Tricycle: Private Transformer Inference with Tricyclic Encodings

Lawrence Lim
lawrenceklim@ucsb.edu
UC Santa Barbara
Santa Barbara, California, USA

Vikas Kalagi
vikaskalagi@ucsb.edu
UC Santa Barbara
Santa Barbara, California, USA

Julia Novick
jnovick@ucsb.edu
UC Santa Barbara
Santa Barbara, California, USA

Jiaming Liu
jiamingliu@ucsb.edu
UC Santa Barbara
Santa Barbara, California, USA

Divyakant Agrawal
divyagrawal@ucsb.edu
UC Santa Barbara
Santa Barbara, California, USA

Amr El Abbadi
elabbadi@ucsb.edu
UC Santa Barbara
Santa Barbara, California, USA

Abstract

The growing deployment of large language models (LLMs) in privacy-sensitive settings demands inference mechanisms that preserve data confidentiality. Homomorphic encryption (HE) offers a principled solution by enabling computation directly on encrypted data; however, existing solutions struggle to scale to full LLMs. We present Tricycle, a system for efficient private transformer inference. At its core, Tricycle introduces tricyclic encodings, a novel packing scheme that enables batch matrix multiplications with optimal multiplicative depth while naturally supporting multi-head attention. We demonstrate that Tricycle's matrix multiplications are particularly amenable for integration with other optimizations including Baby-Step Giant-Step optimizations, optimized block matrix multiplications, lazy relinearization, and complexification, which significantly improve performance. We further introduce statistical max estimation, a lightweight method for stabilizing softmax under HE. We implement Tricycle end-to-end on a GPU-accelerated CKKS pipeline and evaluate it on BERT models. For BERT-Base with 128 tokens, Tricycle achieves 100.5 seconds latency on a single GPU, yielding 6× and 3.4× speedups over prior state-of-the-art systems, Thor and Powerformer, respectively. These results demonstrate that careful design of packing, algorithms, and optimizations can significantly reduce the cost of private LLM inference, bringing practical deployment closer to reality.

CCS Concepts

• **Computing methodologies** → **Natural language processing**;
• **Security and privacy** → **Privacy-preserving protocols**.

Keywords

Private LLM Inference, Homomorphic Encryption

1 Introduction

Through the invention of transformer models, Large Language Models (LLMs) have unlocked a wide range of innovative applications thanks to their powerful text processing capabilities. However, a significant barrier to broader adoption is the need to query these models with sensitive or private data. For example, healthcare professionals may wish to use an LLM to summarize a patient's clinical history, while educators might want help grading student exams more efficiently. Both involve query data that require confidentiality. Even everyday users may seek the benefits of LLMs without compromising their privacy. Despite the potential benefits of private LLM inference in these scenarios, model owners who train LLMs using vast computational and data resources are often reluctant to share their model weights with users, as doing so could erode their competitive advantage.

One solution is to use homomorphic encryption (HE) to enable private inference on LLMs. HE allows computations to be performed directly on encrypted data, preserving privacy throughout the entire process. This approach allows a user to encrypt their query and send it to the model owner without revealing any sensitive information. Then, the model owner homomorphically evaluates the LLM, composed of matrix multiplications and nonlinear operations, on the encrypted input and returns an encrypted result, which only the user can decrypt to obtain the output.

Private LLM inference with HE has attracted significant recent attention [9, 12, 13, 25, 27, 29, 30, 34, 36, 37]. A key obstacle to scaling these approaches to large inputs and full-scale models is the high computational cost of HE operations. Consequently, improving efficiency, particularly by reducing expensive homomorphic operations, is critical for practical deployment. Prior work has explored approximating key transformer components, including softmax [29], layer normalization [30], or both [27], as well as designing more efficient matrix multiplication techniques [25, 27]. Despite these advances, existing systems still exhibit substantial end-to-end latency (e.g., Thor: 602 seconds [25]; Powerformer: 344 seconds [27], both for BERT-Base [11] with 128 tokens), limiting their practicality.

In this paper, we introduce Tricycle, a system for private LLM inference based on a novel packing scheme called tricyclic encodings. These encodings are designed to integrate naturally into end-to-end private transformer pipelines and enable batch matrix multiplications with optimal multiplicative depth. We demonstrate that Tricycle's matrix multiplications are particularly amenable for integration with other optimizations. These optimizations include generic optimizations (e.g. Baby-Step Giant-Step, lazy relinearization, and complexification) and Tricycle-specific optimizations (e.g. block matrix multiplications). Collectively, these optimizations are a significant factor in Tricycle's improved performance over prior work based on bicyclic encodings [4]. We also propose a lightweight method for stabilizing softmax under homomorphic encryption, called statistical max estimation. Together, these design choices in addition to various systems optimizations enable an efficient end-to-end Tricycle system. On a single GPU, Tricycle performs private inference for a BERT-Base model over 128 tokens in 100.5

seconds, achieving 6× and 3.4× speedups over Thor [25] and Powerformer [27], respectively, and establishing new state-of-the-art performance.

2 Homomorphic Encryption

Homomorphic encryption enables computation on encrypted data, allowing data to remain private while still being processed. We use CKKS [7], an approximate homomorphic encryption scheme that operates on floating-point data. In CKKS, each ciphertext encodes a vector of complex numbers (and hence also real numbers), with the number of available slots given by $s = N/2$ where N is the ring dimension. CKKS supports several operations:

- KeyGen(1^λ): Given a security parameter λ, generates a secret key sk and the corresponding public key pk.
- Encrypt(pk, m): Encrypts a vector of complex numbers (represented by pairs of floating-point values) m using the public key pk, yielding a ciphertext.
- Decrypt(sk, c): Decrypts a ciphertext c using the secret key sk to recover the approximate plaintext.
- Add(c, x): Adds two ciphertexts or a ciphertext and plaintext c and x element-wise, producing a ciphertext $c + x$.
- Multiply(c, x): Multiplies two ciphertexts or a ciphertext and plaintext c and x element-wise, producing a ciphertext that encrypts $c \cdot x$. We denote this as $c \odot x$.
- Rotate(c, i): Rotates the ciphertext c by i positions to the left in a circular fashion. We denote the rotation operation with ρ, so $\rho(c, i)$ for a ciphertext c encrypting $(a_1, a_2, \ldots, a_s)$ becomes a ciphertext encrypting $(a_i, a_{i+1}, \ldots, a_s, a_1, a_2, \ldots, a_{i-1})$.
- Conjugate(c): Conjugates each complex-valued slot encrypted in ciphertext c. In particular, a slot containing $a+bi$, where a and b are the real and imaginary components, respectively, becomes $a - bi$ after conjugation.

The efficient use of homomorphic encryption schemes like CKKS mainly depends on two factors: minimizing the multiplicative depth of computations and reducing the number of operations. *Multiplicative depth* refers to the number of sequential multiplications applied to a ciphertext. Each time a ciphertext undergoes a Multiply operation, its multiplicative depth increases. After a certain depth is reached, a *bootstrapping* operation is required to enable further multiplication operations. Since bootstrapping is very computationally intensive, it is essential to minimize the multiplicative depth of any homomorphic computation. Besides that, minimizing the overall number of operations is crucial for performance, particularly *key-switching operations*. Key-switching operations include rotations, conjugations, and the relinearization suboperation required for ciphertext–ciphertext multiplications, and are typically more expensive than ciphertext–plaintext multiplications or additions. Leveraging the Single Instruction Multiple Data (SIMD) programming model through the element-wise operations of Add and Multiply helps reduce the number of operations, improving efficiency.

Reducing the overhead in homomorphic encryption depends on how values are assigned to specific slots, a process known as the *packing of a ciphertext*. Optimized packing strategies reduce the computational cost of composite operations, such as matrix multiplications, by enabling more effective SIMD utilization and by lowering both the total number of operations and the multiplicative depth. Additionally, well-designed packing schemes minimize the number of ciphertexts needed to represent a dataset and help avoid costly packing reorganizations (comprised of masking and rotation operations), which in turn reduces the frequency of costly bootstrapping operations. In this paper, we introduce *tricyclic encodings* as a novel packing strategy for enabling the private inference of LLMs. We discuss this more in Section 5.

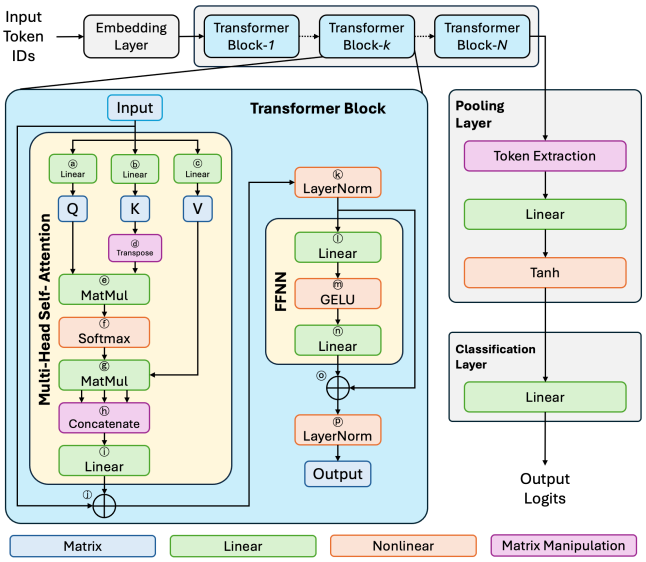

Figure 1: A BERT Model [11].

3 Private LLM Inference

An LLM typically consists of an embedding layer, followed by a series of transformer blocks and one or more output layers. Figure 1 illustrates an example of such a model: BERT [11], in which the output layers include a pooling layer and a classification layer.

3.1 Challenges

The operations of the transformer model are summarized in Li *et al.*'s survey on private LLM inference [22]. Hence, we now focus on the numerous challenges of enabling such inference using homomorphic encryption. Following the common assumption made by Rovida *et al.* [30], Powerformer [27], and Thor [25], we assume that the user computes the embedding layer in plaintext to generate token embeddings. As such, we omit discussion of the embedding layer and concentrate on the remaining computations within the transformer. Even with this assumption, enabling private inference remains highly non-trivial due to several key challenges, which relate to the core encrypted computations illustrated in Figure 1 (we use a circled letter, e.g. (a), to refer to specific transformer operations in Figure 1):

- **Matrix Multiplications:** Both ciphertext–plaintext and ciphertext–ciphertext matrix multiplications are needed.

Work	Batching	Retraining Required	Packing	Matmul Depth		Models	Hardware	Latency (s)
				Ctxt-Ptxt	Ctxt-Ctxt			
Rovida *et al.* [30]	No	No[¶]	row-major	2	2	BERT-Tiny	CPU	≈625[¶]
ELLMo [13]	No	No	row-major	1	3	BERT-Tiny	GPU	8.19
NEXUS [36]	Yes	No	row, column, diagonal-major	$\geq 1^{\dagger}$	$\geq 2^{\dagger}$	BERT-Base	GPU	$37^{\dagger}$
MOAI [37]	Yes	Either	column and diagonal-major	1	1	BERT-Base	GPU	141
EUSTON [12]	Yes	No	row, column, diagonal-major	1	1	BERT-Base	GPU	46
ARION [34]	Yes	No	column and diagonal-major	1	1	BERT-Base	CPU	225
Rho *et al.* [29]	No	Yes	row-major	3	3	BERT-like[‡]	GPU	26.5[‡]
Powerformer [27]	No	Yes	column-major	1	3	BERT-Base	GPU	344
Thor [25]	No	No	diagonal-major	1-3	3	BERT-Base	GPU	602
Tricycle (ours)	No	No	tricyclic encoding	1	1	BERT-Base	GPU	100.5

Table 1: Comparison of Private LLM Inference works. All latency numbers are evaluated over 128 tokens, except as otherwise noted here. [¶]Rovida *et al.*'s [30] system is evaluated over around 45 tokens and although retraining is not required, parts of layer normalization are replaced with learned constants. [‡]Rho *et al.*'s system is evaluated over the matrix dimensions of a BERT-Base model with only 2 transformer layers [29]. It is not specified how many tokens Rho *et al.*'s [29] system is evaluated over. [†]NEXUS [36] often yields misaligned packings, requiring an unspecified amount of additional multiplicative depth to correct and other works [34, 37] have criticized NEXUS's results for benchmarking operations individually rather than end-to-end.

For example, the attention mechanism

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{1}$$

requires two ciphertext–ciphertext matrix multiplications: one for computing $Q \cdot K^T$ (e) and another for multiplying the softmax output by V (g). All other matrix multiplications in a transformer are ciphertext–plaintext (a) (b) (c) (i) (l) (n). Besides the performance costs of matrix multiplications under homomorphic encryption, performing successive matrix multiplications is challenging because these algorithms often require specific input packings that may result in another output packing that is incompatible with the next matrix multiplication. Ciphertext-ciphertext matrix multiplications are especially difficult because they require two specific input packings and often result in a third output packing.

- **Multiple Attention Heads:** The computation of attention is divided into h parallel heads, each computing attention independently (a) – (g). The outputs of these heads are then concatenated (h) and multiplied by an attention output matrix (i). A major challenge is in developing a packing scheme that supports these operations while leveraging the SIMD properties of CKKS to parallelize *all* of the operations in the attention heads.
- **Matrix Transposition:** The attention mechanism also requires transposing the matrix K (d). Under homomorphic encryption, transposing can necessitate changing the packing of a ciphertext, which is computationally expensive.
- **Packing Consistency of Residual Connections:** Each transformer block has two residual connections (j) (o) prior to the layer normalization steps. This requires that the packing of the ciphertext entering and exiting an attention and the feedforward network to remain consistent.
- **Nonlinear Approximations:** The transformer incorporates several nonlinear layers (e.g., softmax (f), layer normalization (k) (o), GELU (m), and tanh) that cannot be directly implemented using only homomorphic additions and multiplications and hence need to be approximated.
- **Row-wise Operations:** Both softmax (f) and layer normalization (k) (o) operate row-wise on their input matrices. Therefore, an effective packing must support efficient row-wise operations.
- **Ciphertext Capacity Limits:** For larger models or when processing many input tokens, it becomes *impossible* to pack entire matrices into the limited slots available in a single ciphertext, even with larger ring dimensions such as $N = 2^{16}$.
- **Bootstrapping Requirement:** Due to the high number of operations in a transformer model, a non-interactive system must incorporate bootstrapping to enable more multiplication operations after a certain multiplicative depth has been reached.

3.2 Prior Work

Private inference for large language models can be broadly divided into interactive approaches based on multiparty computation (MPC) and non-interactive approaches based on HE. Several prior works explore interactive protocols for private transformer inference [16, 17, 23, 24, 26, 38]. While effective, these methods require multiple rounds of communication during inference, introducing significant latency and communication overhead.

Tricycle instead adopts a fully non-interactive HE-based design. Table 1 summarizes recent non-interactive systems [9, 12, 13, 25, 27, 29, 30, 34, 36, 37]. Other concurrent preprints also explore this setting [28, 33, 35]. These systems differ in their assumptions (e.g. batching) as well as in their ciphertext-packing schemes, matrix multiplication algorithms, and target hardware platforms, including CPUs and GPUs.

Some systems rely on batching (e.g., MOAI [37], EUSTON [12], and ARION [34]), which introduces several limitations. First, they assume that many independent inputs (e.g., 32–256) share an encryption key. Although suitable for some deployments, this conflicts with users' preference for individually managed keys. Second, batching reduces amortized latency but not individual end-to-end latency, since each user must wait for the entire batch. For example, MOAI reports an amortized latency of 141 seconds per input for a batch of 256, which we estimate corresponds to a user-perceived latency of approximately 10 hours. Third, batching constrains input flexibility: its efficiency comes from distributing matrices across ciphertexts to reduce rotations and multiplicative depth, but often requires fixed-length padding and large batches (such as 256 inputs in MOAI and ARION) to achieve the reported performance. For these reasons, we argue that batching is inappropriate in many settings.

Other systems improve performance by replacing nonlinearities, such as softmax and layer normalization, with functions that are more efficient to evaluate under HE, such as in Rho *et al.* [29] and Powerformer [27]. These substitutions can substantially reduce the polynomial degree, multiplicative depth, and number of expensive bootstrapping operations required during encrypted inference. However, because the resulting architecture no longer implements the original pretrained model, it must be retrained or fine-tuned using the modified functions. This additional training requirement limits compatibility with existing models and may degrade accuracy relative to the original architecture.

Thor [25] is the closest prior work, as it avoids batching and preserves approximate nonlinear computations without retraining, which Tricycle also targets. We demonstrate that Tricycle achieves significantly improved performance, enabling private inference of BERT-Base with 128 tokens in 100.5 seconds. This yields a 6× latency reduction over Thor and a 3× improvement over Powerformer, despite the latter relying on retraining.

3.3 Bicyclic Encoding as a Starting Point

Our goal is to provide end-to-end private inference of transformer models. A limitation of prior non-batching systems is that the matrix multiplication algorithms introduce additional multiplicative depth, which introduces more expensive bootstrapping operations that compound over multiple transformer layers (Table 1). To bridge this gap, we begin by considering a private LLM inference system built on prior work on bicyclic encodings [4], which introduced generic privacy-preserving matrix multiplications with optimal multiplicative depth. Bicyclic encodings can be viewed as an extension of the diagonal representation underlying Halevi-Shoup matrix–vector multiplications [14].

Bicyclic encodings provide several properties that make them an appealing starting point for private LLM inference. First, bicyclic matrix multiplication achieves optimal multiplicative depth of 1, a critical property in deep neural networks where many expensive bootstrappings lead to high end-to-end latency. Second, bicyclic encodings support both ciphertext–plaintext and ciphertext–ciphertext matrix multiplications while preserving a consistent packing format for inputs and outputs, reducing the need for costly packing transformations across layers. Third, the encoding represents a matrix and its transpose identically, enabling operations such as computing K^T in attention at no additional cost.

Unfortunately, while the bicyclic encoding provides an appealing foundation for private transformer inference, naïvely applying it to transformer models results in poor slot utilization and high computational cost as shown in Table 2.

Approach	$\times W_{Q,K,V}$	$Q \times K^T$	$\times V$	$\times W_o$
Naïve Bicyclic	120960	38592	21504	6528
Tricyclic Encoding	28632	6432	3584	6144
+ Baby-Step Giant-Step	1800	4800	2784	456
+ Lazy Relinearization	1800	2880	1760	456
+ Optimized Block Matmul	1056	1032	656	324
+ Complexification	568	300	272	276
Tricycle (ours)	568	**300**	**272**	276
Thor [25]	192	460	492	122
Powerformer [27]	**122**	380	351	**75**

Table 2: Number of key-switching operations for different matrix multiplications beginning from a naïve bicyclic encoding approach and optimized until Tricycle. These numbers are over a BERT-Base model with parameters $d_{\text{model}} = 768$, $h = 12$, $d_k = 64$, and number of tokens = 128.

4 Overview of Tricycle

Our system, Tricycle, addresses inefficiencies by improving on the bicyclic encoding with a collection of generic and transformer-specific optimizations. Our contributions are as follows:

Tricyclic Encodings: Transformer models employ multi-head attention, in which multiple attention heads are independently evaluated in parallel. A naïve implementation based on bicyclic encodings underutilizes ciphertext slots and therefore fails to fully exploit the SIMD capabilities of CKKS. To address this limitation, we introduce *tricyclic encodings* (§5.1), a novel packing strategy that supports *batched matrix multiplication* (§5.2), enabling multiple attention heads to be evaluated in parallel.

Support for More General Matrix Dimensions: Bicyclic encodings impose restrictions on matrix dimensions that complicate their application to larger models. We develop improved ciphertext–plaintext matrix multiplication algorithms that relax these restrictions and support a broader range of matrix dimensions (§5.3).

Matrix Multiplication Optimizations: We reduce the cost of matrix multiplication through several complementary optimizations that progressively lower the number of key-switching operations, as shown in Table 2. These optimizations are described in §6.1, §6.2, and §6.4.

Efficient Multi-Ciphertext Matrix Multiplications: Matrices in a transformer model can exceed the number of slots available in a single ciphertext, requiring matrix multiplication to be distributed across multiple ciphertexts. We address this challenge by developing efficient block matrix multiplication techniques (§6.3).

Statistical Max Estimation for Softmax: We propose a novel method, called *statistical max estimation*, to stably evaluate softmax under HE (§7.1).

Experimental Evaluation: We implement Tricycle end-to-end and evaluate its performance and accuracy. We demonstrate that Tricycle reduces latency by 6× and 3.4× compared with Thor and Powerformer, respectively. We also conduct ablation studies to quantify the contribution of each optimization (§8).

5 Matrix Multiplication with Tricyclic Encoding

Our first innovation is in proposing the *tricyclic encoding* to encode three-dimensional tensors and enable batch matrix multiplications. Exploiting the multi-head structure of transformers, we develop *batch matrix multiplication* algorithms that parallelize independent attention heads using the SIMD capabilities of homomorphic encryption. This reduces both the number of operations and ciphertexts requiring bootstrapping while preserving the *optimal* multiplicative depth of 1, yielding a nearly h-fold reduction in computation at negligible additional cost. To that end, we introduce three novel matrix multiplication algorithms for multi-head attention and improved ciphertext–plaintext multiplication techniques.

5.1 Tricyclic Encoding

Let $m, n, p \in \mathbb{N}$ be pairwise coprime so that the *least common multiple*, $\mathrm{lcm}(m, n, p) = m \cdot n \cdot p$. Consider the three-dimensional tensor

$$A = \left[a_{i,j,k}\right] \in \mathbb{R}^{m \times n \times p}.$$

A three-dimensional tensor is composed of m matrices with size $(n \times p)$. The *tricyclic encoding* of A is defined by the mapping

$$\varphi : \mathbb{R}^{m \times n \times p} \to \mathbb{R}^{m \cdot n \cdot p},$$

given by

$$\varphi(A)_k = a_{k \bmod m,\, k \bmod n,\, k \bmod p}, \quad \text{for all } k \in \mathbb{Z}_{mnp}.$$

The tricyclic encoding is a strict generalization of the bicyclic encoding since a matrix can be represented as a three-dimensional tensor with its first dimension $m = 1$. Chen *et al.* note that bicyclic encodings can be generalized to an n-cyclic encodings for tensors [4], but they do not further develop this idea or analyze its implications for matrix multiplications or transposes. We address these aspects through tricyclic encodings.

5.1.1 Free Transpose. One particular property of the tricyclic encoding is that it retains the free transpose property of the bicyclic encoding. In fact, for any n-dimensional tensor, the n-cyclic encoding of the tensor and any of its transposes are equivalent. Hence, every transpose is effectively free. We prove this theorem in Appendix B. Tricycle uses this free transpose property for calculating K^T in attention.

5.2 Batch Matrix Multiplication

We develop three different matrix multiplication operations with tricyclic encodings that are each required for calculating multi-head attention.

5.2.1 Batch Matrix Multiplication. Tricycle employs a batch matrix multiplication algorithm to parallelize computation across the h different attention heads in both the $Q \cdot K^T$ product and the subsequent multiplication of the softmax output with V. Given an $(h \times m \times n)$ tensor A and an $(h \times n \times p)$ tensor B, pairwise multiply the h distinct $(m \times n)$ matrices by h corresponding $(n \times p)$ matrices to obtain h distinct $(m \times p)$ matrices, represented by an $(h \times m \times p)$ tensor. The batch matrix multiplication (of $Z = A \cdot B$) is expressed as

$$Z_{i,j,k} = \left[\sum_{c=0}^{n-1} a_{i,j,c} \cdot b_{i,c,k}\right] \in \mathbb{R}^{h \times m \times p}. \tag{2}$$

We prove in Appendix C.1 that the batch matrix multiplication can be evaluated with rotations, multiplications, and sums as follows (where $\varphi(Z)$ is tricyclic encoding of Z):

$$\varphi(Z) = \sum_{c=0}^{n-1} \left(\rho\left(\varphi(A), chm((hm)^{-1} \bmod n)\right) \bmod (hmn)\right) \odot \rho\left(\varphi(B), (chp((hp)^{-1} \bmod n)) \bmod (hnp)\right)\Big). \tag{3}$$

This shows that we can calculate the batch matrix multiplication by computing rotations of every $hm((hm)^{-1} \bmod n)$ and $hp((hp)^{-1} \bmod n)$ for $\varphi(A)$ and $\varphi(B)$, respectively, and summing their product. Here, $(hm)^{-1} \bmod n$ and $(hp)^{-1} \bmod n$ denote the modular inverses of hm and hp modulo n. In Figure 2, we show an example of a batch matrix multiplication for two tensors A and B with tricyclic encodings. Lastly, note that when $h = 1$, the equation simplifies to the same as a matrix multiplication, showing that batch matrix multiplications with tricyclic encodings are a strict generalization of bicyclic matrix multiplications.

5.2.2 Broadcasted Batch Matrix Multiplication. Tricycle uses a broadcasted batch matrix multiplication to compute the h different attention heads for the query (Q), key (K), and value (V) matrices. Given a $(1 \times m \times n)$ tensor A and an $(h \times n \times p)$ tensor B, multiply the single $(m \times n)$ matrix in A by each of the h distinct $(n \times p)$ matrices in B to obtain h distinct $(m \times p)$ matrices, represented by an $(h \times m \times p)$ tensor. This broadcasts the $(m \times n)$ matrix in A across each of the h matrices in B. The broadcasted batch matrix multiplication is expressed as

$$Z_{i,j,k} = \left[\sum_{c=0}^{n-1} a_{0,j,c} \cdot b_{i,c,k}\right] \in \mathbb{R}^{h \times m \times p}. \tag{4}$$

Appendix C.2 proves that the broadcasted batch matrix multiplication is achieved by the following operations:

$$\varphi(Z) = \sum_{c=0}^{n-1} \left(\rho\left(\varphi(A), cm(m^{-1} \bmod n)\right) \bmod (mn)\right) \odot \rho\left(\varphi(B), (chp((hp)^{-1} \bmod n)) \bmod (hnp)\right)\Big). \tag{5}$$

This means that we can compute the broadcasted batch matrix multiplication by computing rotations of every $m(m^{-1} \bmod n)$ and $hp((hp)^{-1} \bmod n)$ for $\varphi(A)$ and $\varphi(B)$, respectively, and summing their product.

$$A \in \mathbb{R}^{2\times5\times7} = \left[a_{i,j,k}\right] \text{ where } i \in \mathbb{Z}_2, j \in \mathbb{Z}_5, k \in \mathbb{Z}_7 \text{ and } B \in \mathbb{R}^{2\times7\times3} = \left[b_{i,j,k}\right] \text{ where } i \in \mathbb{Z}_2, j \in \mathbb{Z}_7, k \in \mathbb{Z}_3$$

$$\varphi(A) = \begin{pmatrix} a_{0,0,0} & a_{1,1,1} & a_{0,2,2} & a_{1,3,3} & a_{0,4,4} & a_{1,0,5} & a_{0,1,6} & a_{1,2,0} & a_{0,3,1} & a_{1,4,2} & a_{0,0,3} & a_{1,1,4} & a_{0,2,5} & a_{1,3,6} & a_{0,4,0} & a_{1,0,1} & \cdots \end{pmatrix}$$

$$\varphi(B) = \begin{pmatrix} b_{0,0,0} & b_{1,1,1} & b_{0,2,2} & b_{1,3,0} & b_{0,4,1} & b_{1,5,2} & b_{0,6,0} & b_{1,0,1} & b_{0,1,2} & b_{1,2,0} & b_{0,3,1} & b_{1,4,2} & b_{0,5,0} & b_{1,6,1} & b_{0,0,2} & b_{1,1,0} & \cdots \end{pmatrix}$$

i	j	$\rho(\varphi(A), i)$	$\rho(\varphi(B), j)$	$\rho(\varphi(A), i) \odot \rho(\varphi(B), j)$
0	0	$(a_{0,0,0}\ a_{1,1,1}\ a_{0,2,2}\ a_{1,3,3}\ a_{0,4,4}\ \cdots)$	$(b_{0,0,0}\ b_{1,1,1}\ b_{0,2,2}\ b_{1,3,0}\ b_{0,4,1}\ \cdots)$	$(a_{0,0,0}\cdot b_{0,0,0}\ a_{1,1,1}\cdot b_{1,1,1}\ a_{0,2,2}\cdot b_{0,2,2}\ a_{1,3,3}\cdot b_{1,3,0}\ a_{0,4,4}\cdot b_{0,4,1}\ \cdots)$
50	36	$(a_{0,0,1}\ a_{1,1,2}\ a_{0,2,3}\ a_{1,3,4}\ a_{0,4,5}\ \cdots)$	$(b_{0,1,0}\ b_{1,2,1}\ b_{0,3,2}\ b_{1,4,0}\ b_{0,5,1}\ \cdots)$	$(a_{0,0,1}\cdot b_{0,1,0}\ a_{1,1,2}\cdot b_{1,2,1}\ a_{0,2,3}\cdot b_{0,3,2}\ a_{1,3,4}\cdot b_{1,4,0}\ a_{0,4,5}\cdot b_{0,5,1}\ \cdots)$
30	30	$(a_{0,0,2}\ a_{1,1,3}\ a_{0,2,4}\ a_{1,3,5}\ a_{0,4,6}\ \cdots)$	$(b_{0,2,0}\ b_{1,3,1}\ b_{0,4,2}\ b_{1,5,0}\ b_{0,6,1}\ \cdots)$	$(a_{0,0,2}\cdot b_{0,2,0}\ a_{1,1,3}\cdot b_{1,3,1}\ a_{0,2,4}\cdot b_{0,4,2}\ a_{1,3,5}\cdot b_{1,5,0}\ a_{0,4,6}\cdot b_{0,6,1}\ \cdots)$
10	24	$(a_{0,0,3}\ a_{1,1,4}\ a_{0,2,5}\ a_{1,3,6}\ a_{0,4,0}\ \cdots)$	$(b_{0,3,0}\ b_{1,4,1}\ b_{0,5,2}\ b_{1,6,0}\ b_{0,0,1}\ \cdots)$	$(a_{0,0,3}\cdot b_{0,3,0}\ a_{1,1,4}\cdot b_{1,4,1}\ a_{0,2,5}\cdot b_{0,5,2}\ a_{1,3,6}\cdot b_{1,6,0}\ a_{0,4,0}\cdot b_{0,0,1}\ \cdots)$
60	18	$(a_{0,0,4}\ a_{1,1,5}\ a_{0,2,6}\ a_{1,3,0}\ a_{0,4,1}\ \cdots)$	$(b_{0,4,0}\ b_{1,5,1}\ b_{0,6,2}\ b_{1,0,0}\ b_{0,1,1}\ \cdots)$	$(a_{0,0,4}\cdot b_{0,4,0}\ a_{1,1,5}\cdot b_{1,5,1}\ a_{0,2,6}\cdot b_{0,6,2}\ a_{1,3,0}\cdot b_{1,0,0}\ a_{0,4,1}\cdot b_{0,1,1}\ \cdots)$
40	12	$(a_{0,0,5}\ a_{1,1,6}\ a_{0,2,0}\ a_{1,3,1}\ a_{0,4,2}\ \cdots)$	$(b_{0,5,0}\ b_{1,6,1}\ b_{0,0,2}\ b_{1,1,0}\ b_{0,2,1}\ \cdots)$	$(a_{0,0,5}\cdot b_{0,5,0}\ a_{1,1,6}\cdot b_{1,6,1}\ a_{0,2,0}\cdot b_{0,0,2}\ a_{1,3,1}\cdot b_{1,1,0}\ a_{0,4,2}\cdot b_{0,2,1}\ \cdots)$
20	6	$(a_{0,0,6}\ a_{1,1,0}\ a_{0,2,1}\ a_{1,3,2}\ a_{0,4,3}\ \cdots)$	$(b_{0,6,0}\ b_{1,0,1}\ b_{0,1,2}\ b_{1,2,0}\ b_{0,3,1}\ \cdots)$	$(a_{0,0,6}\cdot b_{0,6,0}\ a_{1,1,0}\cdot b_{1,0,1}\ a_{0,2,1}\cdot b_{0,1,2}\ a_{1,3,2}\cdot b_{1,2,0}\ a_{0,4,3}\cdot b_{0,3,1}\ \cdots)$

Figure 2: Batch Matrix Multiplication with Tricyclic Encoding. The first two columns of the table list the rotational shifts i and j required for tricyclic encodings $\varphi(A)$ and $\varphi(B)$. The third and fourth columns display the corresponding rotations of these encodings. The rotational shifts for $\varphi(A)$ occur every 50 mod 70**, by setting $h = 2$, $m = 5$, and $n = 7$ in the expression $(chm((hm)^{-1} \bmod n)) \bmod (hmn)$ for $c = 0, 1, \ldots, n-1$. Likewise, the rotational shifts for $\varphi(B)$ occur every** 36 mod 42**, obtained by substituting $h = 2$, $n = 7$, and $p = 3$ into $(chp((hp)^{-1} \bmod n)) \bmod (hnp)$. The final column contains the element-wise product of the third and fourth columns, and summing these values pairwise yields the tricyclic encoding $\varphi(A \cdot B)$.**

5.2.3 Concatenated Matrix Multiplication. Our system computes the final linear projection in attention using a concatenated matrix multiplication algorithm. Given an $(h \times m \times n)$ tensor A and an $(h \times n \times p)$ tensor B, pairwise multiply the h distinct $(m \times n)$ matrices by h distinct $(n \times p)$ matrices to obtain h distinct $(m \times p)$ matrices. Then, sum all h matrices to obtain one $(m \times p)$ matrix, which is represented by a $(1 \times m \times p)$ tensor. Formally, the concatenated matrix multiplication is expressed as

$$Z_{0,j,k} = \left[\sum_{d=0}^{h-1}\sum_{c=0}^{n-1} a_{d,j,c} \cdot b_{d,c,k}\right] \in \mathbb{R}^{1\times m\times p}. \tag{6}$$

We prove in Appendix C.3 that this can be computed using tricyclic encodings as follows:

$$\begin{aligned}\varphi(Z) = \sum_{d=0}^{h-1}\sum_{c=0}^{n-1} (&\rho(\varphi(A), (d\cdot r_1 + c\cdot r_2) \bmod (h\cdot m\cdot n)) \odot \\ &\rho(\varphi(B), (d\cdot r_3 + c\cdot r_4) \bmod (h\cdot n\cdot p))) \\ &\text{where } r_1 = m\cdot n\cdot ((m\cdot n)^{-1} \bmod h), \\ &\quad r_2 = h\cdot m\cdot ((h\cdot m)^{-1} \bmod n), \\ &\quad r_3 = n\cdot p\cdot ((n\cdot p)^{-1} \bmod h), \\ &\quad r_4 = h\cdot p\cdot ((h\cdot p)^{-1} \bmod n).\end{aligned} \tag{7}$$

We briefly elucidate on Equation 7. The variables d and c are summation indices where d iterates over the different matrices in the tensorized operands A and B while c iterates over the columns of each matrix in A and the corresponding rows in each matrix in B. The values r_1, r_2, r_3, and r_4 represent rotation strides. For $\varphi(A)$, r_1 is the rotation step needed to advance to the next matrix while keeping the same row and column indices and r_2 is the rotation step needed to advance to the next column within the same matrix and row. For $\varphi(B)$, r_3 is the rotation step needed to advance to the next matrix while keeping the same row and column indices and r_4 is the rotation step needed to advance to the next row within the same matrix and column.

5.3 Improved Ciphertext–Plaintext Matmuls

In transformer blocks, ciphertext–plaintext matrix multiplications arise in every multiplication involving learned model weights. As a result, it is critical that these operations be both efficient and broadly applicable, with minimal structural constraints. Our next contribution addresses this need by introducing improved ciphertext–plaintext matrix multiplication algorithms that impose *strictly fewer limitations* than prior work [4]. Consider the batch matrix multiplication between a ciphertext tensor of shape $(h_1 \times m \times n)$ and a plaintext tensor of shape $(h_2 \times n \times p)$, yielding a ciphertext tensor of shape $(h_3 \times m \times p)$. Our enhancements apply to all ciphertext–plaintext matrix multiplications in Tricycle and improve upon prior work [4] in the following ways:

- **Coprimeness:** Prior work [4] required all matrices to satisfy coprime dimension constraints. In contrast, our approach requires coprimeness only for the input and output ciphertext matrices, with no such restriction on the plaintext matrix. Formally, we require $\text{lcm}(h_1, m, n) = h_1 \cdot m \cdot n$ and $\text{lcm}(h_3, m, p) = h_3 \cdot m \cdot p$.
- **Larger Plaintext Matrix Size:** Unlike prior work [4], which required the plaintext matrix to fit within half of the available ciphertext slots, our approach removes this restriction. In fact, the plaintext matrix may contain more values than available slots.
- **Slot Efficiency:** Prior work [4] limited ciphertext matrices to using at most half of the available slots. Our approach lifts this limitation and requires only that the encrypted values fit within the available slots. Formally, for s slots, we require $h_1 mn < s$ and $h_3 mp < s$.

As an illustrative example that demonstrates our approach and its fewer limitations, suppose that there are $s = 8$ available slots in a ciphertext. Consider the $(1 \times 2 \times 3)$ ciphertext tensor A and the $(1 \times 3 \times 3)$ plaintext tensor B:

$$A = \left[\begin{pmatrix} a & b & c \\ d & e & f \end{pmatrix}\right] \text{ and } B = \left[\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}\right]$$

The tricyclic encoding of A, $\varphi(A)$ is (a, e, c, d, b, f) and is encrypted with zero padding. Notice the slot efficiency of $\varphi(A)$ and larger, non-coprime matrix dimensions of B. The matrix multiplication can be computed by multiplying rotations of the ciphertext and diagonals of the plaintext.

i	$\rho(\varphi(A), i)$	BatchDiag(B)	$\rho(\varphi(A), i) \odot$ BatchDiag(B)
0	$(a\ e\ c\ d\ b\ f\ 0\ 0)$	$(1\ 5\ 9\ 1\ 5\ 9\ 0\ 0)$	$(1a\ 5e\ 9c\ 1d\ 5b\ 9f\ 0\ 0)$
4	$(b\ f\ 0\ 0\ a\ e\ c\ d)$	$(4\ 8\ 0\ 0\ 0\ 0\ 0\ 0)$	$(4b\ 8f\ 0\ 0\ 0\ 0\ 0\ 0)$
−2	$(0\ 0\ a\ e\ c\ d\ b\ f)$	$(0\ 0\ 3\ 4\ 8\ 3\ 0\ 0)$	$(0\ 0\ 3a\ 4e\ 8c\ 3d\ 0\ 0)$
2	$(c\ d\ b\ f\ 0\ 0\ a\ e)$	$(7\ 2\ 6\ 7\ 0\ 0\ 0\ 0)$	$(7c\ 2d\ 6b\ 7f\ 0\ 0\ 0\ 0)$
−4	$(b\ f\ 0\ 0\ a\ e\ c\ d)$	$(0\ 0\ 0\ 0\ 2\ 6\ 0\ 0)$	$(0\ 0\ 0\ 0\ 2a\ 6e\ 0\ 0)$

By summing the right column, we obtain $\varphi(A \cdot B)$ with zero padding for extra slots. Multiplying by a plaintext is inherently more flexible than multiplying by a ciphertext since plaintext values can be rearranged at no cost and do not need to be represented with a tricyclic encoding to ensure correctness. Our method for determining the plaintext diagonal is similar to how the tricyclic encoding traverses the tensor, except that it does not require every value in the tensor to be visited exactly once. Hence, there is no coprimeness or size requirement for the plaintext tensor.

We derive all ciphertext–plaintext matrix multiplication algorithms in Appendix C and briefly describe the diagonal traversal for batch matrix multiplications (Equation 2) as follows:

$$\varphi(Z) = \sum_{c=0}^{n-1} \big(\rho(\varphi(A), (chm((hm)^{-1} \bmod n)) \bmod (hmn)) \odot \text{BatchDiag}(B, c)\big) \quad (8)$$

where $\text{BatchDiag}(B, c)_k = b_{k \bmod h, (k+c) \bmod n, k \bmod p}$ for $k \in \mathbb{Z}_{hmp}$. Note that Equation 8 does not assume a tricyclic encoding for B, avoiding the coprimeness requirement.

6 Matmul Performance Optimizations

In this section, we introduce several optimizations that improve the performance of our matrix multiplication algorithms and reduce key switching overhead.

6.1 Baby-Step Giant-Step Algorithm

We first reduce the number of rotations required in our matrix multiplications by using the Baby-Step Giant-Step (BSGS) algorithm [6]. BSGS is applicable when a computation involves summing a sequence of periodically incrementing rotations, which we notice naturally occurs in matrix multiplications using tricyclic and bicyclic encodings. As a result, our matrix multiplication algorithms are strictly more efficient than prior work [4], which does not employ BSGS-based rotation optimizations.

6.1.1 Ciphertext–Plaintext Matmuls. We now describe the BSGS algorithm and its application to our ciphertext–plaintext matrix multiplications. Let T be the periodicity of rotations (e.g. rotations of every T) and N be the number of sums of rotations and multiplications required. Let the number of baby steps be $N_1 = O(\sqrt{N})$ that we can precompute. For convenience, let N_1 divide N evenly and the number of giant steps be $N_2 = N/N_1$. Let x be a ciphertext and y be a plaintext. The left side of Equation (9) expresses the computation as a sum of products between rotated ciphertexts x and plaintexts y. BSGS optimizes this summation by reorganizing it as shown in Equation (10).

$$\sum_{c=0}^{N-1} \rho(x, T \cdot c) \odot y = \sum_{j=0}^{N_2-1} \sum_{i=0}^{N_1-1} \rho(x, T \cdot (i + j \cdot N_1)) \odot y \quad (9)$$

$$= \sum_{j=0}^{N_2-1} \rho\left(\sum_{i=0}^{N_1-1} \rho(x, T \cdot i) \odot \rho(y, -T \cdot N_1 \cdot j), T \cdot N_1 \cdot j\right). \quad (10)$$

We remark that applying BSGS does not require $N_1 \cdot N_2 = N$ exactly, and only requires that every rotation be grouped into sets of giant-steps, where the differences between rotations sizes align with precomputed baby-steps. In Equation 10, rotating the plaintext, $\rho(y, -T \cdot N_1 \cdot j)$, is done in advance (effectively free), while the baby-step rotations $\rho(x, T \cdot i)$ for $0 \le i < N_1$ can be precomputed once. For each giant-step index j, the appropriately rotated plaintexts are multiplied with the corresponding baby-step ciphertexts and summed, producing an intermediate giant-step result. The intermediate result is then rotated by the appropriate giant-step offset (the outermost rotation) and all giant-step results are summed to obtain the final output.

By choosing appropriate N_1 and N_2, and substituting for the periodicity $T = hm$ we can reduce the number of rotations in our ciphertext–plaintext matrix multiplications from $O(n)$ to between $2\sqrt{n} - 2$ and $2\sqrt{2n} - 2$. Appendix D shows that our ciphertext–plaintext matrix multiplications in Equation 8 simplify to

$$\sum_{c=0}^{n-1} (\rho(\varphi(A), c \cdot hm) \odot \text{BatchDiag}'(B, c)) \quad (11)$$

where $\text{BatchDiag}'(B, c)_k = b_{k \bmod h, (k+c \cdot hm) \bmod n, k \bmod p}$ for $k \in \mathbb{Z}_{hmp}$. This proves that BSGS can be applied for $x = \varphi(A)$, $y = \text{BatchDiag}'(B, c)$, and $T = hm$.

6.1.2 Ciphertext–Ciphertext Matmuls. Our next innovation is in applying BSGS to ciphertext–ciphertext matrix multiplications. We observe that these matrix multiplications can be expressed as in terms of Equation 12, where x and y are both ciphertexts, T is the periodicity of rotations of x, and $\text{RotY}(c)$ represents the rotations required for y in the c-th summation term. The equation shows a summation of products of periodic rotations of x with rotations of y. We reorganize this sum using BSGS to obtain Equation 14 as follows:

$$\sum_{c=0}^{N-1} \rho(x, T \cdot c) \odot \rho(y, \text{RotY}(c)) \quad (12)$$

$$= \sum_{j=0}^{N_2-1} \sum_{i=0}^{N_1-1} \rho(x, T(i + j \cdot N_1)) \odot \rho(y, \text{RotY}(i + j \cdot N_1)) \quad (13)$$

$$= \sum_{j=0}^{N_2-1} \rho\left(\sum_{i=0}^{N_1-1} \rho(x, T \cdot i) \odot \rho(y, \text{RotY}(i + j \cdot N_1) - T \cdot N_1 \cdot j), T \cdot N_1 \cdot j\right). \quad (14)$$

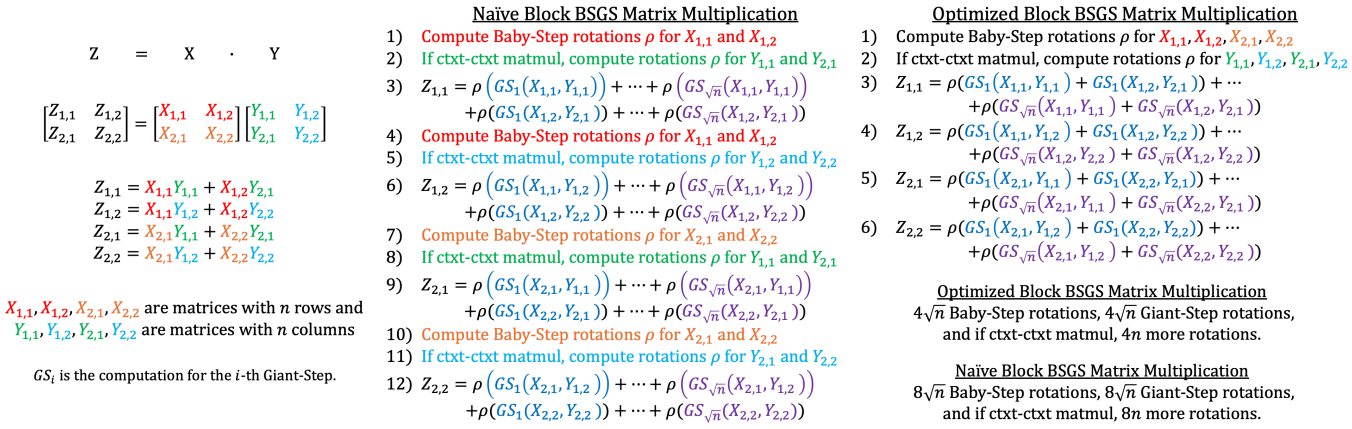

Figure 3: Our optimized block matrix multiplication computes baby-step rotations, rotations for Y, relinearizations, and giant-step rotations only once compared to the naïve block matrix multiplication, improving performance.

In Equation 14, the baby-steps correspond to $\rho(x, T \cdot i)$, while the outermost rotations constitute the giant-steps. This BSGS decomposition reduces the rotation count for ciphertext–ciphertext matrix multiplications from $2n-2$ to approximately $n + 2\sqrt{n} - 3$. Although y still requires $n-1$ rotations, x requires only $N_1 - 1 \approx \sqrt{n} - 1$ rotations to precompute its baby-steps. After that, an additional $N_2 - 1 \approx \sqrt{n} - 1$ giant-step rotations are applied to the intermediate products.

We prove in Appendix E that batch matrix multiplications under tricyclic encodings follow the form

$$\varphi(Z) = \sum_{c'=0}^{n-1} \left(\rho\left(\varphi(A), (c' \cdot hm \bmod (hmn)\right) \odot \rho\left(\varphi(B), (hp(c' \cdot m(p)^{-1} \bmod n)) \bmod (hnp)\right)\right) \quad (15)$$

proving that BSGS can be applied for $x = \varphi(A)$, $y = \varphi(B)$, periodicity $T = hm$, and $\mathrm{RotY}(c) = (hp(c \cdot m(p)^{-1} \bmod n)) \bmod (hnp)$.

6.2 Lazy Relinearization

We next apply the lazy relinearization optimization [25] to ciphertext–ciphertext matrix multiplications. Relinearization is a key-switching operation performed after the product of two ciphertexts. Lazy relinearization delays this operation, allowing multiple ciphertext–ciphertext products to be summed before relinearization is applied. In Tricycle, we apply lazy relinearization by first summing the ciphertext–ciphertext products corresponding to all baby-steps within a giant-step, and then relinearizing the result before rotating the giant-step. This reduces the n relinearizations in a ciphertext–ciphertext matrix multiplication down to just $\sqrt{n}$ relinearizations.

6.3 Optimized Block Matrix Multiplications

Our next optimization reduces rotation and relinearization overhead in block matrix multiplications. When matrix dimensions exceed the available ciphertext slot capacity, the matrices are partitioned into smaller blocks and matrix multiplications are performed across these blocks. A naïve block matrix multiplication implementation, however, repeats many key-switching operations across these block computations. We therefore develop an optimized block matrix multiplication method that eliminates redundant rotations and relinearizations.

Figure 3 contrasts a naïve block matrix multiplication with our optimized approach. Matrices X and Y are each partitioned into four matrix blocks (where each of X's partitions have the same dimensions, and the same is true of Y). Matrix X is encrypted as a ciphertext and Y can be either a plaintext (for a ciphertext–plaintext matrix multiplication) or a ciphertext (for a ciphertext–ciphertext matrix multiplication). The goal is to compute the block matrix product $Z = X \cdot Y$. Computing the four blocks of Z requires eight block-level matrix multiplications. A naïve BSGS-based approach evaluates each matrix multiplication independently, resulting in substantial redundancy in rotations. In contrast, our optimized method eliminates these redundancies by observing that baby-step rotations for each matrix can be computed once and reused across all multiplications. Similarly, giant-step rotation redundancy can be eliminated by aggregating giant-steps (with the same rotation) before applying a single rotation. For ciphertext–ciphertext matrix multiplications, we further optimize by precomputing the rotations of the Y ciphertexts once and deferring relinearization lazily until after aggregating giant-step contributions, reducing overall overhead.

Performance Analysis: A naïve block BSGS matrix multiplication between a matrix X partitioned into an $(a \times b)$ ciphertext block grid (with each block having n rows) with a plaintext block matrix Y partitioned into a $(b \times c)$ block grid (with each block having n columns) requires $2abc(\sqrt{n} - 1)$ rotations. With our optimized block matrix multiplications, this is reduced to $ab(\sqrt{n} - 1)$ baby-step and $ac(\sqrt{n} - 1)$ giant-step rotations. Similarly, for a ciphertext–ciphertext block matrix multiplication (where Y is also a block of ciphertexts), the number of key switching operations reduces from $abc(n + 2\sqrt{n} - 3)$ rotations and $abc(\sqrt{n})$ relinearizations to just $ab(\sqrt{n} - 1)$ baby-step rotations, $ac(\sqrt{n} - 1)$ giant-step rotations, and $bc(n-1)$ ciphertext Y rotations and $ac\sqrt{n}$ relinearizations.

Optimized BSGS Parameters: Tricycle takes this an additional step further by rebalancing the number of baby-steps and giant-steps based on the number of input and output block matrices

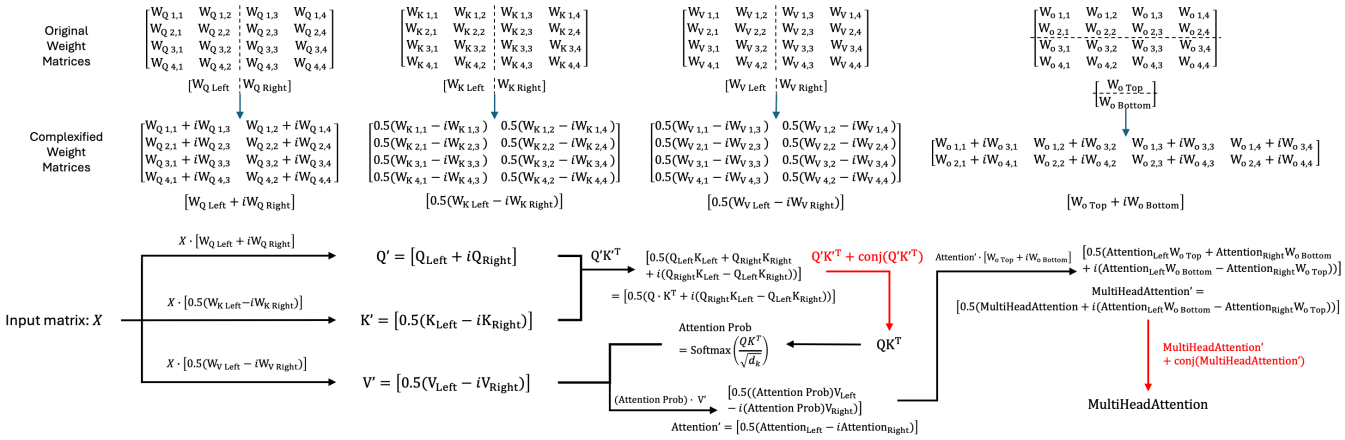

Figure 4: Tricycle's Free Attention Complexification generalizes to multiple attention heads.

to minimize key-switching operations. This further reduces the number of rotations in a ciphertext–plaintext block matrix multiplication to $2a(\sqrt{nbc} - 1)$. For ciphertext–ciphertext block matrix multiplications, our optimized choice of baby-steps and giant-steps reduces the number of total key-switching operations to $2a(\sqrt{2nbc} - 1) + bc(n - 1)$.

We illustrate this rebalancing optimization using an example. Multiplying a (2×2) ciphertext block matrix by a (2×1) plaintext block matrix produces a (2×1) output block matrix. The computation has four input ciphertexts but only two output ciphertexts. Because the baby-step cost scales with the number of input ciphertexts whereas the giant-step cost scales with the number of output ciphertexts, we reduce the total number of rotations by using twice as many giant-steps compared to baby-steps.

6.4 Free Attention Complexification

We introduce a novel technique, called *free attention complexification*, which enables complexified attention computations without increasing multiplicative depth. Complexification leverages CKKS's native complex arithmetic to pack data in both the real and imaginary components of complex numbers, effectively doubling slot utilization and reducing the number of operations. Prior approaches either incurred additional multiplicative depth to apply complexification (e.g. Thor [25]) or apply only when input matrices are partitioned across multiple ciphertexts in specific ways (e.g. Powerformer [27] and Thor [25]), making them less applicable to different models (especially smaller models) and token sequences of varying length. In contrast, our approach is tailored to the structure of attention, incurs no additional multiplicative depth, does not require partitioning of input ciphertext matrices, and applies naturally across transformer model sizes.

Our approach to free attention complexification is shown in Figure 4 and operates by partitioning and complexifying the weight matrices rather than the input ciphertexts, which remain fully packed in the real values. The weight matrices W_Q, W_K, and W_V are partitioned and complexified along their vertical dimension, while the output projection W_o is complexified along its horizontal dimension. When the input matrix X is multiplied by the vertically complexified W_Q, W_K, and W_V, the resulting Q, K, and V matrices encode their left halves in the real component and their right halves in the imaginary component.

To compute attention scores, we multiply the complexified Q and K^T. By exploiting the distributive properties of complex arithmetic, the real and imaginary components of Q and K^T multiply such that the desired QK^T result appears entirely in the real part of the ciphertext, while the imaginary component contains only unwanted values. The real result is then isolated by summing the ciphertext with its conjugate (to eliminate the imaginary component) and folding a division by two into the complexified weights (W_K).

After applying softmax to obtain attention probabilities, the same idea is applied to V and the output projection. The attention probability matrix remains fully real-valued; multiplying it by the complexified V produces an output whose real and imaginary components again encode different halves of the attention matrix. A final multiplication with the horizontally complexified W_o places the desired output only in the real component, which is extracted by adding the result to its conjugate.

6.5 Other Optimizations

Tricycle applies several other notable optimizations for matrix multiplications:

- Lazy rescaling minimizes the rescaling operations (which are required for multiplications) by delaying it until the end of the matrix multiplications [3, 25].
- Hoisting [15] and double hoisting [3] optimizations amortize the costs of multiple rotations. While our Tricycle implementation (in HEaaN [20]) is unable to take advantage of this optimization due to differences in library support, our OpenFHE [2] implementation of the tricyclic matrix multiplications has implemented both forms of hoisting.[1]
- Complexification of the fully connected layers.
- Preprocessing by arranging weights in advance of the matrix multiplications.

[1] https://bitbucket.org/tricyclepaper/tricyclic-matmul.

Approach	$\times W_{Q,K,V}$	$Q \times K^T$	$\times V$	$\times W_o$
Thor [25]	3	4	3	**1**
Powerformer [27]	**1**	5	3	**1**
Bicyclic Encoding	2	**1**	**1**	**1**
Tricycle (ours)	2	**1**	**1**	**1**

Table 3: Multiplicative Depth (i.e. level consumption) of matrix multiplications during self-attention.

Approach	$\times W_{Q,K,V}$	$Q \times K^T$	$\times V$	$\times W_o$	Total
Tricycle (ours)	1136	856	704	552	3248
Thor [25]	260	1432	1496	192	3380
Powerformer [27]	244	1520	1404	150	3318

Table 4: Number of key-switching operations for different matrix multiplications over a BERT-Base model with parameters $d_{\text{model}} = 768$, $h = 12$, $d_k = 64$, and number of tokens = 256.

- GPU pipelining host-to-device memory transfers during matrix multiplication computation.
- Optimized GPU memory allocation and deallocation.

6.6 Complexity Comparison

We briefly compare Tricycle with Thor [25], which uses diagonal packing, and Powerformer [27], which builds on the column-wise packing of Jiang *et al.* [18].

Tricyclic encodings primarily offer lower multiplicative depth (Table 3), reducing costly bootstrapping in an end-to-end system. Every transformer linear layer has an optimal depth of one, except the first, which requires an additional reshaping operation to meet the encoding's coprimality constraints (Appendices H and I). A baseline bicyclic encoding approach would similarly have optimal multiplicative depth for all but the first layer. The tricyclic encoding also supports matrix transposition without additional homomorphic operations. By contrast, Thor and Powerformer do not always achieve depth one, and Powerformer incurs extra operations to transpose K.

For the BERT-Base dimensions in Table 2, Tricycle uses more key-switching operations than Thor and Powerformer for ciphertext–plaintext multiplications, but fewer for ciphertext–ciphertext multiplications. This is because although Tricycle's asymptotic complexity is comparable or better than Thor and Powerformer, Tricycle's constants are higher.

For multiplying an $(m \times n)$ matrix by an $(n \times p)$ matrix, Tricycle's ciphertext–plaintext matrix multiplication algorithm uses $O(\sqrt{n})$, specifically between $2\sqrt{n}$ and $2\sqrt{2n}$, rotations. This is comparable to Powerformer's approximately $2\sqrt{n}$ rotations and favorable to Thor's $O(p)$. For ciphertext–ciphertext matrix multiplication, Tricycle requires $n + 2\sqrt{2n}$ key-switching operations, which is fewer than Powerformer's $n + 2\sqrt{2n} + 5\sqrt{n}$ and Thor's $O(mn/c)$ for $n = p$ and constant c.

Tricycle's higher practical ciphertext–plaintext key-switching count has several causes. First, zero-padding to satisfy coprimality constraints reduces packing efficiency. For example, with 128 tokens in BERT-Base, Tricycle and Thor require four ciphertexts to pack the input matrices, whereas Powerformer requires only three. Second, the approaches differ in how real-valued matrices are complexified. Thor and Powerformer complexify input matrices, which reduces the number of rotations required for ciphertext–plaintext matrix multiplications relative to Tricycle, but imposing additional assumptions on model dimensions and input sizes. Third, when more than half of the available slots are occupied, ciphertext–plaintext matrix multiplications under tricyclic encoding require additional negative rotations, which explains a constant factor up to $\sqrt{2}$ larger. Consequently, Tricycle's primary advantages are its optimal multiplicative depth and favorable asymptotic complexity, while Thor and Powerformer may require fewer key-switching operations for ciphertext–plaintext matrix multiplications.

For longer sequence lengths, Tricycle exhibits more favorable scaling than Thor and Powerformer, requiring the fewest key-switching operations overall at 256 tokens. As shown in Table 4, Tricycle requires 3,248 key-switching operations, compared with 3,380 for Thor and 3,318 for Powerformer. Because the published implementations of Thor and Powerformer only support BERT-Base with sequence lengths of 128 tokens, we derive Thor's key-switching counts directly from its published equations and obtain Powerformer's counts by assuming appropriate scaling of the matrix multiplication dimensions to 256 tokens. Although Tricycle incurs more key-switching operations for the ciphertext–plaintext matrix multiplications, its ciphertext–ciphertext matrix multiplications scale more favorably with sequence length. In particular, its lower costs for QK^T and the softmax result multiplied by V offset the higher ciphertext–plaintext matrix multiplication costs. As a result, Tricycle has the lowest total key-switching count at 256 tokens, indicating its advantage as sequence length increases.

7 Nonlinear Layers

We approximate the nonlinear layers of the transformer using polynomials. Among these, only softmax and layer normalization require special attention to packing, as they involve row-wise sums prior to computing an inverse or inverse square root. For a tricyclically encoded $(h \times m \times n)$ tensor, summing up values in the same matrix and row only requires rotation increments of $h \cdot m$ since values in the same matrix and row are every $h \cdot m$ values in the tricyclic encoding. Our algorithms for evaluating row-wise summations require $O(\log n)$ rotations and are described in Appendix F, demonstrating that tricyclic encodings efficiently support row-wise operations.

7.1 Softmax

To calculate attention, a softmax calculation is required. Softmax is defined as

$$\text{softmax}(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}. \tag{16}$$

In plaintext settings, softmax can suffer from numerical instability because exponentiating large input values may lead to overflow, while very negative values can yield outputs so small they become indistinguishable from zero. A common stabilization technique is to subtract the maximum input value from all terms, taking advantage

Table 5: Comparison of Softmax Approaches

Work	Stable	Faithful Approx.	Mult. Depth
Rovida *et al.* [30]	✗	✓	15
NEXUS [36]	✗	✓	16
Rho *et al.* [29]	✓	✗	16
Powerformer [27]	✓	✗	3
De Castro *et al.* [9]	✓	✓	133
Cho *et al.* [8]	✓	✓	40
Thor [25]	✓	✓	30
Tricycle (ours)	✓	✓	18

of the fact that softmax is invariant to adding or subtracting the same constant across all inputs. This ensures that the largest input becomes zero so exponentiating it yields 1, and the remaining values lie between 0 and 1, greatly improving numerical stability. Under homomorphic encryption, these issues are exacerbated. Large values accelerate ciphertext noise growth, while small values are more easily lost to noise. Applying the same stabilization technique is far more difficult in this setting because computing the maximum value under encryption is both nontrivial and computationally expensive.

Table 5 compares the different softmax approaches under homomorphic encryption in the literature. The table reveals a trilemma in the existing literature: no prior method simultaneously achieves all three of stability, faithful approximation of softmax, and low multiplicative depth. A *faithful approximation* of softmax closely replicates its behavior, enabling it to serve as a drop-in replacement without requiring model retraining. Powerformer [27] and Rho *et al.* [29], for instance, do not faithfully approximate softmax, instead opting for more HE-friendly alternatives with stability. However, these alternatives explicitly require retraining the entire model for the modified nonlinearity and lead to degraded model accuracy due to their departure from the original softmax behavior. Other methods preserve the approximation and maintain low multiplicative depth but give up on stability. NEXUS [36], for example, attempts to stabilize outputs by subtracting a fixed constant from all inputs. This heuristic falls short because (i) if the constant is too small, exponentiation results in large numbers and (ii) if the constant is too large, exponentiation vanishes to zero. Lastly, some methods provide both stability and faithful approximation, but at the cost of high multiplicative depth. De Castro *et al.* [9] achieve this by computing the maximum under homomorphic encryption, incurring a huge multiplicative depth of 133. Thor [25] and Cho *et al.* [8] apply iterative squaring and normalization to stabilize softmax but these methods require a multiplicative depth of at least 30.

Statistical Max Estimation: We resolve the trilemma of stability, faithful approximation, and achieving a low multiplicative depth of 18 by presenting a straightforward yet effective method to stabilize softmax. Our approach is based on the following question: *Could the maximum be estimated for stability from statistics such as mean and variance that are easy to calculate under homomorphic encryption?*

To answer this question, we leverage a result from statistics. Suppose there are n independent and identically distributed random variables $X_1, X_2, ..., X_n$ drawn from the Gaussian distribution $X_i \sim \mathcal{N}(\mu, \sigma^2)$. It is known that that the expected maximum can be upper bounded [21] as follows:

$$\mathbb{E}[\max_{1 \le i \le n} X_i] \le \mu + \sigma\sqrt{2\ln(n)}. \tag{17}$$

Assuming that the inputs to the softmax function are approximately Gaussian, we first compute their mean and variance under homomorphic encryption, and then apply this inequality to estimate the maximum. To reduce the computational cost and multiplicative depth of calculating the standard deviation, σ, we approximate the square root with a simple linear expression ($\sigma \approx \alpha_0 + \alpha_1 \cdot \sigma^2$). We choose $\alpha_0 = 1.25$ and $\alpha_1 = 0.1$ to approximate $\sqrt{x}$ up to 100. Because we only require a rough estimate rather than the exact value, this approximation is adequate for our purposes. Our statistical max estimation algorithm is fully described in Appendix G, Algorithm 3.

A natural question is whether the inputs to the softmax function in a transformer follow a Gaussian distribution. Empirically, we observe that the first three layers of a pretrained BERT-Base model exhibit slightly heavier tails than a Gaussian, whereas the remaining nine layers more closely match Gaussian behavior (Appendix G, Table 11). Accordingly, for the first three layers, we adjust the statistical max estimation by modifying the square root approximation to impose a stronger variance-dependent offset than would be used under a Gaussian assumption. Nevertheless, follow-up work has directly applied our statistical max estimation technique [34] or variants of it [13].

To estimate softmax itself, we estimate the exponentiation with the approximation $e^x \approx \left(1 + x/2^k\right)^{2^k}$ for $k = 7$. We aggressively fold the $1/2^k$ constant used for exponentiation with the max estimation subtraction, which results in the max estimation costing only one extra multiplicative depth, which is inexpensive. Lastly, we approximate the inverse with the Chebyshev approximation [5]. In total, our softmax uses only 9 multiplicative depth for the statistical max estimation and exponentiation and 9 multiplicative depth for the inverse.

7.2 Layer Normalization

Each transformer block applies layer normalization twice: once after the attention layer's residual connection, and again after the feedforward layer's residual connection. In both cases, the normalization is performed row-wise over matrices. For a given row vector $x \in \mathbb{R}_n$ with normalization weights $\gamma \in \mathbb{R}_n$ and biases $\beta \in \mathbb{R}_n$ the layer normalization is defined as

$$\text{LayerNorm}(x)_i = \frac{x_i - \mu}{\sigma} \cdot \gamma_i + \beta_i \tag{18}$$

where μ is the mean of x and σ is its standard deviation.

To compute this normalization, we first calculate the mean and variance in the standard way. Finally, we approximate the inverse square root of the variance using a Chebyshev polynomial approximation [5].

7.3 GELU & Tanh

Every transformer block employs a GELU activation in its feedforward network, while the output layers of the transformer model use the hyperbolic tangent (tanh) function. GELU is defined as $\text{GELU}(x) = \frac{x}{2}\left[1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right]$ where the error function is given as

Operation	BERT-Tiny	BERT-Base						
	Tricycle	Tricycle	Thor [25]	Diff	Powerformer [27]	Diff	Thor [25] in HEaaN [20]	Diff
$\times W_{Q,K,V}$	0.29	20.04	49.77	29.73	57.65	37.61	16.65	-3.39
Reshape	0.15	0.96	–	-0.96	–	-0.96	–	-0.96
$Q \times K^T$	0.18	2.99	16.25	13.26	28.76	25.77	73.98	70.99
Softmax	0.26	3.96	15.53	11.57	0.75	-3.21	6.84	2.88
$\times V$	0.21	1.68	13.08	11.40	18.95	17.27	68.49	66.81
$\times W_o$	0.11	4.96	27.43	22.47	22.54	17.58	11.62	6.66
Layer Norm 1	0.46	4.93	7.13	2.20	0.37	-4.56	4.69	-0.24
FF Layer 1	0.20	17.52	49.80	32.28	59.21	41.69	22.67	5.15
GELU	0.33	7.26	29.42	22.16	8.31	1.05	7.26	0.00
FF Layer 2	0.12	12.33	49.19	36.86	43.77	31.44	20.80	8.47
Layer Norm 2	0.47	4.94	4.10	-0.84	0.30	-4.64	2.77	-2.17
Bootstrapping	1.45	18.14	337.86	319.72	103.72	85.58	61.24	43.10
Total	**4.39**	**100.55**	602.26	501.71	344.32	243.77	297.03	196.48

Table 6: Tricycle's End-to-end Latencies and Comparison with prior work for 128 tokens in seconds. The rightmost columns report the latency of a HEaaN-based [20] implementation of Thor [25]; all other latencies are reported using the respective works' original library implementations.

Optimizations	Latency (seconds)	
	Q, K, V	QK^T
Bicyclic Encoding	438.08	25.95
Tricyclic Encoding	103.28	4.36
+ BSGS	8.29	2.89
+ Lazy Relinearization	–	1.58
+ Optimized Block Matmuls	3.12	0.80
+ Complexification	1.70	0.25

Table 7: Ablation study of Tricycle's optimizations on a single layer of Q, K, and V (excluding reshaping) and QK^T using BERT-Base dimensions with 128 tokens.

Model	BERT-Tiny	BERT-Base
L_1 **Error**	0.017	0.040

Table 8: L_1 error between a Tricycle and plaintext model.

$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$. Meanwhile, tanh is defined as $\tanh(x) = (e^x - e^{-x})/(e^x + e^{-x})$. To evaluate both GELU and tanh efficiently, we use Chebyshev approximations [5], similar to the system by Rovida *et al.* [30].

8 Experimental Evaluation

We evaluate our system by developing an end-to-end GPU implementation of Tricycle on both BERT-Tiny [19] and BERT-Base [11]. A full description of the Tricycle protocol can be found in Appendix H. Our evaluation reports accuracy and latency to measure both output fidelity and system performance. We used HEaaN (generation 1, version 0.3.0) [20], which provides the underlying cryptographic primitives for CKKS on GPU, configured with 128-bit security [1]. Our codebase also implements several other optimizations, including:

- More efficient packing with greater slot utilization and complex numbers to minimize bootstrapping.
- Reordered multiplications and aggressively folded constants to minimize multiplicative depth and number of operations.

We fine-tuned pretrained BERT-Tiny [19] and BERT-Base [11] models on the SST-2 [31] dataset, and used those weights in Tricycle's evaluation. We ran all the experiments on a single NVIDIA A100 GPU with 80 GB of memory, which is the same as the prior works of Powerformer [27] and Thor [25]. For an additional comparison, we also developed a Thor implementation in HEaaN [20], incorporating several HEaaN-specific optimizations to ensure a representative evaluation.

Performance: Table 6 reports end-to-end latencies for Tricycle on BERT-Tiny and BERT-Base with 128 tokens, alongside comparisons to Thor (on Liberate.FHE [10] and HEaaN [20]) and Powerformer (on Liberate.FHE [10]) evaluated on BERT-Base. Tricycle achieves 4.39 seconds for BERT-Tiny and 100.55 seconds for BERT-Base, yielding a 6× speedup over Thor (602 seconds) [25] and a 3.44× speedup over Powerformer (344 seconds) [27]. It also outperforms ELLMo [13], which reports a BERT-Tiny latency of 8.19 seconds and the batching-based approach of MOAI [37], which reports an amortized latency of 141 seconds, both measured on a more powerful, single H200 GPU. Even when compared to a HEaaN-based implementation of Thor (297.03 seconds) on BERT-Base, Tricycle is still 2.95× more efficient. We note that Thor's ciphertext–ciphertext matrix multiplications (e.g. $Q \times K^T$ and the softmax result multiplied by V) perform worse on HEaaN than Liberate.FHE [10], primarily because Thor's Liberate.FHE implementation uses library optimizations that are not available in HEaaN. Even against Thor's best observed latency across both implementations (183.87 seconds total), Tricycle remains 1.82× faster.

	Ciphertext Level			
Model	**3**	**6**	**9**	**12**
BERT-Tiny ($d_{\text{model}} = 128$)	0.10	0.11	0.12	0.14
BERT-Base ($d_{\text{model}} = 768$)	0.86	0.89	0.90	0.95
Llama-2-7B ($d_{\text{model}} = 4096$)	7.34	8.20	9.75	12.09
Llama-2-13B ($d_{\text{model}} = 5120$)	9.22	10.49	12.66	16.05

Table 9: Scalability study of the latency (in seconds) required to evaluate a single layer of Q, K, and V (excluding reshaping) over 32 tokens for models of different sizes. The ciphertext level refers to the remaining multiplicative depth of the ciphertexts at the beginning of the computation. d_{model} is the model's embedding dimension.

	Ciphertext Level			
Model	**3**	**6**	**9**	**12**
BERT-Tiny ($d_{\text{model}} = 128$)	0.02	0.03	0.03	0.04
BERT-Base ($d_{\text{model}} = 768$)	0.02	0.03	0.03	0.05
Llama-2-7B ($d_{\text{model}} = 4096$)	0.48	0.68	0.79	1.03
Llama-2-13B ($d_{\text{model}} = 5120$)	0.62	0.82	0.99	1.30

Table 10: Scalability study of the latency (in seconds) required to evaluate QK^T in a single layer over 32 tokens for models of different sizes. The ciphertext level refers to the remaining multiplicative depth of the ciphertexts at the beginning of the computation. d_{model} is the model's embedding dimension.

Tricycle's performance gains arise from several complementary factors: (a) algorithmic optimizations that reduce the total number of operations, (b) systems-level enhancements that improve GPU utilization, (c) the improved efficiency of HEaaN's implementation (particularly bootstrapping), and (d) reduced multiplicative depth, which directly lowers the frequency of bootstrapping. HEaaN's library-level improvements account for a portion of the performance improvement over Thor and Powerformer. This is reflected by the improved performance of the HEaaN-based implementation of Thor, particularly for bootstrapping. Nevertheless, Tricycle's tricyclic encoding abstractions, matrix multiplication optimizations, and systems-level optimizations are necessary to effectively exploit these improvements. For example, the ablation studies in Table 7 show successive reductions in matrix multiplication latency as optimizations are introduced, demonstrating significant performance gains. Hence, the end-to-end gains reflect both advances in HEaaN's implementation and Tricycle's improvements.

Bootstrapping: In HEaaN, bootstrapping is required after 9 levels of multiplicative depth. For BERT-Base with 128 tokens, Tricycle requires 16 bootstrapping operations per transformer layer: 6 within softmax, 2 after attention, 2 after the first layer normalization, 2 before and 2 after the second layer normalization, and 1 within each layer normalization. In contrast, Thor's original implementation, despite using Liberate.FHE [10] configured to support up to 13 levels before bootstrapping, still requires 25.5 bootstraps per layer. For the HEaaN-based implementation of Thor, additional bootstrapping operations required by HEaaN were added beyond those in the original Thor implementation, resulting in approximately 54 bootstraps per layer. If configured with 11 available levels, Tricycle can eliminate an additional 4 bootstrapping operations per transformer layer: 2 in softmax and 1 in each layer normalization.

Accuracy: Table 8 reports the L_1 error (excluding a few obvious decryption failures from bugs) between Tricycle and the corresponding plaintext models over strings from SST-2, indicating reasonably low approximation error. Examples of input and output from a Tricycle model can be found in Appendix J, showing that the error is often small enough to avoid affecting the prediction.

Scalability: Lastly, we conduct a scalability study to evaluate whether Tricycle's matrix multiplication techniques extend to larger models, including Llama-2-7B and Llama-2-13B [32]. We conducted benchmarks of Tricycle's matrix multiplications (including all proposed matrix multiplication optimizations) across varying model sizes using a sequence length of 32 tokens. Matrix multiplication latency provides a useful measure of scalability because it is particularly sensitive to the model dimension, d_{model}. Tricycle's approach for the remaining operations, including the nonlinear layers, can scale to larger models. The matrix multiplication latencies are shown in Table 9 and Table 10. They demonstrate that Tricycle's matrix multiplication methods remain applicable to these larger models, but incurs higher costs, particularly for ciphertext–plaintext matrix multiplication. The runtime grows superlinearly with d_{model} because larger matrices must be partitioned across an increasing number of ciphertexts, while the amount of computation required along the d_{model} dimension also increases.

9 Conclusion

We presented Tricycle, a system for enabling private inference of transformer models under homomorphic encryption. Tricycle overcomes key challenges in private LLM inference by introducing the tricyclic encoding and proposing numerous optimizations for matrix multiplications, including BSGS, lazy relinearization, optimized block matrix multiplications, complexification, various systems optimizations, and more. We further propose a statistical max estimation technique stabilizing softmax, which has already seen some adoption in follow-up works [13, 34]. We demonstrate a significant performance improvement over prior work, which marks a step towards practical private LLM systems.

Acknowledgments

This work was supported in part by a Google PSS Privacy Research Faculty Award and the Air Force Research Laboratory / Information Directorate in Rome, NY.

References

[1] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. 2019. Homomorphic Encryption Standard. Cryptology ePrint Archive, Paper 2019/939. https://eprint.iacr.org/2019/939

[2] Ahmad Al Badawi, Andreea Alexandru, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, Zeyu Liu, Daniele Micciancio, Carlo Pascoe, Yuriy Polyakov, Ian Quah, Saraswathy R.V., Kurt Rohloff, Jonathan Saylor, Dmitriy Suponitsky, Matthew Triplett, Vinod Vaikuntanathan, and Vincent Zucca. 2022. OpenFHE: Open-Source Fully Homomorphic Encryption Library. Cryptology ePrint Archive, Paper 2022/915. https://eprint.iacr.org/2022/915 https://eprint.iacr.org/2022/915.

[3] Jean-Philippe Bossuat, Christian Mouchet, Juan Troncoso-Pastoriza, and Jean-Pierre Hubaux. 2021. Efficient Bootstrapping for Approximate Homomorphic Encryption with Non-sparse Keys. In *Advances in Cryptology – EUROCRYPT 2021*, Anne Canteaut and François-Xavier Standaert (Eds.). Springer International Publishing, Cham, 587–617.

[4] Jingwei Chen, Linhan Yang, Wenyuan Wu, Yang Liu, and Yong Feng. 2025. Homomorphic Matrix Operations Under Bicyclic Encoding. *IEEE Transactions on Information Forensics and Security* 20 (2025), 1390–1404. doi:10.1109/TIFS.2024.3490862

[5] E. W. Cheney. 1966. *Introduction to approximation theory.* McGraw-Hill Book Co., New York.

[6] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. 2018. Bootstrapping for Approximate Homomorphic Encryption. In *Advances in Cryptology – EUROCRYPT 2018*, Jesper Buus Nielsen and Vincent Rijmen (Eds.). Springer International Publishing, Cham, 360–384.

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

[8] Wonhee Cho, Guillaume Hanrot, Taeseong Kim, Minje Park, and Damien Stehlé. 2024. Fast and Accurate Homomorphic Softmax Evaluation. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) *(CCS '24)*. Association for Computing Machinery, New York, NY, USA, 4391–4404. doi:10.1145/3658644.3670369

[9] Leo de Castro, Antigoni Polychroniadou, and Daniel Escudero. 2024. Privacy-Preserving Large Language Model Inference via GPU-Accelerated Fully Homomorphic Encryption. In *Neurips Safe Generative AI Workshop 2024*.

[10] DESILO. 2023. Liberate.FHE: A New FHE Library for Bridging the Gap between Theory and Practice with a Focus on Performance and Accuracy. https://github.com/Desilo/liberate-fhe.

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. doi:10.18653/v1/N19-1423

[12] Xinwen Gao, Shaojing Fu, Lin Liu, Zhuotao Liu, Yuchuan Luo, and Yongjun Wang. 2026. Euston: Efficient and User-Friendly Secure Transformer Inference with Non-Interactivity . In *2026 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 1443–1462. doi:10.1109/SP63933.2026.00048

[13] Seyda Nur Guzelhan, Lohit Daksha, Carlos Agulló Domingo, Gilbert Jonatan, John Kim, Jose L. Abellan, David Kaeli, and Ajay Joshi. 2026. ELLMo: Packing- and Depth-Aware Encrypted Transformer Inference. Cryptology ePrint Archive, Paper 2026/198. https://eprint.iacr.org/2026/198

[14] Shai Halevi and Victor Shoup. 2014. Algorithms in helib. In *Advances in Cryptology–CRYPTO 2014: 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I 34*. Springer, 554–571.

[15] Shai Halevi and Victor Shoup. 2018. Faster Homomorphic Linear Transformations in HElib. In *Advances in Cryptology – CRYPTO 2018: 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2018, Proceedings, Part I* (Santa Barbara, CA, USA). Springer-Verlag, Berlin, Heidelberg, 93–120. doi:10.1007/978-3-319-96884-1_4

[16] Meng Hao, Hongwei Li, Hanxiao Chen, Pengzhi Xing, Guowen Xu, and Tianwei Zhang. 2022. Iron: private inference on transformers. In *Proceedings of the 36th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '22)*. Curran Associates Inc., Red Hook, NY, USA, Article 1143, 14 pages.

[17] Xiaoyang Hou, Jian Liu, Jingyu Li, Yuhan Li, Jiawen Zhang, Wen-jie Lu, Cheng Hong, and Kui Ren. 2026. CipherGPTCipherGPT: Secure Two-Party GPT Inference. *IEEE Transactions on Dependable and Secure Computing* 23, 3 (2026), 6748–6762. doi:10.1109/TDSC.2026.3667722

[18] Xiaoqian Jiang, Miran Kim, Kristin Lauter, and Yongsoo Song. 2018. Secure Outsourced Matrix Computation and Application to Neural Networks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 1209–1222. doi:10.1145/3243734.3243837

[19] Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. 2020. TinyBERT: Distilling BERT for Natural Language Understanding. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, Online, 4163–4174. doi:10.18653/v1/2020.findings-emnlp.372

[20] Wonkyung Jung, Eojin Lee, Sangpyo Kim, Jongmin Kim, Namhoon Kim, Keewoo Lee, Chohong Min, Jung Hee Cheon, and Jung Ho Ahn. 2021. Accelerating fully homomorphic encryption through architecture-centric analysis and optimization. *IEEE Access* 9 (2021), 98772–98789.

[21] Gautam Kamath. 2015. Bounds on the expectation of the maximum of samples from a gaussian. http://www.gautamkamath.com/writings/gaussian_max.pdf

[22] Yang Li, Xinyu Zhou, Yitong Wang, Liangxin Qian, and Jun Zhao. 2024. A Survey on Private Transformer Inference. arXiv:2412.08145 [cs.CR] https://arxiv.org/abs/2412.08145

[23] Zhengyi Li, Kang Yang, Jin Tan, Wen-jie Lu, Haoqi Wu, Xiao Wang, Yu Yu, Derun Zhao, Yancheng Zheng, Minyi Guo, et al. 2024. Nimbus: Secure and Efficient Two-Party Inference for Transformers. *Advances in Neural Information Processing Systems* 37 (2024), 21572–21600.

[24] Wen-jie Lu, Zhicong Huang, Zhen Gu, Jingyu Li, Jian Liu, Cheng Hong, Kui Ren, Tao Wei, and WenGuang Chen. 2023. Bumblebee: Secure two-party inference framework for large transformers. *Cryptology ePrint Archive* (2023).

[25] Jungho Moon, Dongwoo Yoo, Xiaoqian Jiang, and Miran Kim. 2025. THOR: Secure Transformer Inference with Homomorphic Encryption. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security* (Taipei, Taiwan) *(CCS '25)*. Association for Computing Machinery, New York, NY, USA, 3765–3779. doi:10.1145/3719027.3765150

[26] Qi Pang, Jinhao Zhu, Helen Möllering, Wenting Zheng, and Thomas Schneider. 2024. Bolt: Privacy-preserving, accurate and efficient inference for transformers. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 4753–4771.

[27] Dongjin Park, Eunsang Lee, and Joon-Woo Lee. 2025. Powerformer: Efficient and High-Accuracy Privacy-Preserving Language Model with Homomorphic Encryption. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (Eds.). Association for Computational Linguistics, Vienna, Austria, 11090–11111. doi:10.18653/v1/2025.acl-long.543

[28] Jaiyoung Park, Sejin Park, Jai Hyun Park, Jung Ho Ahn, Jung Hee Cheon, Guillaume Hanrot, Jung Woo Kim, Minje Park, and Damien Stehlé. 2026. Scaling up Privacy-Preserving ML: A CKKS Implementation of Llama-2-7B. arXiv:2601.18511 [cs.CR] https://arxiv.org/abs/2601.18511

[29] Donghwan Rho, Taeseong Kim, Minje Park, Jung Woo Kim, Hyunsik Chae, Ernest Ryu, and Jung Hee Cheon. 2025. Encryption-Friendly LLM Architecture. In *International Conference on Learning Representations*, Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (Eds.), Vol. 2025. 41638–41664. https://proceedings.iclr.cc/paper_files/paper/2025/file/6715b4e97be055687c1ecaf33913d358-Paper-Conference.pdf

[30] Lorenzo Rovida and Alberto Leporati. 2024. Transformer-based Language Models and Homomorphic Encryption: An Intersection with BERT-tiny. In *Proceedings of the 10th ACM International Workshop on Security and Privacy Analytics* (Porto, Portugal) *(IWSPA '24)*. Association for Computing Machinery, New York, NY, USA, 3–13. doi:10.1145/3643651.3659893

[31] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Ng, and Christopher Potts. 2013. Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Seattle, Washington, USA, 1631–1642. https://www.aclweb.org/anthology/D13-1170

[32] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288 [cs.CL] https://arxiv.org/abs/2307.09288

[33] Zihao Wang, Rongmao Chen, Xinwen Gao, Yi Wang, Lin Liu, Zixin Lan, Zhaoyu Wang, Shaojing Fu, Qiong Wang, and Xinyi Huang. 2026. STIP: Efficient and Secure Non-Interactive Transformer Inference via Compact Packing. Cryptology ePrint Archive, Paper 2026/174. https://eprint.iacr.org/2026/174

[34] Linhan Yang, Jingwei Chen, Wangchen Dai, Shuai Wang, Wenyuan Wu, and Yong Feng. 2025. ARION: Attention-Optimized Transformer Inference on Encrypted Data. Cryptology ePrint Archive, Paper 2025/2271. https://eprint.iacr.org/2025/2271

[35] Wenshao Yang, Zhenhua Liu, Yihan Zhang, Xin Bie, and Dongdong Yao. 2026. SecretVoice1.0: Non-interactive Secure Transformer Inference. *TechRxiv* 2026, 0227 (2026). arXiv:https://www.techrxiv.org/doi/pdf/10.36227/techrxiv.177220349.97070665/v1 doi:10.36227/techrxiv.177220349.97070665/v1

[36] Jiawen Zhang, Xinpeng Yang, Lipeng He, Kejia Chen, Wen-jie Lu, Yinghao Wang, Xiaoyang Hou, Jian Liu, Kui Ren, and Xiaohu Yang. 2025. Secure Transformer Inference Made Non-interactive. In *32nd Annual Network and Distributed System Security Symposium, NDSS 2025, San Diego, California, USA, February 24-28, 2025*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/secure-transformer-inference-made-non-interactive/

[37] Linru Zhang, Xiangning Wang, Jun Jie Sim, Zhicong Huang, Jiahao Zhong, Huaxiong Wang, Pu Duan, and Kwok Yan Lam. 2026. MOAI: Module-Optimizing Architecture for Non-Interactive Secure Transformer Inference. In *International Conference on Learning Representations*, C. Vondrick, B. Hariharan, C. Raffel, L. Pinto, D. Yang, and A. Faust (Eds.), Vol. 2026. 27927–27953. https://proceedings.iclr.cc/paper_files/paper/2026/file/2f2b1d6bbd50865eca40e2774a057eef-Paper-Conference.pdf

[38] Mengxin Zheng, Qian Lou, and Lei Jiang. 2023. Primer: Fast private transformer inference on encrypted data. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

Generative AI Usage

The HEaaN-based [20] implementation of Thor [25] was developed with assistance from OpenAI's Codex. We manually verified the resulting code for consistency with Thor. This paper was also edited for grammar with ChatGPT.

Open Science

The Tricycle implementation is available at https://bitbucket.org/tricyclepaper/tricycle. An OpenFHE (CPU) implementation of our matrix multiplications is available at https://bitbucket.org/tricyclepaper/tricyclic-matmul.

A Chinese Remainder Theorem

Let $m, n \in \mathbb{Z}_{>0}$ such that $\gcd(m, n) = 1$, and let $a, b \in \mathbb{Z}$. The system of congruences

$$\begin{cases} x \equiv a \bmod m \\ x \equiv b \bmod n \end{cases}$$

has a unique solution modulo mn which can be explicitly constructed as

$$x = (a + m \cdot (m^{-1} \bmod n) \cdot (b - a)) \bmod (mn).$$

$(m^{-1} \bmod n)$ is an inverse modulo n such that $m \cdot m^{-1} \equiv 1 \bmod n$. Symmetrically, x can be constructed as

$$x = (b + n \cdot (n^{-1} \bmod m) \cdot (a - b)) \bmod (mn).$$

The Chinese Remainder Theorem can generalize to systems of three or more congruences. The system of n congruences

$$\begin{cases} x \equiv r_1 \bmod m_1 \\ x \equiv r_2 \bmod m_2 \\ \quad \cdots \\ x \equiv r_n \bmod m_n \end{cases}$$

with $\mathrm{lcm}(m_1, m_2, \ldots, m_n) = \prod_{i=1}^{n} m_i$ (e.g. pairwise coprime) has a unique solution modulo M where $M = \prod_{i=1}^{n} m_i$, which can be expressed as

$$x \bmod M = \left(\sum_{i=1}^{n} r_i \cdot \frac{M}{m_i} \cdot \left(\left(\frac{M}{m_i} \right)^{-1} \bmod m_i \right) \right) \bmod M$$

where M/m_i is the product of all $m_1, m_2, ..., m_n$ except m_i.

B Free Transpose Theorem

We prove that for any n-cyclic encoding of an n-dimensional tensor, all transposes are provided for free.

Definition 1: Let $d_1, d_2, ..., d_n \in \mathbb{N}$. An n-dimensional tensor is defined by

$$A = \left[a_{i_1, i_2, ..., i_n} \right] \in \mathbb{R}^{d_1 \times d_2 \times ... \times d_n}.$$

Definition 2: Assume that $\mathrm{lcm}(d_1, d_2, \ldots, d_n) = \prod_{i=1}^{n} d_i$. The n-cyclic encoding of A is defined by the mapping

$$\varphi \colon \mathbb{R}^{d_1 \times d_2 \times ... \times d_n} \longrightarrow \mathbb{R}^{\prod_{i=1}^{n} d_i},$$

given by

$$\varphi(A)_k = a_{k \bmod d_1, k \bmod d_2, ..., k \bmod d_n},$$

for all $k \in \mathbb{Z}_{\prod_{i=1}^{n} d_i}$.

Definition 3: A transpose of A is defined by choosing a new ordering of its indices. That is, if we reassign the indices according to

$$\tau : (1, 2, ..., n) \longrightarrow (a_1, a_2, ..., a_n),$$

where $(a_1, a_2, \ldots, a_n)$ is a permutation of $(1, 2, \ldots, n)$ then the transposed tensor A^τ has components

$$\begin{aligned} A^\tau &= \left[a^\tau_{i_1, i_2, ..., i_n} \right] \\ &= \left[a_{i_{\tau(1)}, i_{\tau(2)}, ..., i_{\tau(n)}} \right] \in \mathbb{R}^{d_{\tau(1)} \times d_{\tau(2)} \times ... \times d_{\tau(n)}}. \end{aligned} \tag{19}$$

The inverse of the transpose is defined as

$$\begin{aligned} A = (A^\tau)^{\tau^{-1}} = \left[a_{i_1, i_2, ..., i_n} \right] &= \left[a^{\tau^{-1}}_{i_{\tau(1)}, i_{\tau(2)}, ..., i_{\tau(n)}} \right] \\ &= \left[a^{\tau}_{i_{\tau^{-1}(1)}, i_{\tau^{-1}(2)}, ..., i_{\tau^{-1}(n)}} \right] \\ &= \left[a_{i_{\tau^{-1}(\tau(1))}, i_{\tau^{-1}(\tau(2))}, ..., i_{\tau^{-1}(\tau(n))}} \right]. \end{aligned} \tag{20}$$

Theorem: For an n-dimensional tensor A with dimensions $(d_1 \times d_2 \times \ldots \times d_n)$ and for any transpose A^τ defined by a permutation $(a_1, a_2, \ldots, a_n)$, then $\varphi(A^\tau) = \varphi(A)$.

Starting with the definition of n-cyclic encoding of A^τ, we get

$$\varphi(A^\tau)_k = a^\tau_{k \bmod d_{\tau(1)}, k \bmod d_{\tau(2)}, ..., k \bmod d_{\tau(n)}}, \quad \text{for all } k \in \mathbb{Z}_{\prod_{i=1}^{n} d_i}. \tag{21}$$

Using the inverse of the transpose, we get

$$= a_{k \bmod d_1, k \bmod d_2, ..., k \bmod d_n} = \varphi(A)_k \tag{22}$$

which proves the theorem.

C Matrix Multiplication Algorithm Proofs

C.1 Batch Matrix Multiplication

For an $(h \times m \times n)$ tensor A and an $(h \times n \times p)$ tensor B, the batch matrix multiplication is expressed as

$$Z_{i,j,k} = \left[\sum_{c=0}^{n-1} a_{i,j,c} \cdot b_{i,c,k} \right] \in \mathbb{R}^{h \times m \times p}. \tag{23}$$

Applying the definition of tricyclic encoding to Z, we get

$$\varphi(Z)_k = \sum_{c=0}^{n-1} \left(a_{k \bmod h,\, k \bmod m,\, (k+c) \bmod n} \cdot b_{k \bmod h,\, (k+c) \bmod n,\, k \bmod p} \right). \tag{24}$$

Using CRT, we can determine that $x = k + chm((hm)^{-1} \bmod n) \bmod (hmn)$ is a solution for the congruence $x \equiv k \bmod h$, $x \equiv k \bmod m$, and $x \equiv k + c \bmod n$ (see Appendix A). Using this solution for x helps us obtain the following:

$$= \sum_{c=0}^{n-1} \left(a_{x \bmod h,\, x \bmod m,\, x \bmod n} \cdot b_{k \bmod h,\, (k+c) \bmod n,\, k \bmod p} \right). \tag{25}$$

Note that Equation 25 simplifies to Equation 24, proving that the two are equivalent. Next, we can apply the definition of the tricyclic encoding to A and obtain

$$= \sum_{c=0}^{n-1} \left(\varphi(A)_{(k+chm((hm)^{-1} \bmod n)) \bmod (hmn)} \odot b_{k \bmod h,\, (k+c) \bmod n,\, k \bmod p} \right). \tag{26}$$

Likewise, we can apply the same steps to B and get

$$= \sum_{c=0}^{n-1} \left(\varphi(A)_{(k+chm((hm)^{-1} \bmod n)) \bmod (hmn)} \odot \varphi(B)_{(k+chp((hp)^{-1} \bmod n)) \bmod (hnp)} \right). \tag{27}$$

Finally, assuming sufficient duplication of $\varphi(A)$ and $\varphi(B)$ and rotational correctness, we can simplify to

$$\varphi(Z) = \sum_{c=0}^{n-1} \left(\rho\left(\varphi(A), chm((hm)^{-1} \bmod n)\right) \bmod (hmn) \right) \odot \rho\left(\varphi(B), (chp((hp)^{-1} \bmod n)) \bmod (hnp)\right)\Big). \tag{28}$$

Here, we show how to derive the diagonal traversal for ciphertext–plaintext matrix multiplication. Recall that the batch matrix multiplication $Z = A \cdot B$ from Equation 2 allows us to simplify to Equation 26 as follows:

$$\varphi(Z)_k = \sum_{c=0}^{n-1} \left(\varphi(A)_{(k+chm((hm)^{-1} \bmod n)) \bmod (hmn)} \odot b_{k \bmod h,\, (k+c) \bmod n,\, k \bmod p} \right) \tag{29}$$

Assuming sufficient duplication of $\varphi(A)$ and rotational correctness, we can simplify to

$$\varphi(Z) = \sum_{c=0}^{n-1} \left(\rho(\varphi(A), (chm((hm)^{-1} \bmod n)) \bmod (hmn)) \odot \text{BatchDiag}(B, c) \right) \tag{30}$$

where $\text{BatchDiag}(B, c)_k = b_{k \bmod h,\, (k+c) \bmod n,\, k \bmod p}$ for $k \in \mathbb{Z}_{hmp}$.

C.2 Broadcasted Batch Matrix Multiplication

For a $(1 \times m \times n)$ tensor A and an $(h \times n \times p)$ tensor B, the broadcasted batch matrix multiplication is expressed as

$$C_{i,j,k} = \left[\sum_{c=0}^{n-1} a_{0,j,c} \cdot b_{i,c,k} \right] \in \mathbb{R}^{h \times m \times p}. \tag{31}$$

Applying the tricyclic encoding, we get

$$\varphi(Z)_k = \sum_{c=0}^{n-1} \left(a_{0,\, k \bmod m,\, (k+c) \bmod n} \cdot b_{k \bmod h,\, (k+c) \bmod n,\, k \bmod p} \right). \tag{32}$$

By applying the Chinese Remainder Theorem and definition of tricyclic encoding, we get

$$= \sum_{c=0}^{m-1} \left(\varphi(A)_{(k+cm(m^{-1} \bmod n)) \bmod (mn)} \odot \varphi(B)_{(k+chp((hp)^{-1} \bmod n)) \bmod (hnp)} \right). \tag{33}$$

Finally, converting to rotations, we get

$$\varphi(Z) = \sum_{c=0}^{n-1} \left(\rho\left(\varphi(A), cm(m^{-1} \bmod n)\right) \bmod (mn) \right) \odot \rho\left(\varphi(B), (chp((hp)^{-1} \bmod n)) \bmod (hnp)\right)\Big). \tag{34}$$

This shows that we can compute the broadcasted batch matrix multiplication by computing rotations of every $m(m^{-1} \bmod n)$ and $hp((hp)^{-1} \bmod n)$ for $\varphi(A)$ and $\varphi(B)$, respectively, and summing their product.

To show the ciphertext–plaintext broadcasted batch matrix multiplication between a ciphertext A and plaintext B, we can start with Equation 32 and simplify A to

$$= \sum_{c=0}^{m-1} \left(\varphi(A)_{(k+cm(m^{-1} \bmod n)) \bmod (mn)} \odot b_{k \bmod h,\, (k+c) \bmod n,\, k \bmod p} \right). \tag{35}$$

A can now be converted rotations and B to a plaintext vector as follows:

$$\varphi(Z) = \sum_{c=0}^{n-1} \left(\rho\left(\varphi(A), cm(m^{-1} \bmod n)\right) \bmod (mn) \right) \odot \text{BroadcastDiag(B,c)}) \tag{36}$$

where $\text{BroadcastDiag}(B, c) = b_{k \bmod h,\, (k+c) \bmod n,\, k \bmod p}$.

C.3 Concatenated Matrix Multiplication

For an $(h \times m \times n)$ ciphertext tensor A and an $(h \times n \times p)$ plaintext tensor B, the concatenated matrix multiplication is expressed as

$$C_{0,j,k} = \left[\sum_{d=0}^{h-1} \sum_{c=0}^{n-1} a_{d,j,c} \cdot b_{d,c,k} \right] \in \mathbb{R}^{1 \times m \times p}. \tag{37}$$

We can factor out the inner summation into C' as follows:

$$C_{0,j,k} = \sum_{d=0}^{h-1} C'_{d,j,k}$$

$$\text{where } C'_{d,j,k} = \left[\sum_{c=0}^{n-1} a_{d,j,c} \cdot b_{d,c,k} \right] \in \mathbb{R}^{h \times m \times p} \tag{38}$$

First, we will simplify C'. Note that the equation for C' is exactly the same as a batch matrix multiplication. Using a similar process, we get

$$\varphi(C')_{k'} = \sum_{c=0}^{n-1} \big(a_{k' \bmod h, k' \bmod m, (k'+c) \bmod n} \cdot b_{k' \bmod h, (k'+c) \bmod n, k' \bmod p} \big). \tag{39}$$

We remark that the d, j, and k in $C'_{d,j,k}$ became substituted for $k' \bmod h$, $k' \bmod m$, and $k' \bmod p$, respectively.

We return to simplifying C. We first apply the definition of tricyclic encoding to C.

$$\varphi(Z)_k = \sum_{d=0}^{h-1} c'_{d,\, k \bmod m, k \bmod p} \tag{40}$$

$$= \sum_{d=0}^{h-1} c'_{(k+d) \bmod h, k \bmod m, k \bmod p}. \tag{41}$$

This means that to calculate the k-th value of $\varphi(Z)$, we want to sum up values from C'. These values are obtained from the $((k + d) \bmod h)$-th matrix of C', the $(k \bmod m)$-th row of that matrix, and $(k \bmod p)$-th column of that row. We can now substitute C' from Equation 39 back into Equation 40. We can substitute $(k+d) \bmod h$ in for the first dimension of $C'_{d,j,k}$ and likewise for the rest of the dimensions as follows

$$\varphi(Z)_k = \sum_{d=0}^{h-1} \sum_{c=0}^{n-1} \big(a_{(k+d) \bmod h, k \bmod m, (k+c) \bmod n} \cdot b_{(k+d) \bmod h, (k+c) \bmod n, k \bmod p} \big). \tag{42}$$

Using CRT, we can find that $x = (d \cdot (mn) \cdot ((mn)^{-1} \bmod h) + c \cdot (hm) \cdot ((hm)^{-1} \bmod n)) \bmod (hmn)$ is a solution to the congruence $x \equiv d \bmod h$, $x \equiv 0 \bmod m$, and $x \equiv c \bmod n$. This means we can simplify to

$$\varphi(Z)_k = \sum_{d=0}^{h-1} \sum_{c=0}^{n-1} \big(a_{(k+x) \bmod h, (k+x) \bmod m, (k+x) \bmod n} \cdot b_{(k+d) \bmod h, (k+c) \bmod n, k \bmod p} \big). \tag{43}$$

Now, we apply the tricyclic encoding to A and simplify to

$$\varphi(Z)_k = \sum_{d=0}^{h-1} \sum_{c=0}^{n-1} \big(\varphi(A)_{(k+d \cdot r_1 + c \cdot r_2) \bmod (h \cdot m \cdot n)} \cdot b_{(k+d) \bmod h, (k+c) \bmod n, k \bmod p} \big)$$
$$\text{where } r_1 = m \cdot n \cdot ((m \cdot n)^{-1} \bmod h),$$
$$r_2 = h \cdot m \cdot ((h \cdot m)^{-1} \bmod n). \tag{44}$$

We can likewise take the same steps for B. Using CRT, we can find that $y = (d \cdot (np) \cdot ((np)^{-1} \bmod h) + c \cdot (hp) \cdot ((hp)^{-1} \bmod n)) \bmod (hnp)$ is a solution to the congruence $y \equiv d \bmod h$, $y \equiv c \bmod n$, and $y \equiv 0 \bmod p$. We use this and apply the definition of tricyclic encoding to simplify to

$$\varphi(Z)_k = \sum_{d=0}^{h-1} \sum_{c=0}^{n-1} \big(\varphi(A)_{(k+d \cdot r_1 + c \cdot r_2) \bmod (h \cdot m \cdot n)} \cdot \varphi(B)_{(k+y) \bmod (h \cdot n \cdot p)} \big). \tag{45}$$

We can finally convert to using rotations and show that the equation for a ciphertext–ciphertext matrix multiplication is as follows:

$$\varphi(Z) = \sum_{d=0}^{h-1} \sum_{c=0}^{n-1} (\rho(\varphi(A), (d \cdot r_1 + c \cdot r_2) \bmod (h \cdot m \cdot n)) \odot \rho(\varphi(B), (d \cdot r_3 + c \cdot r_4) \bmod (h \cdot n \cdot p)))$$
$$\text{where } r_1 = m \cdot n \cdot ((m \cdot n)^{-1} \bmod h),$$
$$r_2 = h \cdot m \cdot ((h \cdot m)^{-1} \bmod n),$$
$$r_3 = n \cdot p \cdot ((n \cdot p)^{-1} \bmod h),$$
$$r_4 = h \cdot p \cdot ((h \cdot p)^{-1} \bmod n). \tag{46}$$

Discussion on this equation can be found in Section 5.2.3.

For a ciphertext–plaintext concatenated matrix multiplication between a ciphertext A and plaintext B, we can derive the diagonals required by converting from Equation 44 to

$$\varphi(Z) = \sum_{d=0}^{h-1} \sum_{c=0}^{n-1} (\rho(\varphi(A), (d \cdot r_1 + c \cdot r_2) \bmod (h \cdot m \cdot n)) \odot \text{ConcatDiag}(B, c, d))$$
$$\text{where } \forall k \in \mathbb{Z}_{m \cdot p}, \text{ConcatDiag}(B, c, d)_k = b_{(k+d) \bmod h, (k+c) \bmod n, k \bmod p}. \tag{47}$$

d is a summation index that iterates over matrices in A and B. c is a summation index that iterates columns and rows in those matrices in A and B, respectively. r_1 represents the rotation amount required to obtain the next matrix in the $\varphi(A)$ while r_2 represents the rotation amount required to obtain the next column in the same matrix. Finally, $\text{ConcatDiag}(B, c, d)$ is the function to calculate the plaintext needed to multiply by the rotated ciphertext to obtain the correct output.

D Baby-Step Giant-Step Ciphertext–Plaintext Matrix Multiplications

We prove without loss of generality to broadcasted batch matrix multiplication and concatenated batch matrix multiplication that an $(h \times m \times n)$ ciphertext X multiplied by an $(h \times n \times p)$ plaintext Y under tricyclic encoding can apply BSGS by showing that it can be expressed in the form $\sum_{c=0}^{N-1} \rho(x, T \cdot c) \odot y$ with periodicity $T = h \cdot m$. Beginning with Equation 8, which shows ciphertext–plaintext batch matrix multiplications under tricyclic encoding, we have

$$\varphi(Z) = \sum_{c=0}^{n-1} \big(\rho(\varphi(X), (chm((hm)^{-1} \bmod n)) \bmod (hmn)) \odot \mathrm{BatchDiag}(Y, c)\big) \quad (48)$$

where $\mathrm{BatchDiag}(Y, c)_k = y_{k \bmod h,\, (k+c) \bmod n,\, k \bmod p}$ for $k \in \mathbb{Z}_{hmp}$. To prove that BSGS is applicable, we can reindex c through a series of steps as follows:

$$= \sum_{c=0}^{n-1} \rho(\varphi(X), (chm((hm)^{-1} \bmod n)) \bmod (hmn)) \odot \mathrm{BatchDiag}(Y, c) \quad (49)$$

$$= \sum_{c=0}^{n-1} \big(\rho(\varphi(X), ((c \bmod n) \cdot ((hm)^{-1} \bmod n) \cdot hm) \bmod (hmn)) \odot \mathrm{BatchDiag}(Y, c \bmod n)\big) \quad (50)$$

$$= \sum_{c=0}^{n-1} \big(\rho(\varphi(X), ((c \cdot (hm)^{-1} \bmod n) \cdot hm) \bmod (hmn)) \odot \mathrm{BatchDiag}(Y, c \cdot (hm)^{-1} \cdot hm \bmod n)\big) \quad (51)$$

Since h, m, and n are coprime, multiplying each element of $\{0, 1, \ldots, n-1\}$ by hm or its inverse modulo n simply permutes the set, obtaining the same numbers in a different order. This allows us to reindex c to $c' = (c \cdot (hm)^{-1} \bmod n)$, which becomes

$$= \sum_{c'=0}^{n-1} (\rho(\varphi(X), (c' \cdot hm) \bmod (hmn)) \odot \mathrm{BatchDiag}(Y, (c' \cdot hm) \bmod n)) \,. \quad (52)$$

$$= \sum_{c'=0}^{n-1} (\rho(\varphi(X), c' \cdot hm) \odot \mathrm{BatchDiag}'(Y, c')) \quad (53)$$

where $\mathrm{BatchDiag}'(Y, c')_k = y_{k \bmod h,\, (k+c' \cdot hm) \bmod n,\, k \bmod p}$ for $k \in \mathbb{Z}_{hmp}$. This simplification means that BSGS can be applied for $x = \varphi(A)$, $y = \mathrm{BatchDiag}'(Y, c)$, and periodicity $T = hm$.

E Baby-Step Giant-Step For Ciphertext–Ciphertext Matrix Multiplications

We prove without loss of generality to broadcasted batch matrix multiplication and concatenated batch matrix multiplication that an $(h \times m \times n)$ ciphertext A multiplied by an $(h \times n \times p)$ ciphertext B under tricyclic encoding can apply BSGS by showing that it can be expressed in the form $\sum_{c=0}^{N-1} \rho(x, T \cdot c) \odot y$ with periodicity $T = h \cdot m$. Beginning with Equation 28, which shows ciphertext–plaintext batch matrix multiplications under tricyclic encoding, we have

$$\varphi(Z) = \sum_{c=0}^{n-1} \big(\rho\left(\varphi(A), chm((hm)^{-1} \bmod n)) \bmod (hmn)\right) \odot \rho\left(\varphi(B), (chp((hp)^{-1} \bmod n)) \bmod (hnp)\right)\big) \,. \quad (54)$$

To prove that BSGS is applicable, we can reindex c through a series of steps as follows:

$$\varphi(Z) = \sum_{c=0}^{n-1} \big(\rho\left(\varphi(A), (c \bmod n) \cdot ((hm)^{-1} \bmod n)) \cdot hm \bmod (hmn)\right) \odot \rho\left(\varphi(B), (chp((hp)^{-1} \bmod n)) \bmod (hnp)\right)\big) \quad (55)$$

$$= \sum_{c=0}^{n-1} \big(\rho\left(\varphi(A), (c(hm)^{-1} \bmod n)) \cdot hm \bmod (hmn)\right) \odot \rho\left(\varphi(B), (chp((hp)^{-1} \bmod n)) \bmod (hnp)\right)\big) \quad (56)$$

$$= \sum_{c=0}^{n-1} \big(\rho\left(\varphi(A), (c(hm)^{-1} \bmod n)) \cdot hm \bmod (hmn)\right) \odot \rho\left(\varphi(B), ((c(hm)^{-1} \cdot hm \bmod n) \cdot hp((hp)^{-1} \bmod n)) \bmod (hnp)\right)\big) \quad (57)$$

Similar to applying BSGS to ciphertext–plaintext matrix multiplications, h, m, and n are coprime, so multiplying each element of $\{0, 1, ..., n-1\}$ by hm or its inverse modulo n simply permutes the set, obtaining the same numbers in a different order. This allows us to reindex c to $c' = (c \cdot (hm)^{-1} \bmod n)$, which becomes

$$= \sum_{c'=0}^{n-1} (\rho\left(\varphi(A), (c' \cdot hm \bmod (hmn)\right) \odot \rho\left(\varphi(B), ((c' \cdot hm \bmod n) \cdot hp((hp)^{-1} \bmod n)) \bmod (hnp)\right)\big) \quad (58)$$

$$= \sum_{c'=0}^{n-1} (\rho\left(\varphi(A), (c' \cdot hm \bmod (hmn)\right) \odot \rho\left(\varphi(B), (hp(c' \cdot hm(hp)^{-1} \bmod n)) \bmod (hnp)\right)\big) \quad (59)$$

$$\varphi(Z) = \sum_{c'=0}^{n-1} (\rho\left(\varphi(A), (c' \cdot hm \bmod (hmn)\right) \odot \rho\left(\varphi(B), (hp(c' \cdot m(p)^{-1} \bmod n)) \bmod (hnp)\right)\big) \,. \quad (60)$$

This simplification means that BSGS can be applied for $x = \varphi(A)$, $y = \varphi(B)$, periodicity $T = hm$, and $\mathrm{RotY}(c') = (hp(c' \cdot m(p)^{-1} \bmod n)) \bmod (hnp)$.

F Row-wise Summation under Tricyclic Encoding

Row-wise summations are used in softmax and layer normalization operations. In softmax, our statistical max estimation, described in Section 7.1, requires two row-wise summations for calculating the mean and variance and the softmax itself requires calculating a sum of exponentials, a third row-wise summation. Layer normalization requires two row-wise summations for mean and variance. The row-wise summation algorithms are described in Algorithms 1 and 2, where Algorithm 1 is used when only half or fewer of the ciphertext slots are used while Algorithm 2 is otherwise used.

Algorithm 1 Row-wise Summation under Tricyclic Encoding. This algorithm assumes that no more than half of available slots are used. $\lfloor \log_2(n) \rfloor + 3$ homomorphic rotations and additions are required.

Input: Ciphertext c encrypting a tricyclically encoded tensor with dimensions $(h \times m \times n)$ and zero padding.
Output: Ciphertext encrypting a tensor of the same shape containing row-wise sums, with no additional multiplicative depth.
Prerequisite: $h \cdot m \cdot n \leq$ (number of slots)/2

$p \leftarrow 2^{\lfloor \log_2(n) \rfloor}$
$r \leftarrow n - p$
sum $\leftarrow c$
for $i = 0$ to $\lfloor \log_2(n) \rfloor - 1$ **do**
    sum $\leftarrow$ sum + $\rho(\text{sum}, h \cdot m \cdot 2^i)$
    if $i + 1 = \lceil \log_2(r) \rceil$ **then**
        right_tail $\leftarrow$ sum
    end if
    if $i + 1 = \lceil \log_2(r-1) \rceil$ **then**
        left_tail $\leftarrow$ sum
    end if
end for
sum $\leftarrow$ sum + $\rho(\text{sum}, -h \cdot m \cdot p)$
if $r > 0$ **then**
    sum $\leftarrow$ sum + $\rho(\text{left_tail}, h \cdot m \cdot p)$
end if
if $r - 1 > 0$ **then**
    sum $\leftarrow$ sum + $\rho(\text{right_tail}, -h \cdot m \cdot \left(p + 2^{\lceil \log_2(r-1) \rceil}\right))$
end if
return sum

Algorithm 2 Row-wise Summation under Tricyclic Encoding. This is less efficient (requires more rotations and one extra multiplicative depth) than Algorithm 1, so it is only used when more than half of the available slots are used, e.g., in layer normalization.

Input: Ciphertext c encrypting a tricyclically encoded tensor of dimensions $(h \times m \times n)$.
Output: Ciphertext of dimensions $(h \times m \times n)$ containing row-wise sums of c, with zero padding and one additional multiplicative depth.

$\ell \leftarrow \lfloor \log_2(n) \rfloor + 1$
Initialize list U with $U[0] \leftarrow c$
for $i = 1$ to $\ell - 1$ **do**
    $U[i] \leftarrow U[i-1] + \rho(U[i-1], h \cdot m \cdot 2^{i-1})$
end for
$s \leftarrow U[\ell - 1]$
for $i = 0$ to $\ell - 2$ **do**
    if $\left\lfloor \frac{n}{2^i} \right\rfloor \bmod 2 = 1$ **then**
        $s \leftarrow s + \rho(U[i], h \cdot m \cdot (n - (n \bmod 2^{i+1})))$
    end if
end for
mask $\leftarrow$ Plaintext vector with 1 for $0 \leq i < h \cdot m$, else 0
Initialize list V with $V[0] \leftarrow s \odot$ mask
for $i = 1$ to $\ell - 1$ **do**
    $V[i] \leftarrow V[i-1] + \rho(V[i-1], -h \cdot m \cdot 2^{i-1})$
end for
$t \leftarrow V[\ell - 1]$
for $i = 0$ to $\ell - 2$ **do**
    if $\left\lfloor \frac{n}{2^i} \right\rfloor \bmod 2 = 1$ **then**
        $t \leftarrow \rho(t, -h \cdot m \cdot 2^i) + V[i]$
    end if
end for
return t

Layer	1	2	3	4	5	6	7	8	9	10	11	12
Skewness	0.80	1.07	0.92	0.91	0.68	0.60	0.52	0.47	0.45	0.29	0.59	0.19
Excess Kurtosis	2.94	2.11	11.00	1.55	1.07	0.82	0.67	0.40	0.45	0.26	1.60	-0.22

Table 11: Skewness and kurtosis of softmax inputs in BERT-Base across layers

G Statistical Max Estimation

We first show how to derive the statistical max estimation. Recall that our goal is to subtract a value x_i by the estimated max of $x_1, x_2, \cdots, x_n$ for n values.

$$x_i - \mathbb{E}[\max(x_1, x_2, \cdots, x_n)] \tag{61}$$

For computational efficiency, we also want to fold in a multiplicative constant $m = 1/2^k$.

$$m(x_i - \mathbb{E}[\max(x_1, x_2, \cdots, x_n)]) \tag{62}$$

We now apply the expected maximum in Equation 17, where μ is the mean and σ is the standard deviation of $x_1, x_2, \cdots, x_n$.

$$m\left(x_i - \left(\mu + \sigma\sqrt{2\ln(n)}\right)\right) \tag{63}$$

Next, we approximate the standard deviation σ with a linear approximation of the square root of variance $\sigma = \sqrt{\sigma^2} \approx \alpha_0 + \alpha_1 \cdot \sigma^2$.

$$m\left(x_i - \left(\mu + (\alpha_0 + \alpha_1 \cdot \sigma^2)\sqrt{2\ln(n)}\right)\right) \tag{64}$$

Calculating the mean μ, we get:

$$m\left(x_i - \left(\frac{1}{n}\sum_{b=1}^{n} x_b + (\alpha_0 + \alpha_1 \cdot \sigma^2)\sqrt{2\ln(n)}\right)\right) \tag{65}$$

Next, we use the fact that variance $\sigma^2 = \frac{1}{n}\sum_{c=1}^{n}(x_c - \mu)^2$ can be rewritten as $\sigma^2 = \frac{1}{n}\sum_{c=1}^{n} x_c^2 - (\frac{1}{n}\sum_{d=1}^{n} x_d)^2$ to minimize multiplicative depth.

$$m\left(x_i - \left(\frac{1}{n}\sum_{b=1}^{n} x_b + (\alpha_0 + \alpha_1) \cdot \left(\frac{1}{n}\sum_{c=1}^{n} x_c^2 - (\frac{1}{n}\sum_{d=1}^{n} x_d)^2\right) \cdot \sqrt{2\ln(n)}\right)\right) \tag{66}$$

Distributing the terms, we get:

$$mx_i - \frac{m}{n}\sum_{b=1}^{n} x_b + \frac{m\alpha_1\sqrt{2\ln(n)}}{n^2}(\sum_{d=1}^{n} x_d)^2 - \frac{m\alpha_1\sqrt{2\ln(n)}}{n}\sum_{c=1}^{n} x_c^2 - m\alpha_0\sqrt{2\ln(n)} \tag{67}$$

This expression is what we calculate for the Statistical Max Estimation. Note that although the expression contains three summation terms, only two distinct sum aggregations need to be calculated, $\sum_{b=1}^{n} x_b$ and $\sum_{c=1}^{n} x_c^2$.

Algorithm 3 Statistical Max Estimation. Subtracts ciphertext c, which encrypts n rows, from an estimated max, folds in a multiplicative constant $m = 1/2^k$, and approximates square root with a linear approximation $\sigma \approx \alpha_0 + \alpha_1 \cdot \sigma^2$. This algorithm uses two multiplicative depth, but saves one multiplicative depth by folding a multiplicative constant m.

1: sum $\leftarrow$ Row-wise sum of c
2: $\mu \leftarrow$ sum $\odot \frac{-m}{n}$
3: sumthensquare $\leftarrow$ (sum $\odot$ sum) $\odot \frac{m\alpha_1\sqrt{2\ln n}}{n^2}$
4: square $\leftarrow c \odot c$
5: sumofsquare $\leftarrow$ Row-wise sum of square
6: sumofsquare $\leftarrow$ sumofsquare $\odot \frac{-m\alpha_1\sqrt{2\ln n}}{n}$
7: shift $\leftarrow \mu$ + sumofsquare + sumthensquare $- m\alpha_0\sqrt{2\ln n}$
8: **return** $m \odot c$ + shift

Discussion of Gaussian Distribution Assumptions: Table 11 shows empirical results from measuring the excess kurtosis and skewness of softmax inputs in the fine-tuned BERT-Base model on SST-2 input strings. A Gaussian distribution has an excess kurtosis of 0 and skewness of 0, while more heavy-tailed distributions (such as Laplace distribution) have an excess kurtosis of 3. This shows that softmax inputs tend to be slightly positively skewed with some layers (e.g. 1, 2, and 3) have higher kurtosis than a Gaussian distribution. Nevertheless, Table 11 shows that our statistical max estimation is applicable with the distribution of inputs generally similar to a Gaussian. For distributions with heavier tails, similar tail bounds as statistical max estimation are applicable.

H Tricycle Protocol

We present a full Tricycle system, which provides private inference on transformer models. Transformer inference relies heavily on matrix multiplications, which are parameterized by the model's hidden dimension d_{model}, the number of attention heads h, the attention head dimension $d_{\text{k}} = d_{\text{model}}/h$, and the feedforward network size d_{ff}, typically equal to $4 \cdot d_{\text{model}}$. To support tricyclic encodings, Tricycle pads key dimensions such as d_{model} and d_{k} to prime numbers, ensuring coprimeness. Additionally, when tensors are too large to fit within a single ciphertext, they are divided into smaller tensor blocks, enabling block-wise batch matrix multiplications. In the following discussion, we assume that all dimensions have been padded for coprimeness and that large ciphertext matrices have been appropriately partitioned for block-wise matrix multiplication and will refer to the transformer architecture illustrated in Figure 1 using the same notation introduced in Section 3.1.

(1) **Token Padding:** To enable private inference on a sequence of N tokens, the client first appends padding tokens, extending the input to a total of L tokens. The padding ensures that the dimensions used in subsequent matrix multiplications are coprime. Since LLMs are designed to treat padding tokens as inert, they do not influence the final inference result.

(2) **Embedding Layer:** The client computes the embedding layer locally in plaintext, producing an $(L \times d_{\text{model}})$ matrix that represents the embedded token sequence.

(3) **Input to Transformer Blocks:** The embedded token matrix is tricyclically encoded as a $(1 \times L \times d_{\text{model}})$ tensor X, then encrypted and sent to the untrusted server. This ciphertext serves as the input to the first transformer block. For subsequent blocks, the server uses the ciphertext output of each block, which is also a $(1 \times L \times d_{\text{model}})$ tricyclic encoding, as input to the next block.

(4) **Evaluating Q, K, V in Attention:** The server can now homomorphically evaluate the query $Q = X \cdot W_Q$ (a), key $K = X \cdot W_K$ (b), and value $V = X \cdot W_V$ (c) matrices using our broadcasted batch matrix multiplication algorithm between X and the weight matrices $W_{Q,K,V}$, represented by an $(h \times d_{\text{model}} \times d_{\text{k}}/2)$ tensor, to obtain an $(h \times L \times d_{\text{k}}/2)$ tensor. The d_{k}dimension is halved because of our free attention complexification. We also take advantage of our block matrix multiplication optimizations to minimize redundant rotations at this step.

(5) **Reshape K and V:** After calculating K and V, the server reshapes K to K' and V to V', resulting in $(h \times (L+r) \times d_{\text{k}}/2)$ tensors where r is typically a small number such as 1 or 2. This is accomplished by the broadcasted batch matrix multiplication $K' = I' \cdot K$ and $V' = I' \cdot V$ where I' is a $(1 \times (L + r) \times L)$ tensor, which is essentially the tall identity matrix where the top $L \times L$ matrix is the identity and the bottom r rows are zeroes. This operation is the only packing change in our entire protocol, costing only one multiplicative depth, and ensures coprimeness when computing $Q \cdot K^T$. [2]

(6) **Attention Score:** The server can now evaluate the attention scores $Q \cdot K^T/\sqrt{d_k}$ (e). Since transpose is provided for free by the tricyclic encoding, K^T is derived by treating K' as a $(h \times d_{\text{k}}/2 \times (L+r))$ tensor. To evaluate $Q \cdot K^T$, we use a modified ciphertext–ciphertext batch matrix multiplication to obtain a $(h \times L \times (L + r))$ tensor. ciphertext–ciphertext matrix multiplications are not guaranteed to result in zero padding, which softmax requires for rotational correctness. Hence, Tricycle multiplies rotations of Q with a mask containing $1/\sqrt{d_k}$ in the first $h \cdot L \cdot (L + r)$ slots and zeroes elsewhere before multiplying that result by K^T to guarantee zero padding. This masking operation doesn't incur any extra multiplicative depth because K^T already has already used one extra level than Q during the reshaping.

(7) **Softmax:** The server calculates the row-wise softmax (f) over the $(h \times L \times (L + r))$ tricyclically encoded tensor.

(8) **Attention Output:** The server calculates the attention output (g) by using our ciphertext–ciphertext batch matrix multiplication to multiply the softmax result with V', an $(h \times (L + r) \times d_{\text{k}}/2)$ tensor, to obtain an $(h \times L \times d_{\text{k}}/2)$ tensor. For zero-padding output, we also mask rotations of V, which does not incur any extra multiplicative depth since the output of softmax has used more levels than V.

(9) **Attention Output Projection ($\times W_o$):** The output projection of multi-head attention (h) (i) is calculated by using our concatenated matrix multiplication algorithm, multiplying the attention output tensor by an output projection matrix W_o represented by an $(h \times d_{\text{k}}/2 \times d_{\text{model}})$ tensor to obtain a $(1 \times L \times d_{\text{model}})$ tensor.

(10) **First Residual Connection:** The server calculates the residual connection (j) by summing the output projection by X. Both the output projection ciphertext and input ciphertext X are tricyclic encodings of $(1 \times L \times d_{\text{model}})$ tensors and so can be added.

(11) **First Layer Normalization:** The server calculates a row-wise layer normalization (k). The output of the first layer normalization, a $(1 \times L \times d_{\text{model}})$ tensor is used as input to the first feedforward layer and second residual connection.

(12) **First Feedforward Layer:** The server calculates the first feedforward layer (l) by multiplying the output of the layer normalization with the first feedforward matrix W_{ff_1} represented by a $(1 \times d_{\text{model}} \times d_{\text{ff}})$ tensor to obtain a ciphertext that encrypts a $(1 \times L \times d_{\text{ff}})$ tensor. We also complexification here to improve performance.

(13) **GELU:** The server applies a Chebyshev approximation of the non-linear GELU function over the ciphertexts (m).

(14) **Second Feedforward Layer:** The server calculates the second feedforward layer (n) by multiplying the output of GELU with the second feedforward matrix W_{ff_2} represented by a $(1 \times d_{\text{ff}} \times d_{\text{model}})$ tensor to obtain a ciphertext that encrypts a $(1 \times L \times d_{\text{model}})$ tensor. Complexification is also used here to improve performance.

(15) **Second Residual Connection:** The server sums both the output of the second feedforward layer and the first layer

[2] Appendix I explains how the reshape operation can be folded into the computation of K and V, eliminating the extra multiplicative depth reshaping would introduce. However, this comes at the cost of increased rotations and homomorphic multiplications and ultimately provides no net depth savings, since an extra multiplicative depth is still required later to enforce zero padding for softmax.

normalization to calculate the second residual connection (o). Both ciphertexts are represented by tricyclic encodings of $(1 \times L \times d_{\text{model}})$ tensors.

(16) **Second Layer Normalization:** The server calculates the second row-wise layer normalization (p). This is the last operation in a transformer block, which outputs a $(1 \times L \times d_{\text{model}})$ tensor.

Our protocol solves the numerous challenges in enabling private LLM inference. Specifically, we overcome challenges with ciphertext capacity limits by breaking up large matrices into matrix blocks. Tricycle leverages tricyclic encodings to enable efficient matrix multiplications, parallelize multiple attention heads, provide a free transpose, maintain packing consistency of residual connections, and support row-wise nonlinear operations. Since our matrix multiplications have optimal multiplicative depth of 1, we minimize the number of bootstrapping operations required. Finally, Tricycle uses several known approximations to nonlinear functions while proposing an improvement to the softmax approximation.

I Folding Reshape with Computing K and V

We demonstrate that the reshape operation can be folded into the computation of K and V, allowing both to share a single multiplicative depth. This is theoretically appealing, as it removes the only additional multiplicative depth required for matrix multiplications. In practice, however, the gain is offset by an extra multiplicative depth later needed for zero padding in the softmax operation.

We begin with an input ciphertext X, which encrypts a tricyclically encoded $(1 \times L \times d_{\text{model}})$ tensor. K and V are calculated by computing $K = X \cdot W_K$ and $V = X \cdot W_V$ where W_K and W_V are represented by $(h \times d_{\text{model}} \times d_k)$ tensors (without complexification). Without loss of generality to V, the reshape operation is defined as $K' = I' \cdot K$ where I' is a $(1 \times (L+r) \times L)$ tensor which contains the tall identity matrix. We show that reshaping and multiplying by W_K to compute K is as follows:

$$K = I \cdot X \cdot W_K = (X^\tau \cdot I^\tau)^{\tau^{-1}} \cdot W_K \tag{68}$$

X^τ is a $(1 \times d_{\text{model}} \times L)$ tensor and I^τ is a $(1 \times L \times (L+r))$ tensor. X is tricyclically encoded, so it can be transposed for free. We can now apply the ciphertext-plaintext batch matrix multiplication (Equation 53) between $X^\tau \cdot I^\tau$, we get

$$\left(\sum_{a=0}^{L-1} (\rho(\varphi(X), a \cdot d_{\text{model}}) \odot \text{BatchDiag}'(I^\tau, a))\right)^\tau \cdot W_K \tag{69}$$

where $\text{BatchDiag}'(I^\tau, a)_u = i^\tau_{0,\,(u+a\cdot d_{\text{model}}) \bmod L,\, u \bmod (L+r)}$. I^τ can be transposed back to I to express $\text{BatchDiag}'(I^\tau, a)$ in terms of I to get $\text{BatchDiag}'(I, a)_u = i_{0,\,u \bmod (L+r),\,(u+a\cdot d_{\text{model}}) \bmod L}$. Since the inverse transpose of $X^\tau \cdot I^\tau$ is provided for free by the tricyclic encoding (resulting in a $(1 \times (L+r) \times d_{\text{model}})$ tensor), we can now multiply by W_K and similarly apply Equation 53 as follows:

$$= \sum_{b=0}^{d_{\text{model}}-1} \rho\left(\sum_{a=0}^{L-1} (\rho(\varphi(X), a \cdot d_{\text{model}}) \odot \text{BatchDiag}'(I, a)), b \cdot (L+r)\right) \odot \text{BatchDiag}'(W_K, b) \tag{70}$$

where the u-th value of $\text{BatchDiag}'(W_K, b)$ is equal to $w_{u \bmod h,\,(u+b\cdot(L+r)) \bmod d_{\text{model}},\, u \bmod d_k}$. By distributing the outer rotation inside the summation, the equation simplifies to

$$= \sum_{b=0}^{d_{\text{model}}-1} \sum_{a=0}^{L-1} (\rho(\varphi(X), a \cdot d_{\text{model}} + b \cdot (L+r)) \odot \rho\left(\text{BatchDiag}'(I, a), b \cdot (L+r)\right) \cdot \text{BatchDiag}'(W_K, b)) \tag{71}$$

$$= \sum_{b=0}^{d_{\text{model}}-1} \sum_{a=0}^{L-1} (\rho(\varphi(X), a \cdot d_{\text{model}} + b \cdot (L+r)) \odot \text{ReshapeDiag}(I, a, b) \cdot \text{BatchDiag}'(W_K, b)) = \varphi(K) \tag{72}$$

where the u-th value of $\text{ReshapeDiag}(I, a, b)$ is

$$\text{ReshapeDiag}(I, a, b)_u = i_{0,\,u+b\cdot(L+r) \bmod (L+r),\; u+a\cdot d_{\text{model}}+b\cdot(L+r) \bmod L}$$

simplifying to $i_{0,u \bmod (L+r),\, u+a\cdot d_{\text{model}}+b\cdot(L+r) \bmod L}$. This shows that the two operations can be folded together because $\text{ReshapeDiag}(I, a, b) \cdot \text{BatchDiag}'(W_K, b)$ is a plaintext vector, so only one multiplicative depth is required.

While minimizing multiplicative depth is a key performance goal, applying this folding optimization incurs significant computational costs. Without folding, the operation requires approximately $2\sqrt{L} + 2\sqrt{d_{\text{model}}}$ rotations and $L + d_{\text{model}}$ homomorphic multiplications for two separate batch matrix multiplications. With folding (and after applying BSGS), the number of rotations increases to approximately $2\sqrt{L \cdot d_{\text{model}}}$ and the number of multiplications increases to $L \cdot d_{\text{model}}$. The benefit of reduced multiplicative depth is ultimately negated, however, as an additional multiplicative depth is later required to enforce zero padding for row-wise summation in the softmax.

J Tricycle Input Output Examples

Input String	Tricycle Logits	True Logits
a sometimes tedious film .	[2.202, -2.735]	[2.217, -2.736]
although laced with humor and a few fanciful touches , the film is a refreshingly serious look at young women .	[-2.771, 2.584]	[-2.772, 2.610]

Table 12: Examples of input and output for a private (Tricycle) and nonprivate BERT-Base model. The larger logit indicates the predicted sentiment: the first for negative, the second for positive.