = \sum_i X^{2ij\frac{N}{n}}\mathtt{ct}_i$ for each $j \in [n]$
 1: if $n = 1$ then
 2:     return $\mathtt{ct}$
 3: end if
 4: $\mathtt{ct}'_0 = \mathtt{ct}_0$
 5: for $\ell \leftarrow 0$ to $\log n - 1$ do
 6:     $\mathtt{aux} \leftarrow$ TWEAK $\left(2^\ell, \{\mathtt{ct}_{(2j+1)n/2^{\ell+1}}\}_{j\in[2^\ell]}\right)$
 7:     for $j \leftarrow 0$ to $2^\ell - 1$ do
 8:         $\mathtt{ct}'_{j+2^\ell} \leftarrow \mathtt{ct}'_j - X^{\frac{N}{2^\ell}j}\mathtt{aux}_j$
 9:         $\mathtt{ct}'_j \leftarrow \mathtt{ct}'_j + X^{\frac{N}{2^\ell}j}\mathtt{aux}_j$
10:     end for
11: end for
12: return $\{\mathtt{ct}'_j\}_{j\in[n]}$
```

We prove the correctness of TWEAK algorithm in Theorem 1.

Theorem 1. *Algorithm 1 is correct.*

Proof. We prove the correctness by using induction on n. When $n = 1$, the correctness is trivial. Then, for each $n > 1$, suppose that the algorithm is correct for all integer n_0 less than n.

We claim that $\mathtt{ct}'_j = \sum_{i\in[2^{\ell+1}]} X^{2ij\frac{N}{2^{\ell+1}}} \cdot \mathtt{ct}_{i\frac{n}{2^{\ell+1}}}$ for each $j \in [2^{\ell+1}]$, after ℓ-th loop of line 6–11. Before starting the loop, the claim trivially holds with $\ell = -1$. Suppose the claim holds after $(\ell-1)$-th iteration. By the induction hypothesis,

$$\begin{aligned}
\mathtt{ct}'_j &= \sum_{i\in[2^\ell]} X^{2ij\frac{N}{2^\ell}} \cdot \mathtt{ct}_{i\frac{n}{2^\ell}} + X^{\frac{N}{2^\ell}j}\left(\sum_{i\in[2^\ell]} X^{2ij\frac{N}{2^\ell}} \cdot \mathtt{ct}_{(2i+1)n/2^{\ell+1}}\right)\\
&= \sum_{i\in[2^\ell]} X^{2(2i)j\frac{N}{2^{\ell+1}}} \cdot \mathtt{ct}_{2i\frac{n}{2^{\ell+1}}} + \sum_{i\in[2^\ell]} X^{2(2i+1)j\frac{N}{2^{\ell+1}}} \cdot \mathtt{ct}_{(2i+1)n/2^{\ell+1}}\\
&= \sum_{i\in[2^{\ell+1}]} X^{2ij\frac{N}{2^{\ell+1}}} \cdot \mathtt{ct}_{i\frac{n}{2^{\ell+1}}} ,
\end{aligned}$$

for each $j \in [2^\ell]$. Also, since $X^N = -1$ in $\mathcal{R}_Q$,

$$\begin{aligned}
\mathtt{ct}'_{j+2^\ell} &= \sum_{i\in[2^\ell]} X^{2ij\frac{N}{2^\ell}} \cdot \mathtt{ct}_{i\frac{n}{2^\ell}} - X^{\frac{N}{2^\ell}j}\left(\sum_{i\in[2^\ell]} X^{2ij\frac{N}{2^\ell}} \cdot \mathtt{ct}_{(2i+1)n/2^{\ell+1}}\right)\\
&= \sum_{i\in[2^\ell]} X^{2(2i)j\frac{N}{2^{\ell+1}}} \cdot \mathtt{ct}_{2i\frac{n}{2^{\ell+1}}} - \sum_{i\in[2^\ell]} X^{2(2i+1)j\frac{N}{2^{\ell+1}}} \cdot \mathtt{ct}_{(2i+1)n/2^{\ell+1}}\\
&= \sum_{i\in[2^{\ell+1}]} (-1)^i X^{2ij\frac{N}{2^{\ell+1}}} \cdot \mathtt{ct}_{i\frac{n}{2^{\ell+1}}} = \sum_{i\in[2^{\ell+1}]} X^{2i(j+2^\ell)\frac{N}{2^{\ell+1}}} \cdot \mathtt{ct}_{i\frac{n}{2^{\ell+1}}} .
\end{aligned}$$

To put it all together, after ℓ-th loop, $\mathtt{ct}'_j = \sum_{i\in[2^{\ell+1}]} X^{2ij\frac{N}{2^{\ell+1}}} \cdot \mathtt{ct}_{i\frac{n}{2^{\ell+1}}}$ for each $j \in [2^{\ell+1}]$, and the claim holds for all ℓ's. In particular, after the last loop iteration, ℓ becomes $\log n - 1$, and

$$\mathtt{ct}'_j = \sum_{i\in[n]} X^{2ij\frac{N}{n}} \cdot \mathtt{ct}_i$$

for each $j \in [n]$. This equation means that Algorithm 1 is correct for n. By mathematical induction, Algorithm 1 is correct for all power-of-two integers $n \geq 1$. □

We now show that the Tweak algorithm can be done with $\tilde{O}(nN)$ operations in Theorem 2.

Theorem 2. *Algorithm 1 uses* $\tilde{O}(nN)$ *operations in* $\mathbb{Z}_Q$.

Proof. Let the cost of Tweak algorithm on n ciphertexts be $T(n)$. Each ℓ-th loop of line 6–11 can be done with $T(2^\ell) + 4 \cdot 2^\ell N$ operations. Therefore,

$$T(n) = T(n/2) + T(n/4) + \cdots + T(1) + 4nN = 2T(n/2) + 2nN,$$

which implies that $T(n) = O(Nn\log n) = \tilde{O}(nN)$ □

3.3 C-MT algorithm

We propose a C-MT algorithm with $\tilde{O}(N^2)$ operations in $\mathbb{Z}_Q$, by putting Tweak algorithm into our approach in Section 3.1.

We describe our full algorithm in Algorithm 2. In a nutshell, we tweak given input ciphertexts $\mathtt{ct}$ to obtain $\mathtt{aux}$ and tweak it again to obtain the transposed ciphertexts $\mathtt{ct}'$.

Algorithm 2 Transpose

```
Require: $N$ ciphertexts $\mathtt{ct}$ such that $\mathtt{ct}_i$ encrypts $\boldsymbol{m}_i = \{m_{i,0}, \cdots, m_{i,N-1}\}$ in its coefficient for each $i \in [N]$.
Ensure: $N$ ciphertexts $\mathtt{ct}'$ such that $\mathtt{ct}'_j$ encrypts $\boldsymbol{m}'_j = \{m_{0,j}, \cdots, m_{N-1,j}\}$ for each $j \in [N]$.
 1: $\mathtt{aux} \leftarrow \textsc{Tweak}\left(N, \{X^i \cdot \mathtt{ct}_i\}_{i\in[N]}\right)$
 2: for $j \leftarrow 0$ to $N-1$ do
 3:     $\mathtt{aux}'_j \leftarrow (N^{-1} \bmod Q) \cdot \mathtt{aux}_{((2j+1)^{-1} \bmod 2N-1)/2}$
 4:     $\mathtt{aux}'_j \leftarrow \mathtt{Auto}(\mathtt{aux}'_j; 2j+1)$
 5: end for
 6: $\mathtt{ct}'' \leftarrow \textsc{Tweak}(N, \mathtt{aux}')$
 7: for $j \leftarrow 1$ to $N$ do
 8:     $\mathtt{ct}'_{j \bmod N} \leftarrow -X^{N-j} \cdot \mathtt{ct}''_{(N-j) \bmod N}$
 9: end for
10: return $\mathtt{ct}'$
```

Algorithm 2 does not consume multiplicative level during computation. We note that we multiply the constant $N^{-1} \bmod Q$ before key switchings. Our algorithm is related to the trace over $\mathcal{R}_Q$, which creates a scaling of the plaintext by a factor of N. Pre-multiplying by $N^{-1} \bmod q$ allows us to avoid this scaling. We remark that for FHE schemes (especially when using RNS HE schemes), it is a convention to choose N as a power-of-two integer and Q as a product of primes that are congruent to 1 modulo $2N$. This trick is also introduced in [CDKS21].

We now prove the correctness of Transpose algorithm in Theorem 3.

Theorem 3. *Algorithm 2 is correct.*

Proof. From the correctness of Algorithm 1, after line 1,

$$\mathtt{aux}_t = \sum_i X^{2it} \cdot (X^i \cdot \mathtt{ct}_i) = \sum_i X^{i(2t+1)} \cdot \mathtt{ct}_i$$

for each $t \in [N]$. After rearrangement and key switching (line 2–5),

$$N \cdot \mathtt{aux}'_t = \mathtt{Auto}\left(\sum_i X^{i(2t+1)^{-1}} \cdot \mathtt{ct}_i \ ; \ 2t+1\right) = \sum_i X^i \cdot \mathtt{ct}_i(X^{2t+1})$$

for each $t \in [N]$. Again by Theorem 1, after TWEAK algorithm in line 6, $\texttt{ct}''$ satisfies $i \in [N]$:

$$N \cdot \texttt{ct}''_j = N \cdot \sum_t (X^{2jt} \cdot \texttt{aux}'_t) = \sum_t \sum_i X^{2jt} \cdot X^i \cdot \texttt{ct}_i(X^{2t+1}).$$

Finally, for each $j \in [N]$,

$$\begin{aligned} N \cdot \texttt{ct}'_{(N-j) \bmod N} &= -X^j \cdot \sum_t \sum_i X^{2jt} \cdot X^i \cdot \texttt{ct}_i(X^{2t+1}) \\ &= -\sum_{t,i} X^{j(2t+1)} \cdot X^i \cdot \texttt{ct}_i(X^{2t+1}) \ = -\sum_i X^i \cdot \texttt{Tr}(X^j \cdot \texttt{ct}_i) \\ &= -\sum_i X^i \cdot \texttt{Tr}(\texttt{Enc}(\{-m_{i,N-j}, \cdots, m_{i,N-j-1}\})) \\ &= N \cdot \texttt{Enc}(\sum_i m_{i,N-j} \cdot X^i) \end{aligned}$$

Therefore, Algorithm 2 is correct. □

We finally show that our C-MT algorithm (Algorithm 2) costs $\tilde{O}(N^2)$ operations.

Theorem 4. *Algorithm 2 uses $\tilde{O}(N^2)$ operations in $\mathbb{Z}_Q$.*

Proof. Algorithm 2 consists of two Tweak and N key switching. The cost of each Tweak is $\tilde{O}(N^2)$ by Theorem 2, and the cost of each key switching is $\tilde{O}(N)$. Consequently, the overall cost of Algorithm 2 is $\tilde{O}(N^2)$.

Asymptotically, our C-MT algorithm is almost optimal. We stress that as we manipulate N^2 messages, the lower bound of the cost of C-MT is $\Omega(N^2)$. In addition, our algorithm does not consume moduli.

4 Ciphertext-Ciphertext Matrix Multiplication

In this section, we propose a new algorithm for homomorphic matrix multiplication. Our algorithm focuses on large matrices whose dimension is larger or equal to the RLWE ring dimension N. For ease of discussion, we focus on the square matrices of size $N \times N$. Extending our algorithm to larger matrices is easy.

One of the difficulties of CC-MM involves homomorphic computation with a mathematical structure. Even though there exist several works including [JKLS18], which are asymptotically optimal, the overhead from homomorphic computation, e.g., inefficient memory access, makes it hardly practical for large matrices. While PP-MM is a well-studied problem and there exist PP-MM libraries with low-level optimizations, CC-MM needs to be more scalable.

To this end, we reduce CC-MM to PP-MM. This reduction enables us to utilize highly optimized linear algebra libraries for ciphertext matrix multiplications, significantly improving the timing. This reduction-based approach is inspired by [BCH$^+$24] and also references [LZ22], both of which construct and utilize reductions from PC-MM to PP-MM.

4.1 Matrix form of RLWE ciphertexts

We introduce the matrix form from [BCH$^+$24] and use it to construct the reduction from CC-MM to PP-MM. To manipulate matrices whose dimension is equal to N, we use N RLWE ciphertexts. For example, we can encrypt each row of an $N \times N$ matrix $\mathbf{M}$ in N ciphertexts $\mathtt{ct}$ such that $\mathtt{ct}_i = (a_i, b_i)$ where $b_i = a_i \cdot \mathtt{sk} + m_i$ in $\mathcal{R}_Q$. In particular, $m_i = \lfloor \Delta(\sum_j \mathbf{M}_{i,j} X^j) + e(X) \rceil$, which is a coefficients-encoding of the i-th row of $\mathbf{M}$. We represent this as a matrix equation using the Toeplitz matrices using Eq. (3) as follows.

$$\mathbf{A} \cdot \mathtt{Toep}(\mathtt{sk}) + \mathbf{B} \approx \mathbf{M} \tag{5}$$

We remark that each row of $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{M}$ correspond to a_i, b_i and m_i for each i. We stress that we can view N RLWE ciphertexts using Eq. (3), and conversely, we can view Eq. (3) as N RLWE ciphertexts by extracting each row of $\mathbf{A}$ and $\mathbf{B}$. We refer to Section 2.1 for more details.

It is also possible to encrypt each column of M. Using Eq. (4), the following equation corresponds to the column-wise encryptions $(\underline{\mathbf{A}}, \underline{\mathbf{B}})$ of $\mathbf{M}$.

$$\mathtt{Toep}(\widetilde{\mathtt{sk}}) \cdot \underline{\mathbf{A}} + \underline{\mathbf{B}} \approx \mathbf{M}. \tag{6}$$

Each column of $\underline{\mathbf{A}}$, $\underline{\mathbf{B}}$, and $\mathbf{M}$ involves to the a, b, and m parts of each ciphertexts. Here, $\widetilde{\mathtt{sk}} = \mathtt{sk}(X^{-1})$. We refer to Section 2.1 for more details.

In this paper, we set the row-wise encryptions as a default format. For $N \times N$ matrices $\mathbf{A}$ and $\mathbf{B}$, if we say $\mathtt{ct} = (\mathbf{A}, \mathbf{B})$ encrypts an $N \times N$ matrix $\mathbf{M}$, it means that they encrypt each row of $\mathbf{M}$ and $\mathbf{A} \cdot \mathtt{Toep}(\mathtt{sk}) + \mathbf{B} = \mathbf{M}$ holds. When we perform key switching or rescale the ciphertexts $(\mathbf{A}, \mathbf{B})$, we compute over each row. Once we used column-wise encryptions, we clarified it with an underbar. For example, $\underline{\mathtt{ct}} = (\underline{\mathbf{A}}, \underline{\mathbf{B}})$ encrypts $\mathbf{M}$ implies that it encrypts each column of $\mathbf{M}$ and $\mathtt{Toep}(\widetilde{\mathtt{sk}}) \cdot \underline{\mathbf{A}} + \underline{\mathbf{B}} = \mathbf{M}$ holds.

C-MT in matrix form. We can represent Algorithm 3 with the matrix form (Eq. (5) and (6)). Algorithm 2 converts ciphertexts that encrypt each row in their coefficient to ciphertexts that encrypt each column in their coefficient. In the matrix form, the algorithm is a conversion between $(\mathbf{A}, \mathbf{B})$ and $(\underline{\mathbf{A}}, \underline{\mathbf{B}})$ such that

$$\mathbf{A} \cdot \mathtt{Toep}(\mathtt{sk}) + \mathbf{B} \;\approx\; \mathtt{Toep}(\widetilde{\mathtt{sk}}) \cdot \underline{\mathbf{A}} + \underline{\mathbf{B}}.$$

4.2 CC-MM algorithm

We propose a new CC-MM algorithm. Our algorithm takes two row-wise encryptions of matrices as inputs and outputs a row-wise encryption of the product matrix.

Assume that we are given matrices $\mathbf{M}$ and $\mathbf{M}'$ of size $N \times N$, and their row-wise encryptions $\mathtt{ct} = (\mathbf{A},\mathbf{B})$ and $\mathtt{ct}' = (\mathbf{A}', \mathbf{B}')$. We can represent it in a matrix form as follows.

$$\mathbf{M} \;\approx\; \mathbf{A} \cdot \mathtt{Toep}(\mathtt{sk}) + \mathbf{B}$$

and

$$\mathbf{M}' \approx \mathbf{A}' \cdot \mathtt{Toep}(\mathtt{sk}) + \mathbf{B}'.$$

Then, by using C-MT, we transpose the row-wise encryption ($\mathbf{A}$,$\mathbf{B}$) to obtain the column-wise encryption ($\underline{\mathbf{A}}$,$\underline{\mathbf{B}}$) of $\mathbf{M}$:

$$\mathbf{M} \approx \mathtt{Toep}(\widetilde{\mathtt{sk}}) \cdot \underline{\mathbf{A}} + \underline{\mathbf{B}}.$$

Then, we multiply $(\underline{\mathbf{A}}, \underline{\mathbf{B}})$ and $(\mathbf{A}', \mathbf{B}')$ in matrix forms:

$$\begin{aligned}\mathbf{M}\mathbf{M}' &= (\mathtt{Toep}(\widetilde{\mathtt{sk}}) \cdot \underline{\mathbf{A}} + \underline{\mathbf{B}}) \cdot (\mathbf{A}' \cdot \mathtt{Toep}(\mathtt{sk}) + \mathbf{B}') \\ &= \mathtt{Toep}(\widetilde{\mathtt{sk}}) \cdot \underline{\mathbf{A}}\mathbf{A}' \cdot \mathtt{Toep}(\mathtt{sk}) + \mathtt{Toep}(\widetilde{\mathtt{sk}}) \cdot \underline{\mathbf{A}}\mathbf{B}' + \underline{\mathbf{B}}\mathbf{A}' \cdot \mathtt{Toep}(\mathtt{sk}) + \underline{\mathbf{B}}\mathbf{B}'.\end{aligned}$$

We transpose two column-wise encryptions $(\underline{\mathbf{A}}\mathbf{A}', \mathbf{0})$ of $\left(\mathtt{Toep}(\widetilde{\mathtt{sk}}) \cdot \underline{\mathbf{A}}\mathbf{A}'\right)$ and $(\underline{\mathbf{A}}\mathbf{B}', \mathbf{0})$ of $\left(\mathtt{Toep}(\widetilde{\mathtt{sk}}) \cdot \underline{\mathbf{A}}\mathbf{B}'\right)$, respectively. It outputs two row-wise encryptions $(\hat{\mathbf{A}}, \hat{\mathbf{B}})$ of $\left(\mathtt{Toep}(\widetilde{\mathtt{sk}}) \cdot \underline{\mathbf{A}}\mathbf{A}'\right)$ and $(\check{\mathbf{A}}, \check{\mathbf{B}})$ of $\left(\mathtt{Toep}(\widetilde{\mathtt{sk}}) \cdot \underline{\mathbf{A}}\mathbf{B}'\right)$. To be more precise, in matrix form,

$$\mathtt{Toep}(\widetilde{\mathtt{sk}}) \cdot \underline{\mathbf{A}}\mathbf{A}' \approx \hat{\mathbf{A}} \cdot \mathtt{Toep}(\mathtt{sk}) + \hat{\mathbf{B}},$$

and

$$\mathtt{Toep}(\widetilde{\mathtt{sk}}) \cdot \underline{\mathbf{A}}\mathbf{B}' \approx \check{\mathbf{A}} \cdot \mathtt{Toep}(\mathtt{sk}) + \check{\mathbf{B}}.$$

Putting it all together results in:

$$\begin{aligned}\mathbf{M}\mathbf{M}' &\approx \hat{\mathbf{A}} \cdot \mathtt{Toep}(\mathtt{sk}) \cdot \mathtt{Toep}(\mathtt{sk}) + (\hat{\mathbf{B}} + \check{\mathbf{A}} + \underline{\mathbf{B}}\mathbf{A}') \cdot \mathtt{Toep}(\mathtt{sk}) + (\check{\mathbf{B}} + \underline{\mathbf{B}}\mathbf{B}') \\ &= \hat{\mathbf{A}} \cdot \mathtt{Toep}(\mathtt{sk}^2) + (\hat{\mathbf{B}} + \check{\mathbf{A}} + \underline{\mathbf{B}}\mathbf{A}') \cdot \mathtt{Toep}(\mathtt{sk}) + (\check{\mathbf{B}} + \underline{\mathbf{B}}\mathbf{B}').\end{aligned}$$

As the last step, we relinearize and rescale each row of $(\hat{\mathbf{A}},\ \hat{\mathbf{B}}+\check{\mathbf{A}}+\underline{\mathbf{B}}\mathbf{A}',\ \mathbf{B} + \check{\underline{\mathbf{B}}}\mathbf{B}')$. This step outputs the row-wise encryptions of $\mathbf{M}\mathbf{M}'$. This completes the CC-MM procedure.

To sum up, from the row-wise encryptions of $\mathbf{M}$ and $\mathbf{M}'$, we obtained the row-wise encryptions of the product matrix $\mathbf{M}\mathbf{M}'$. Algorithm 3 describes our algorithm in full detail.

The correctness of Algorithm 3 is directly derived from the above discussion and Theorem 3. While Algorithm 3 focuses on $N \times N$ matrices, we can use it for larger matrices by using a block approach [JKLS18]: splitting the large matrices into submatrices of size $N \times N$ and multiplying two matrices of submatrices. To reduce costs, we can use the Strassen algorithm [Str69] for the block approach.

Cost analysis. Our algorithm consists of three transpositions (line 1, 3, and 4), N key switching (line 5), N rescaling (line 6), and a PP-MM (line 2). The PP-MM part is the heaviest, while we can use highly optimized linear algebra libraries to implement it. The components other than PP-MM can be done with $\widetilde{O}(N^2)$ operations.

Algorithm 3 CC-MM

```
Require: $N$ ciphertexts $\mathtt{ct}^U = (\mathbf{A}^U, \mathbf{B}^U)$ encrypting an $N \times N$ matrix $\mathbf{U}$;
    $\mathbf{A}^U \mathtt{Toep}(\mathtt{sk}) + \mathbf{B}^U = \mathbf{U}$.
Require: $N$ ciphertexts $\mathtt{ct}^V = (\mathbf{A}^V, \mathbf{B}^V)$ encrypting an $N \times N$ matrix $\mathbf{V}$;
    $\mathbf{A}^V \mathtt{Toep}(\mathtt{sk}) + \mathbf{B}^V = \mathbf{V}$.
Ensure: $N$ ciphertexts $\mathtt{ct}^W = (\mathbf{A}^W, \mathbf{B}^W)$ encrypting an $N \times N$ matrix $\mathbf{W}$;
    $\mathbf{A}^W \mathtt{Toep}(\mathtt{sk}) + \mathbf{B}^W = \mathbf{W}$, where $\mathbf{W}{=}\mathbf{UV}$ is the $N \times N$ product matrix.
1: $(\underline{\mathbf{A}}^U, \underline{\mathbf{B}}^U) \leftarrow \mathrm{Transpose}(\mathtt{ct}^U)$
2: $\begin{bmatrix} \underline{\mathbf{M}}_{00} & \underline{\mathbf{M}}_{01} \\ \mathbf{M}_{10} & \mathbf{M}_{11} \end{bmatrix} \leftarrow \begin{bmatrix} \underline{\mathbf{A}}^U \\ \underline{\mathbf{B}}^U \end{bmatrix} \begin{bmatrix} \mathbf{A}^V & \mathbf{B}^V \end{bmatrix}$
3: $(\check{\mathbf{A}}, \check{\mathbf{B}}) \leftarrow \mathrm{Transpose}((\underline{\mathbf{M}}_{01}, \mathbf{0}))$
4: $(\hat{\mathbf{A}}, \hat{\mathbf{B}}) \leftarrow \mathrm{Transpose}((\underline{\mathbf{M}}_{00}, \mathbf{0}))$
5: $(\hat{\mathbf{A}}, \hat{\mathbf{B}}) \leftarrow \mathrm{KS}_{s^2 \to s}((\hat{\mathbf{A}}, \mathbf{0})) + (\hat{\mathbf{B}}, \mathbf{0})$
6: $(\mathbf{A}^W, \mathbf{B}^W) \leftarrow \mathtt{Rescale}\left((\hat{\mathbf{A}}, \hat{\mathbf{B}}) + (\check{\mathbf{A}}, \check{\mathbf{B}}) + (\mathbf{M}_{10}, \mathbf{M}_{11})\right)$
7: return $(\mathbf{A}^W, \mathbf{B}^W)$
```

The large PP-MM uses $O(N^\omega)$ operations, where ω is a constant. In most practical implementations, ω is 3. However, we can use highly optimized open libraries for PP-MM, and they significantly reduce the practical timing. For example, in our implementation on CPU with $N = 2^{12}$, an $N \times N$ square matrix multiplication with OpenBLAS [oBlv] library takes 1.47 seconds, which was faster than the schoolbook implementation by a factor more than 30. It partly implies that our algorithm would be practically faster than any other CC-MM algorithms without reduction to PP-MM. The importance of reduction to PP-MM has also been stressed in [BCH+24], but they focused on the PC-MM rather than CC-MM.

Any improvement in the implementation of PP-MM will directly benefit our algorithm. While this paper implemented our algorithm on a CPU, it can be easily adapted to any hardware architecture unless it does not support efficient matrix operations.

Overall, the cost of our CC-MM algorithm is four PP-MMs (which is $O(N^3)$ with BLAS libraries) with $\tilde{O}(N^2)$ operations for others. For CC-MM between $d \times d$ matrices where $d > N$, we use block approach, and the cost would be $\tilde{O}\left((d/N)^{\omega_1} N^{\omega_0}\right)$. Here, ω_0 is the constant for PP-MM, in which typically $\omega_0 = 3$ if we use BLAS libraries, and ω_1 is a constant for block approach, in which $\omega_1 = \log_2 7$ if we use Strassen algorithm [Str69].

Moduli consumption and fusing CC-MM with bootstrapping. If we use the fastest matrix transposition algorithm (Algorithm 2), Algorithm 3 requires $O(N)$ switching keys. However, each key is small, as we can use the switching keys with almost the smallest parameters. Our CC-MM algorithm consumes only one level during multiplication, and the switching keys are sufficient to support key switching at level 1.

Our matrix multiplication algorithms are done in coefficient-encodings. The ciphertexts are coefficient-encoded at the lowest moduli when using FHE with the current fastest bootstrapping methods ($\mathsf{S2C}$-first bootstrapping). Consequently, we can perform our CC-MM algorithm in the lowest modulus while utilizing the fast $\mathsf{S2C}$-first bootstrapping algorithms, as in the case of the state-of-the-art PC-MM in [BCH+24].

4.3 Encoding structure: row or column

Algorithm 3 takes two row-wise encryptions of matrices as inputs and outputs a row-wise encryption of the product. However, it is easy to generalize it to column-wise encrypted inputs and outputs (or even to column-wise inputs and row-wise outputs, and vice versa). In general, the consistent structure of input and output encryptions (e.g., both row-wise encryption or both column-wise encryption) is advantageous for easier implementation in general applications. We note that our algorithms are flexible in choosing the encoding structure of inputs and outputs as we have an efficient C-MT algorithm.

One interesting scenario is that when we have column-wise encryption of $\mathbf{U}$ and row-wise encryption of $\mathbf{V}$. Then, we can complete CC-MM with two C-MTs rather than three, skipping the first C-MT (line 1 in Algorithm 3). More importantly, this gives flexibility on the shapes of $\mathbf{U}$ and $\mathbf{V}$; Algorithm 3 with omitting line 1 works well with $N \times n$ matrix $\mathbf{U}$ and $n \times N$ matrix $\mathbf{V}$ for $1 \leq n \leq N$.

5 Lightweight Algorithms with Small Key Size

One of the possible concerns about Algorithms 2 and 3 is the key size. Without optimization, Algorithms 2 and 3 require N switching keys for all automorphisms.

Many switching keys might be acceptable for large devices, as all keys can have small parameters. Since Algorithm 2 does not consume any modulus, we can assume that all input ciphertexts are in the low modulus (e.g., less than 64-bits primes), with the low ring degree (e.g., $N = 2^{12}$). For example, we can transpose at the bottom modulus and recover the moduli using HalfBTS as in PC-MM [BCH+24]. Thus, each key for C-MT and CC-MM is smaller than other keys for homomorphic computation, and N switching keys might be affordable. However, even with the small parameters, N switching keys might not be affordable for small devices.

In this section, we introduce "lightweight" algorithms with small key sizes for C-MT and CC-MM. With a slight modification, they are based on Algorithms 2 and 3.

5.1 Lightweight C-MT algorithm

In Algorithm 2, we use N switching keys for N homomorphic automorphisms, applied sequentially (line 2–5 in Algorithm 2). To reduce the key size, our lightweight algorithm uses a single switching key, which is updated before each use. This is possible because in RLWE-based FHE schemes, a switching key is a tuple of ciphertexts, and we can update it by using another switching key. This observation was also introduced in [LLKN23] to reduce the communication cost during key setup. Building on this idea, we further reduce both the communication cost and the on-chip memory requirements for switching keys in C-MT and CC-MM algorithms.

For ease of discussion, assume the gadget rank of switching keys is 1. The fundamental observation is that the switching key is an encryption of $P \cdot \mathtt{sk}(X^{2t+1})$ for each $t \in [N]$, where P is a constant auxiliary modulus for key switchings. Following the conventional encoding structure of CKKS, the set of all switching keys is

$$\{\mathtt{Enc}(P \cdot \mathtt{sk}(X^{5^i}))\}_{i\in[N/2]} \cup \{\mathtt{Enc}(P \cdot \mathtt{sk}(X^{-5^i}))\}_{i\in[N/2]}.$$

Consequently, we can generate another switching key from a given switching key by performing automorphism to the given switching key. If we have two master rotation keys ($\mathtt{Enc}(PP' \cdot \mathtt{sk}(X^5))$ and $\mathtt{Enc}(PP' \cdot \mathtt{sk}(X^{-1}))$ and an initial switching key $\mathtt{Enc}(P \cdot \mathtt{sk}(X))$, we can generate all switching keys. Here, P' is an auxiliary modulus for generating the key.

In Algorithm 2, we sequentially use the switching keys in the loop 2–5. Using the above technique, we can use a single switching key by repeatedly updating it. The switching key is initially set to $\mathtt{Enc}(P \cdot \mathtt{sk}(X))$. For the first $N/2$ iterations, we use the key and update it using the master rotation key $\mathtt{Enc}(PP' \cdot \mathtt{sk}(X^5))$. Precisely, for each i th loop, the switching key will be updated to $\mathtt{Enc}(P \cdot \mathtt{sk}(X^{5^i}))$. After $N/2$ iterations, we update the switching key using the master conjugation key $\mathtt{Enc}(PP' \cdot \mathtt{sk}(X^{-1}))$ and continue the last $N/2$ iterations. In particular, for each $(i + N/2)$ th loop, the switching key will be updated to $\mathtt{Enc}(P \cdot \mathtt{sk}(X^{-5^i}))$.

With the above strategy, we can exploit only three switching keys. This significantly reduces the key size. The computational cost would increase since N additional key switchings have been introduced. However, we stress that the asymptotic complexity is still $\tilde{O}(N^2)$. In the case with a gadget rank $\mathtt{dnum}$ larger than 1, we have the same result by repeating the above procedures for $\mathtt{dnum}$ times.

5.2 Lightweight CC-MM algorithm

The C-MT algorithm with a small key size directly implies a CC-MM algorithm with a small key size. We note that in Algorithm 3, we used C-MT in a black-box manner, and we can adopt lightweight C-MT to achieve a small key size.

With this modification, our lightweight CC-MM algorithm requires four switching keys: one for relinearization, one for automorphisms, and two for updating the automorphism key. This modification might increase the cost of C-MT by a small constant factor, but it is still minor compared to the cost of PP-MM (line 2 in Algorithm 3).

6 Experiments

We implement our algorithms with HEaaN CKKS library [HEa22]. For the matrix multiplication, we use FLINT library [tea24] to implement modular matrix multiplications. All experiments are measured on an Intel Xeon Gold 6242 CPU running at 2.80GHz, using a single thread. The HEaaN library takes advantage of the AVX512 instruments.

For the accuracy, we measured the relative error bit. Precisely, if the ideal output matrix is W and the experimental result is W', we measured

$$-\log_2 \left(\max_i \|W_i - W_i'\|_\infty \;/\; \max_i \|W_i\|_\infty \right),$$

where each M_i indicates the i-th row of matrix M.

For the key size, we calculated it as:

$$N \cdot \texttt{dnum} \cdot \log_2 PQ,$$

where N is the RLWE ring degree, PQ is the modulus that the key switching occurs, and dnum is the rank of the gadget decomposition of the key.

Here, we assumed the a parts of the keys are from an output of an extendable output-format function (XOF) on public seeds. Then, we may send or store the keys without the a part. Note that we exclude the cost of sending seed, as it is relatively minor, and we can use a counter to use one seed for multiple keys. This technique has been introduced in the literature, such as [CDKS21,BCK+23].

6.1 C-MT algorithm

We present the experimental results of C-MT algorithms with our proof-of-concept implementations. We transpose N ciphertexts that each encrypt a random vector from $[-1, 1]^N$ in its coefficients.

Parameter	$\log_2 N$	$\log_2 Q$	$\log \Delta$	$\log_2(QP)$	dnum	hwt
Fst11	11	26	24	52	1	256
Lt12	12	28	27	$(64, 104)$	$(1, 2)$	256

Table 1. Parameters for C-MT experiments.

Parameter selection We conducted experiments on two parameter sets. The parameters are presented in Table 1. All parameters are $\approx$ 128-bit secure based on [APS15].

In Table 1, N is the ring degree of RLWE ciphertexts, and QP is the modulus at which key switchings occur. Q is the ciphertext modulus, and dnum is the rank of the gadget decomposition of key switchings. Δ is the scaling factor, and hwt is the hamming weight of secret keys.

For QP and dnum of Lt12 parameter, the first entries are used for homomorphic automorphisms, and the second entries are used for updating the key.

Experimental results. We report the timing and the accuracy of the C-MT algorithms in Table 2. The first two rows present the results of Algorithm 2, and the third row shows the results of the lightweight C-MT algorithm (Section 5.1). We take the average of latency and accuracy among 100 experiments.

Parameter	d	Latency (s)	Accuracy (bit)	Key size (MB)
Fst11	2 048	0.764	10.7	27.3
Lt12	4 096	3.04	16.3	134
Lt12*	4 096	4.92	14.2	0.246

Table 2. Experimental results of C-MT algorithms on $d \times d$ matrices.

The result on the parameter Fst11 takes less than a second to transpose a large matrix of size 2 048 × 2 048. As it uses relatively small FHE parameters, the key size is reasonable (less than 30 MB) without the lightweight algorithm. The first two rows experimentally show that the C-MT algorithm has $\tilde{O}(N^2)$ complexity, as Lt12 is slower than Fst11 by a factor 4. The last two rows show the impact of the lightweight algorithm. The lightweight algorithm mildly degrades performance while allowing a more than 540 times smaller key size, from 134 MB to 0.25 MB.

6.2 CC-MM algorithm

We describe the experimental results of the ciphertext matrix multiplication algorithm with our proof-of-concept implementations. We multiplied two vectors of ciphertexts of length N that each vector encrypts the rows of a random $N \times N$ matrix in $[-1, 1]^{N \times N}$.

Parameter selection. We conducted experiments on two parameter sets. The parameters are presented in Table 3. All parameters are ≈ 128-bit secure based on [APS15].

Parameter	$\log_2 N$	$\log_2 Q$	$\log \Delta$	$\log_2(QP)$	`dnum`	`hwt`
Fst12	12	$(64, 36)$	28	104	2	256
Lt13	13	$(66, 38)$	28	$(117, 178)$	$(2, 3)$	256

Table 3. Parameters for CC-MM experiments.

In Table 3, as in Table 1, N is the ring degree of RLWE ciphertexts, and `dnum` is the rank of the gadget decomposition of key switchings. Δ is the scaling factor, and `hwt` is the hamming weight of secret keys. Q is the ciphertext modulus. Note that the CC-MM algorithm contains rescale procedures, and the input and output ciphertext moduli are different. In Table 3, the first entry of $\log Q$ indicates the modulus of input ciphertexts, and the second entry is that of output ciphertexts. QP is the modulus at which key switchings occur. For QP and `dnum` of Lt13 parameter, the first and second entries are used for homomorphic automorphisms and updating the key, respectively.

Experimental results. We report the measured timing and the accuracy in Table 4 and 5 with the key size. In Table 4, we present the total latency, accuracy, and the key size of the CC-MM algorithm. The first two rows present the results of Algorithm 3, and the third reports the results of the lightweight CC-MM algorithm (Section 5.2). We take the average of latency and accuracy among 10 experiments.

Parameter	d	Latency (s)	Accuracy (bit)	Key size (MB)
Fst12	4 096	85.2	18.7	436
Lt13	8 192	596	18.5	1 960
Lt13*	8 192	672	18.5	1.57

Table 4. Experimental results of CC-MM algorithms for multiplying two encrypted $d \times d$ matrices.

The result for Fst12 parameter shows that multiplying two encrypted 4 096× 4 096 matrices takes less than 1.5 minutes. The results for Lt13 parameter demonstrate the impact of the lightweight algorithm. The lightweight algorithm reduces the key size by a factor of over 1 200, from 1.96 GB to 1.57 MB. Notably, the performance degradation in the lightweight CC-MM algorithm is less significant than in C-MT, as the cost of C-MT during CC-MM is relatively minor compared to that of PP-MM.

In Table 5, we report the timing of each step in the CC-MM algorithm. In the first column (Transpose), we present the latency of three transposes (line 1, 3, and 4 of Algorithm 3), and in the second column, we present the latency of PP-MM with FLINT library (line 2 of Algorithm 3). In the third and fourth columns (Relin and Rescale, respectively), we report the latency of relinearization (line 5) and rescale (line 6), respectively.

Parameter	**Latency of each step (s)**				**Total (s)**
	Transpose	PP-MM	Relin	Rescale	
Fst12	25.5	57.1	1.36	1.22	85.2
Lt13	104	481	6.11	5.71	596
Lt13*	186	474	6.28	5.51	672

Table 5. Timing of each step of CC-MM algorithm.

We stress that the PP-MM step was the heaviest step of our ciphertext matrix multiplication. Improvements in the clear matrix multiplication implementation, e.g., improvements in the linear algebra libraries or adopting GPU or a better hardware architecture for modular matrix multiplication, will directly benefit our algorithm.

In Table 6, we provide the timing of clear floating-point matrix multiplication, PC-MM, and CC-MM. The floating-point matrix multiplication uses OpenBLAS library [oBlv], and the numbers for PC-MM are borrowed from [BCH+24].

Matix dimension	PP-MM ([oBlv])	PC-MM ([BCH+24])	CC-MM (Ours)
4 096	1.47	17.1	85.2
8 192	11.4	64.6	596

Table 6. Timing of matrix multiplications in a single thread. All timings are given in seconds.

We note that our reduction from CC-MM is to modular PP-MM rather than floating-point PP-MM. There is a notable gap between floating-point PP-MM and modular PP-MM timings. For example, for matrix dimension 4 096, modular PP-MM takes around 14.3 seconds on our machine, while floating-point PP-MM takes 1.47 seconds.

Table 6 demonstrates the efficiency of our CC-MM algorithm, which processes large matrices in minutes. In contrast, based on estimates, existing works would take tens of hours to process matrices of this size. The performance of our approach significantly reduces the gap between CC-MM and PC-MM or PP-MM, demonstrating its practicality in real-world applications.

Acknowledgements. We thank Dr. Alain Passelègue and Dr. Damien Stehlé for their helpful comments. We also thank Professor Jung Hee Cheon and Dr. Guillaume Hanrot for their support.

References

[ABB+99] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[APS15] M. R. Albrecht, R. Player, and S. Scott. On the concrete hardness of learning with errors. *J. Math. Cryptol.*, 2015. Software available at https://github.com/malb/lattice-estimator, git commit# 5350825.

[BCH+24] Y. Bae, J. H. Cheon, G. Hanrot, J. H. Park, and D. Stehlé. Plaintext-ciphertext matrix multiplication and FHE bootstrapping: Fast and fused. In *CRYPTO*, 2024.

[BCK+23] Y. Bae, J. H. Cheon, J. Kim, J. H. Park, and D. Stehlé. HERMES: efficient ring packing using MLWE ciphertexts and application to transciphering. In *CRYPTO*, 2023.

[BEHZ16] Jean-Claude Bajard, Julien Eynard, M Anwar Hasan, and Vincent Zucca. A full rns variant of fv like somewhat homomorphic encryption schemes. In *SAC*, pages 423–442. Springer, 2016.

[BGV14] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. *ACM Trans. Comput. Theory*, 2014.

[BLP+13] Z. Brakerski, A. Langlois, C. Peikert, O. Regev, and D. Stehlé. Classical hardness of learning with errors. In *STOC*, 2013.

[Bra12] Z. Brakerski. Fully homomorphic encryption without modulus switching from classical gapSVP. In *CRYPTO*, 2012.

[CDKS21] H. Chen, W. Dai, M. Kim, and Y. Song. Homomorphic conversion between (ring) LWE ciphertexts. In *ACNS*, 2021.

[CGGI17] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. Faster packed homomorphic operations and efficient circuit bootstrapping for TFHE. In *ASIACRYPT*, 2017.

[CHK+18] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. Bootstrapping for approximate homomorphic encryption. In *EUROCRYPT*, pages 360–384. Springer, 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*, pages 347–368. Springer, 2019.

[CKKS17] J. H. Cheon, A. Kim, M. Kim, and Y. Song. Homomorphic encryption for arithmetic of approximate numbers. In *ASIACRYPT*, 2017.

[CYW+24] J. Chen, L. Yang, W. Wu, Y. Liu, and Y. Feng. Homomorphic matrix operations under bicyclic encoding, 2024. Available at https://chenjingwei.github.io/download/hemm.pdf.

[DCLT18] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding, 2018. Available at https://arxiv.org/abs/1810.04805.

[DM15] Léo Ducas and Daniele Micciancio. Fhew: bootstrapping homomorphic encryption in less than a second. In *EUROCRYPT*, pages 617–640. Springer, 2015.

[FV12] J. Fan and F. Vercauteren. Somewhat practical fully homomorphic encryption. *Available at http://eprint.iacr.org/2012/144*, 2012.

[GHV10] Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. A simple bgn-type cryptosystem from lwe. In *EUROCRYPT*, 2010.

[GKPV10] S. Goldwasser, Y. T. Kalai, C. Peikert, and V. Vaikuntanathan. Robustness of the learning with errors assumption. In *ICS*, 2010.

[GQH+24] Y. Gao, G. Quan, S. Homsi, W. Wen, and L. Wang. Secure and efficient general matrix multiplication on cloud using homomorphic encryption. *The Journal of Supercomputing*, 2024.

[GSB+23] Mirko Günther, Lars Schütze, Kilian Becher, Thorsten Strufe, and Jeronimo Castrillon. Helium: A language and compiler for fully homomorphic encryption with support for proxy re-encryption. *arXiv preprint arXiv:2312.14250*, 2023.

[HCJG24] S. Hong, Y. A. Choi, D. S. Joo, and G. Gürsoy. Privacy-preserving model evaluation for logistic and linear regression using homomorphically encrypted genotype data. *Journal of biomedical informatics*, 2024.

[HEa22] CryptoLab. HEaaN library, 2022. Available at https://www.cryptolab.co.kr/en/products-en/heaan-he/.

[HHW$^+$21] Z. Huang, C. Hong, C. Weng, W. Lu, and H. Qu. More efficient secure matrix multiplication for unbalanced recommender systems. *IEEE Transactions on Dependable and Secure Computing*, 2021.

[HK20] Kyoohyung Han and Dohyeong Ki. Better bootstrapping for approximate homomorphic encryption. In *Cryptographers' Track at the RSA Conference*, 2020.

[HLC$^+$22] M. Hao, H. Li, H. Chen, P. Xing, G. Xu, and T. Zhang. Iron: Private inference on transformers. *Advances in Neural Information Processing Systems*, 2022.

[JKLS18] X. Jiang, M. Kim, K. Lauter, and Y. Song. Secure outsourced matrix computation and application to neural networks. In *CCS*, 2018.

[LLKN23] J. W. Lee, E. Lee, Y. S. Kim, and J. S. No. Rotation key reduction for client-server systems of deep neural network on fully homomorphic encryption. In *ASIACRYPT*, 2023.

[LPR10] V. Lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and learning with errors over rings. In *EUROCRYPT*, 2010.

[LS15] A. Langlois and D. Stehlé. Worst-case to average-case reductions for module lattices. *Des. Codes Cryptogr.*, 2015.

[LZ22] J. Liu and L. F. Zhang. Privacy-preserving and publicly verifiable matrix multiplication. *IEEE Transactions on Services Computing*, 2022.

[MMJG24] X. Ma, C. Ma, Y. Jiang, and C. Ge. Improved privacy-preserving pca using optimized homomorphic matrix multiplication. *Computers & Security*, 2024.

[MS18] D. Micciancio and J. Sorrell. Ring packing and amortized FHEW bootstrapping. In *ICALP*, 2018.

[MTPBH21] Christian Mouchet, Juan Troncoso-Pastoriza, Jean-Philippe Bossuat, and Jean-Pierre Hubaux. Multiparty homomorphic encryption from ring-learning-with-errors. *Proceedings on Privacy Enhancing Technologies*, 2021(4):291–311, 2021.

[oBlv] OpenBLAS: An optimized BLAS library version 0.3.26. Available at https://www.openblas.net/.

[PW08] C. Peikert and B. Waters. Lossy trapdoor functions and their applications. In *STOC*, 2008.

[PZM$^+$24] Q. Pang, J. Zhu, H. Möllering, W. Zheng, and T. Schneider. Bolt: Privacy-preserving, accurate and efficient inference for transformers. In *2024 IEEE Symposium on Security and Privacy (SP)*, 2024.

[RNSS18] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever. Improving language understanding by generative pre-training, 2018. Available at https://openai.com/research/language-unsupervised.

[SSTX09] D. Stehlé, R. Steinfeld, K. Tanaka, and K. Xagawa. Efficient public key encryption based on ideal lattices. In *ASIACRYPT*, 2009.

[Str69] Volker Strassen. Gaussian elimination is not optimal. *Numerische mathematik*, 13(4):354–356, 1969.

[tea24] The FLINT team. *FLINT: Fast Library for Number Theory*, 2024. Version 3.2.0-dev, https://flintlib.org – commit # c29a090.

[TLI$^+$23] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample. LLaMA: Open and efficient foundation language models, 2023. Available at https://arxiv.org/abs/2302.13971.

[ZL23] W. Zhu and X. Li. Secure mutual learning with low interactions for deep model training. In *International Conference on Mobility, Sensing and Networking*, 2023.

[ZLW23] X. Zheng, H. Li, and D. Wang. A new framework for fast homomorphic matrix multiplication, 2023. Available at https://eprint.iacr.org/2023/1649.

[ZLY$^+$23] J. Zhang, J. Liu, X. Yang, Y. Wang, K. Chen, X. Hou, K. Ren, and X. Yang. Secure transformer inference made non-interactive, 2023. Available at https://eprint.iacr.org/2024/136.