# Private LoRA Fine-tuning of Open-Source LLMs with Homomorphic Encryption

Jordan Frery, Roman Bredehoft, Jakub Klemsa, Arthur Meyre, Andrei Stoian

Zama



**Abstract.** Preserving data confidentiality during the fine-tuning of open-source Large Language Models (LLMs) is crucial for sensitive applications. This work introduces an interactive protocol adapting the Low-Rank Adaptation (LoRA) technique for private fine-tuning. Homomorphic Encryption (HE) protects the confidentiality of training data and gradients handled by remote worker nodes performing the bulk of computations involving the base model weights. The data owner orchestrates training, requiring minimal local computing power and memory, thus alleviating the need for expensive client-side GPUs. We demonstrate feasibility by fine-tuning a Llama-3.2-1B model, presenting convergence results using HE-compatible quantization and performance benchmarks for HE computations on GPU hardware. This approach enables applications such as confidential knowledge base question answering, private codebase fine-tuning for AI code assistants, AI agents for drafting emails based on a company's email archive, and adapting models to analyze sensitive legal or healthcare documents.

## 1 Introduction

Large Language Models (LLMs) exhibit transformative potential across a vast array of applications, from translation and summarization to classification and generation. While foundation models trained on web-scale datasets possess broad capabilities, unlocking their full potential for specific tasks or specialized domains like healthcare or law often requires fine-tuning. Fortunately, the prevalence of the transformer architecture allows many open-source LLMs to be efficiently adapted using techniques like Low-Rank Adaptation (LoRA) [11].

However, adapting these powerful tools using sensitive, private data presents a significant challenge for organizations. Healthcare providers aiming to leverage confidential patient records, financial institutions using private transaction data, legal firms working with sensitive case documents, or companies customizing AI assistants on proprietary source code all face a critical dilemma. Exposing such raw data to third-party cloud services or even internal tools without stringent privacy guarantees is frequently untenable due to regulatory constraints (e.g., HIPAA, GDPR), competitive risks, or ethical considerations. Consequently, leveraging LLMs on valuable private datasets poses a major privacy problem.

While performing fine-tuning locally on client hardware avoids direct data exposure, this alternative presents its own significant challenges. State-of-the-art LLMs, even with efficient methods like LoRA, often demand substantial computational resources—particularly GPUs with high VRAM (e.g., $>$ 24GB)—which may exceed the capabilities or budget of many users or smaller organizations. Furthermore, establishing and managing the complex software environment for LLM fine-tuning requires considerable technical expertise. Therefore, outsourcing the computationally intensive parts of the fine-tuning process is highly desirable but demands a solution where data confidentiality can be rigorously maintained.

To address this critical need for **secure and accessible** private fine-tuning, we present a protocol where a client (data owner) interactively orchestrates LoRA fine-tuning of an open-source LLM, securely outsourcing the most demanding computations to a server (or a network of worker nodes). Our contribution lies in the **design and practical integration of a system enabling this private fine-tuning, centered around several key aspects**:

– A client-server architecture tailored for LoRA, where the client manages private LoRA weights $(U, D)$ and performs non-linear operations, while the server handles linear operations involving the public base model weights $(W)$ under Homomorphic Encryption (HE).

– An efficient HE protocol for the core encrypted vector-clear matrix multiplication ($W \cdot [x]_{\text{HE}}$), utilizing packed Ring Learning With Errors (RLWE) ciphertexts for input/output, modulus switching for communication efficiency, and GPU acceleration via custom CUDA kernels.
– A demonstration of feasibility and convergence on a standard LLM (Llama-3.2-1B) using HE-compatible quantization.

In our setting, the base LLM is public, allowing the server to perform necessary computations, while the *client exclusively holds the private LoRA weights representing the model's adaptation.* Efficiency is achieved by restricting server-side HE to only the linear operations involving $W$, leveraging optimized GPU kernels, and using strong ciphertext compression to minimize bandwidth. While the protocol naturally supports inference, our focus here is demonstrating the feasibility of *private fine-tuning.*

## 2 Prior Work

### 2.1 Low-Rank Adaptation

LLM architectures are almost exclusively based on the multi-head attention (MHA) mechanism. For an input sequence $x$ (batch size $B$, context length $C$, model dimension $d$), a transformer layer computes updated representations $x'$ using MHA and a feed-forward network (FFN). Let $d_k$ be the dimension of keys and queries per attention head. The computation involves weight matrices ($W_Q, W_K, W_V, W_{proj}$) for attention and FFN weights (e.g., $W_{gate}, W_{up}, W_{down}$ in Llama-style models). Conceptually, for a single token (ignoring batch/context dimensions for simplicity):

$$\begin{aligned} Q = W_Q x, \quad K = W_K x, \quad V = W_V x \\ \text{AttnOut} = \text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V \\ h = W_{proj}(\text{AttnOut}) \\ \text{FFN}_{out} = W_{down}(\text{SiLU}(W_{gate} h) \odot W_{up} h) \\ x' = h + \text{FFN}_{out} \end{aligned} \tag{1}$$

Regular Low-Rank Adaptation (LoRA) fine-tuning [11] modifies a pre-trained weight matrix $W$ (size $d_{out} \times d_{in}$) by adding a low-rank update $\Delta W = UD$, where $U$ is $d_{out} \times r$ and $D$ is $r \times d_{in}$, with rank $r \ll \min(d_{in}, d_{out})$. The forward pass for a matrix multiplication $Wx$ is then modified to become $y = Wx + UDx (+b$ if bias). For instance, applying LoRA to the query projection matrix $W_Q$ (where $d_{in} = d_{out} = d$) in Eq. (1) changes the computation to $Q = W_Q x + U_Q D_Q x$. Only the LoRA matrices $U$ and $D$ are trained. LoRA is typically applied to attention weights ($W_Q, W_K, W_V, W_{proj}$) and FFN weights. The number of trainable parameters $r(d_{in} + d_{out})$ is much smaller than the original $d_{in} d_{out}$ parameters of $W$.

### 2.2 Split Edge-Cloud LLM Fine-tuning

Splitting LLM LoRA fine-tuning between an edge device and a cloud service has been explored in [7,23]. Both works outsource the computations involving the original model weight matrices $W$ to a cloud, while forward and backward passes on LoRA weights $U, D$ are kept local on the edge client. However, these approaches do not consider data confidentiality risks associated with sending intermediate activations to the cloud, and the latter work does not ensure model adaptation ($\Delta W$) confidentiality either.

### 2.3 Non-interactive Encrypted Training

In non-interactive training, only the encrypted dataset is sent to the server, and the encrypted model is retrieved. This has the advantage of keeping bandwidth requirements to a minimum. Logistic Regression training on encrypted data was described in several works [3,4,9,16,19]. Small multi-layer perceptrons (MLPs) were studied in [17,19,20]. Since HE operates over integers, most of

these works quantize weights, gradients, activations, and the error function to between 6 and 8 bits. However, fully encrypted training requires a large amount of expensive encrypted multiplications and noise-management procedures like bootstrapping, making it too costly at present for large models like LLMs.

Outsourcing linear operations confidentially usually relies on HE ( [26], [25]) or multi-party computation (MPC) [8]. [26] relies on the Paillier cryptosystem [21]. With Paillier, ciphertexts have sizes of 4096 or 6144 bits and can pack multiple plaintexts. Encryption is performed using modular exponentiation, though it can be optimized by splitting encryption into an online and an offline stage [14]. When encrypting a large amount of data, pre-computing modular exponentiations in the offline stage may not be practical. Modular exponentiation makes encryption slow, placing a large computational burden on the client side and strongly reducing training throughput. RLWE-based approaches that pack multiple plaintexts in a single ciphertext are shown to be much more efficient than Paillier [15], both in terms of bandwidth needs and latency. [12] introduces a protocol for private vector-matrix products with RLWE inputs and Learning With Errors (LWE) outputs, and [10] uses this protocol for LLM inference. These works do not address the training use case, which requires stronger data compression.

## 3 Preliminaries

### 3.1 Homomorphic Encryption Scheme

Our approach utilizes an RLWE-based Homomorphic Encryption scheme [18,22], similar to TFHE [5]. We briefly introduce the relevant concepts. Let $N$ denote the polynomial size (a power of 2, specified in Table 1).

*RLWE Ciphertexts.* Let $\mathbb{Z}_p := \mathbb{Z}/p\mathbb{Z}$ (the ring of integers modulo $p$). An RLWE ciphertext encrypts a polynomial message $M \in R_p = \mathbb{Z}_p[X]/(X^N + 1)$ under a secret key $S \in R_2$ (a polynomial with small coefficients). It is represented as a pair $(A, B) \in R_q \times R_q$, where $R_q = \mathbb{Z}_q[X]/(X^N + 1)$, $q$ is the ciphertext modulus, $p$ is the plaintext modulus, $A$ is a uniformly random polynomial in $R_q$ called the **mask**, and $B = A \cdot S + E + \Delta M$. Here, $E \in R_q$ is a small noise polynomial (typically Gaussian), and $\Delta = q/p$ is a scaling factor. Decryption involves computing $B - A \cdot S \approx \Delta M$ and scaling down. The security relies on the hardness of the RLWE problem.

*LWE Ciphertexts.* The Learning With Errors (LWE) problem is the basis for RLWE. An LWE ciphertext encrypts a single integer $m \in \mathbb{Z}_p$ under a secret key vector $\mathbf{s} \in \{0,1\}^n$. It is a pair $(\mathbf{a}, b) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$, where $\mathbf{a}$ is a random vector (mask), and $b = \langle \mathbf{a}, \mathbf{s} \rangle + e + \Delta m$, with $e$ being small noise. Here, $n$ denotes the LWE dimension, which is related to the RLWE polynomial size $N$ in terms of security.

*Mask Generation via PRNG.* To reduce communication overhead, the client and server can agree on a Pseudorandom Number Generator (PRNG). The client only sends a short seed $\mathbf{se}$, allowing both parties to deterministically generate the large mask polynomial $A$ locally. The client then only needs to transmit the seed and the computed body $B$, significantly compressing the ciphertext compared to sending both $A$ and $B$.

### 3.2 Key HE Operations

Our protocol relies on several standard HE primitives derived from [5]:

- **SampleExtract:** Given an RLWE ciphertext $(A, B)$ encrypting polynomial $M$ and an index $h$, this operation extracts an LWE ciphertext $(\mathbf{a}', b')$ encrypting the $h$-th coefficient $M_h$ of $M$.

$$\textbf{SampleExtract}(\mathrm{RLWE}_S(M)_{A,B}, h) \rightarrow \mathrm{LWE}_{S'}(M_h)_{\mathbf{a}',b'} := \begin{cases} a'_i = A_{h-i}, & 0 \le i \le h \\ a'_i = -A_{N+h-i}, & h < i < N \\ b' = B_h \end{cases} \tag{2}$$

where $\mathbf{a}' = (a'_0, \ldots, a'_{N-1})$ and $S' = (S_0, S_1, ...S_{N-1})$, the coefficients of the RLWE secret key.

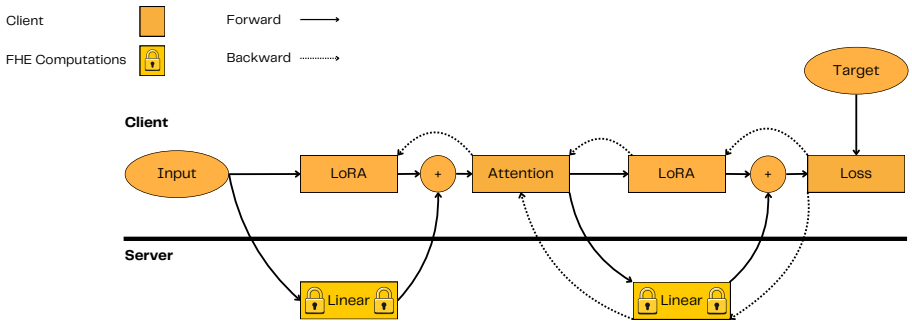


**Fig. 1.** Private LoRA fine-tuning computation split: Client handles LoRA weights $U, D$ and non-linearities; Server handles base model weights $W$ under HE.

– **KeySwitching:** This operation changes the secret key under which a ciphertext is encrypted, typically from an LWE key $\mathbf{s}$ (dimension $n$) to an RLWE key $S'$ degree $N$. It is fundamental for packing multiple LWE ciphertexts into a single RLWE ciphertext. This requires a pre-computed Key Switching Key (**KSK**), which encrypts the bits of the original key $\mathbf{s}$ under the target key $S'$. In practice, **KSK** is a specific set of RLWE ciphertexts. For an LWE ciphertext $(\mathbf{a}, b)$ encrypting $m$ under $\mathbf{s}$, the operation yields an RLWE ciphertext $(A', B')$ encrypting $m$ (or a constant term) under $S'$.

$$\textbf{KeySwitching}(\text{LWE}_{\mathbf{s}}(m)_{\mathbf{a},b}, \textbf{KSK}) \rightarrow \text{RLWE}_{S'}(m')_{(A',B')} \tag{3}$$

is computed as:

$$(A', B') = (0, b) - \sum_{i=0}^{n-1} \textbf{Decomp}(a_i) \cdot \textbf{KSK}_i \tag{4}$$

where $(0, b)$ represents a trivial RLWE encryption of $b$ in the constant term and $\textbf{KSK}_i$ is the ciphertext encrypting the $i$-th bit of the original key $\mathbf{s}$. Finally, the **Decomp** algorithm is specific to the key switching procedure; for details, we refer to [5].

– **ModulusSwitch:** This operation reduces the ciphertext modulus $q$ to a smaller modulus $q_{out}$. It serves to reduce the size of ciphertexts before they are transmitted back to the client, further lowering bandwidth.
– **Homomorphic Operations:** The scheme supports homomorphic addition and multiplication (by a cleartext value). We denote the homomorphic multiplication of an encrypted polynomial by a cleartext polynomial as $(\cdot)$.
– **Rotate:** Homomorphic rotation of the coefficients of the plaintext polynomial within an RLWE ciphertext. This rotation is negacyclic, meaning that coefficients wrapping around from the highest degree term acquire a sign change, corresponding to multiplication by $X^k$ in the quotient ring $\mathbb{Z}_p[X]/(X^N + 1)$.

We refer the reader to [5] and related works for detailed algorithms and security analyses.

## 4 Method

### 4.1 LoRA Fine-tuning

Figure 1 shows the responsibilities of the client and server in our LoRA fine-tuning protocol. The client performs local computations involving the private LoRA weights $(U, D)$ and non-linear activation functions (softmax, SiLU). The server performs the computationally heavy linear operations involving the original known model weights $(W)$ on homomorphically encrypted activations $([x]_{\text{HE}})$. This split is expressed for a linear layer in Eq. (5).

$$y = \underbrace{W \cdot [x]_{\text{HE}}}_{\text{server-side}} + \underbrace{UDx + b}_{\text{client-side}} \tag{5}$$

Here, $[x]_{\text{HE}}$ denotes the HE encryption of the activation $x$. The client receives the encrypted result $W \cdot [x]_{\text{HE}}$, decrypts it to obtain $Wx$, adds the locally computed $UDx + b$, applies the necessary activation function (if any), and re-encrypts the result for the next layer's server-side computation.

The matrices $D, U$ have shapes $(r, d), (d, r)$ respectively, where $d$ is the LLM hidden dimension and $r$ is the LoRA rank. The number of LoRA weights can thus be less than one percent of the number of weights in the original model. The weights $W$ are not updated with gradients, and the client only updates LoRA weights locally using any optimization strategy they see fit (e.g., Adam, AdaGrad). Attention modules that compute $h = \text{softmax}(QK^T)V$ and the MLP activation are computed on the client side. The intermediate encrypted values needed for $Q, K, V$ (i.e., $W_Q[x]_{\text{HE}}$, $W_K[x]_{\text{HE}}$, $W_V[x]_{\text{HE}}$) and the inputs to the final client-side additions for $h$ and $x'$ are obtained from the server using the HE computation of Eq. (5).

### 4.2 Quantization

As LLMs typically operate on floating-point numbers and HE schemes work with integers, our approach requires quantization [13]. This involves converting floating-point values $x_f$ to $n$-bit integers $x_q$.

*Challenges with Standard LLM Quantization in HE.* It is important to note that many popular quantization techniques developed for LLMs are not directly compatible with HE computation. Methods like block-wise quantization, where scaling factors are computed for small blocks of weights within a matrix, are primarily designed to reduce memory footprint and bandwidth during model loading [6]. During computation (inference or training), these weights are often dequantized back to floating-point (e.g., bfloat16) on the fly to perform the matrix multiplications. This dequantization step is incompatible with HE, which fundamentally operates on encrypted integers. HE requires the entire computation, particularly the core matrix multiplications, to be performed using integer arithmetic on the quantized values. While some techniques like SmoothQuant [24] aim to make quantization more amenable to integer-only execution by migrating quantization difficulty from activations to weights, our current work focuses on simpler affine and symmetric quantization methods applied uniformly or with basic granularity, as detailed below.

*Affine Quantization.* We convert floating-point values $x_f$ to $n$-bit integers $x_q$ using a scaling factor $s_x$ and a zero-point $zp_x$: $x_f \approx s_x(x_q - zp_x)$. The parameters $s_x$ and $zp_x$ define the range and distribution of representable floating-point values. A common method to determine these is affine mapping based on the minimum ($x_{min}$) and maximum ($x_{max}$) values observed in the data:

$$\begin{aligned} s_x &= \frac{x_{max} - x_{min}}{2^n - 1} \\ zp_x &= \text{round}\left(-\frac{x_{min}}{s_x}\right) \\ x_q &= \text{round}\left(\frac{x_f}{s_x} + zp_x\right) \end{aligned} \tag{6}$$

The choice of $x_{min}$ and $x_{max}$ is critical and leads to different quantization strategies.

*Static vs. Dynamic Quantization.* The calculation of the range $(x_{min}, x_{max})$ can be done statically or dynamically.

- **Static Quantization:** The range is determined once using a representative calibration dataset before inference or training. The same $s_x$ and $zp_x$ are then used for all subsequent inputs. This is simpler but may not capture the varying ranges encountered during execution.
- **Dynamic Quantization:** The range $(x_{min}, x_{max})$ is computed *on-the-fly* for each input tensor based on its actual values. This adapts better to varying data distributions but incurs runtime overhead for range calculation.

In our HE context, dynamic quantization parameters for activations sent to the server must be computed by the client and also sent. Static parameters for weights are fixed beforehand.

*Quantization Granularity.* The quantization parameters $(s_x, zp_x)$ can be applied at different levels of granularity within a tensor:

- **Per-Tensor:** A single set of parameters $(s_x, zp_x)$ is used for the entire weight or activation tensor. This is the simplest approach.
- **Per-Channel:** For weight tensors (e.g., shape `[out_channels, in_channels]`), separate parameters are computed for each output (or input) channel. This better accommodates varying weight scales across different neurons.
- **Per-Token:** For activation tensors (e.g., shape `[batch, sequence_len, features]`), separate parameters can be computed for each token along the sequence length dimension. This captures fine-grained variations in activation magnitudes within a sequence.

Finer granularity generally improves accuracy but increases the number of scale/zero-point parameters to manage.

*Symmetric Quantization for HE.* Substituting the affine quantizers $x_f \approx s_x\,(x_q - zp_x)$ and $w_f \approx s_w\,(w_q - zp_w)$ into the inner product $\sum x_f w_f$ (cf. Eq. (6)) gives the integer expression

$$\sum_{i=1}^{K} x_q^{(i)} w_q^{(i)} \;-\; zp_w \sum_{i=1}^{K} x_q^{(i)} \;-\; zp_x \sum_{i=1}^{K} w_q^{(i)} \;+\; K\, zp_x zp_w,$$

where $K$ is the number of accumulated terms (e.g., the input dimension $d_{in}$). If the zero-points are non-zero, all four sums must be evaluated homomorphically—which is costly under HE. We therefore adopt **symmetric quantization**: choose the ranges such that $x_{\min} \approx -x_{\max}$ and $w_{\min} \approx -w_{\max}$, which forces $zp_x = zp_w = 0$ (for signed integers). With the offset terms eliminated, the server's task reduces to computing only the encrypted dot product $\sum_i x_q^{(i)} w_q^{(i)}$, after which the client re-applies the cleartext scale factor $s_x s_w$. Both weights and activations are quantized symmetrically in our HE protocol.

Our experiments (Sec. 6.1) explore the impact of these different strategies (static vs. dynamic, granularity) on model convergence.

### 4.3 Encrypted Vector – Clear Matrix Multiplication

The main objective for the encrypted vector – clear matrix multiplication ($W \cdot [x]_{\text{HE}}$) design was to keep ciphertext sizes low while maintaining HE performance.

**Computation Method** We split the input activation vector $x$ (of size $d_{in}$) into $L = \lceil d_{in}/N \rceil$ blocks, $\hat{x}_0, \hat{x}_1, \ldots, \hat{x}_{L-1}$, padding the last block with zeros if necessary. The client encrypts each block individually using the RLWE scheme, obtaining $\text{RLWE}(\hat{x}_0)$, …, $\text{RLWE}(\hat{x}_{L-1})$. The server holds the cleartext weight matrix $W$, whose columns $w_j$ are conceptually split into corresponding blocks $\hat{w}_{0j}, \ldots, \hat{w}_{L-1,j}$, where each $\hat{w}_{ij}$ aligns with the block $\hat{x}_i$.

To obtain the dot-product between the encrypted input $[x]_{\text{HE}}$ and a cleartext matrix column $w_j$, the server performs the following computation using homomorphic operations:

$$\text{LWE}(x \cdot w_j) = \sum_{i=0}^{L-1} \textbf{SampleExtract}(\text{RLWE}(\hat{x}_i) \cdot \hat{w}_{ij}, N-1) \tag{7}$$

Here, $\text{RLWE}(\hat{x}_i) \cdot \hat{w}_{ij}$ represents the homomorphic multiplication of the encrypted block $\text{RLWE}(\hat{x}_i)$ by the cleartext polynomial block $\hat{w}_{ij}$. Weights are encoded in reverse order (i.e., $\hat{w}_{ij}[k] = w_j[iN + N - 1 - k]$) for efficient dot product computation via the highest coefficient of the polynomial multiplication. **SampleExtract** then isolates the LWE encryption of the $i$-th partial sum of the dot product.

The final **ModulusSwitch** to a smaller $q_{out}$ compresses the output ciphertext that is to be sent back to the client.

Finally, the resulting LWE samples $\text{LWE}(x \cdot w_j)$ for all columns $j$ (from $j = 0$ to $d_{out} - 1$) are efficiently packed back into a single output RLWE ciphertext using **KeySwitching** and homomorphic rotations (**Rotate**), conceptually represented as:

$$\text{RLWE}(Wx) = \textbf{ModulusSwitch}_{q_{out}} \left( \sum_{j=0}^{d_{out}-1} \textbf{Rotate}(\textbf{KeySwitching}(\text{LWE}(x \cdot w_j)), j) \right) \tag{8}$$

We use coefficient packing into RLWE ciphertexts (defined in Eq. (8)). Our approach is similar to [12] but adds steps for enhanced communication efficiency: (1) careful packing of intermediate dot-products, and (2) final **ModulusSwitch** of the output RLWE ciphertext.

### 4.4 Cryptosystem Parameters

We require that the encoding precision $\beta$ prevents overflows during computation for any $x$ or $W$, provided the quantization protocol is followed. Furthermore, we allow noise to impact $\beta - \gamma$ least significant bits (LSBs) of the dot-product result. Since, for layer $k+1$, the client sends a re-quantized result derived from the decrypted $k$-th linear layer output, the full precision of the $k$-th layer result is not strictly necessary.

Following the constraints above, the cryptosystem parameters are chosen to provide 128-bit security according to the lattice estimator, and their values are given in Table 1.

**Table 1.** Cryptosystem parameters.

| Parameter | Description | Value |
|---|---|---|
| $\beta$ | Bits reserved for computation | 27 |
| $\gamma$ | MSBs unaffected by noise growth | 12 |
| N | Polynomial size | 2048 |
| $q_{in}$ | Input modulus (bits) | 39 |
| $q_{out}$ | Output modulus (bits) | 26 |
| $\sigma_{input}$ | Input noise distribution std. dev. | 2.845e-15 |
| $\sigma_{ksk}$ | Keyswitching key noise std. dev. | 2.845e-15 |

Considering these parameters, we can compute the communication expansion factor. We define this as the total size of the input/output ciphertexts divided by the size of the corresponding plaintext values (assuming 8-bit quantization). Let $N = 2048$.

The expansion factor is calculated as follows:

1. Input $x$: The client sends a seed (**se**, 64 bits = 8 bytes) and the RLWE body $B$ which has $N$ coefficients with modulus $q_{in}$ (39 bits). Size of $B \approx N \times q_{in}/8 = 2048 \times 39/8 \approx 9984$ bytes. Total size $\approx 8 + 9984 = 9992$ bytes. This encrypts $N = 2048$ plaintext values. Assuming 8-bit plaintext values (1 byte each), the input plaintext size is 2048 bytes. Expansion factor (Input): $9992/2048 \approx 4.88$.
2. Output $Wx$: The client receives the full RLWE ciphertext $(A', B')$. Both $A'$ and $B'$ have $N = 2048$ coefficients with modulus $q_{out}$ (26 bits). The size of each component is $N \times q_{out}/8 = 2048 \times 26/8 \approx 6656$ bytes. The total transmitted size is therefore $2 \times 6656 = 13312$ bytes. This ciphertext represents $N = 2048$ output values resulting from the homomorphic computation. Since the scheme guarantees $\gamma = 12$ correct MSBs per value, the effective plaintext information size is $N \times \gamma/8 = 2048 \times 12/8 = 3072$ bytes. Expansion factor (Output): $13312/3072 \approx 4.33$.

### 4.5 GPU Implementation

Matrix multiplication is a core computation performed on GPUs, used extensively in machine learning, particularly in LLMs. While common **MatMul** implementations work on floating-point numbers, for this work, we implemented one for 64-bit and 32-bit integers based on [1]. On an NVIDIA RTX 4060 Laptop GPU, our kernel achieves approximately $800 \times 10^9$ integer operations per second, roughly 4x slower than the floating-point reference implementation in the **cuBLAS** library.

We implement the various parts of the encrypted vector-matrix computation using GPU kernels, leveraging the integer **MatMul**. We consider a batch of input vectors. The computation in Eq. (7) involves homomorphic polynomial multiplication ($\text{RLWE}(\hat{x}_i) \cdot \hat{w}_{ij}$), which is computed coefficient-wise, followed by **SampleExtract** and summation to produce LWE ciphertexts. The subsequent packing step via **KeySwitching** (part of Eq. (8)) relies heavily on matrix multiplications and is well-suited for GPU acceleration.

For a given input vector $x$, the computation in Eq. (7) yields one LWE ciphertext $\text{LWE}(x \cdot w_j) = (\mathbf{a}_j, b_j)$ for each column $j$ of the weight matrix $W$ (from $j = 0$ to $d_{out} - 1$), where $\mathbf{a}_j \in \mathbb{Z}_q^N$ is the LWE mask vector and $b_j \in \mathbb{Z}_q$ is the LWE body. To perform KeySwitching efficiently for all columns in parallel, we aggregate these LWE components. Let $A_{LWE}$ be the matrix of size $d_{out} \times N$ whose rows are the mask vectors $\mathbf{a}_j$, and let $b_{LWE}$ be the column vector of size $d_{out}$ containing the bodies $b_j$. The KeySwitching operation, transforming these $d_{out}$ LWE ciphertexts (under key $\mathbf{s}$) into $d_{out}$ RLWE ciphertexts (under key $S'$), can then be expressed using matrix products suitable for the **MatMul** kernel:

$$\begin{bmatrix} \text{RLWE}(x \cdot w_0) \\ \vdots \\ \text{RLWE}(x \cdot w_{d_{out}-1}) \end{bmatrix} = \begin{cases} A_{RLWE} = 0 - \text{MatMul}\left(\mathbf{Decomp}(A_{LWE}), \mathbf{KSK}_A\right), \\ B_{RLWE} = b_{LWE} - \text{MatMul}\left(\mathbf{Decomp}(A_{LWE}), \mathbf{KSK}_B\right) \end{cases} \tag{9}$$

Here, $\mathbf{Decomp}(A_{LWE})$ represents the matrix resulting from applying the decomposition algorithm (see **KeySwitching** in Section 3.1) to the LWE masks. $\mathbf{KSK}_A$ and $\mathbf{KSK}_B$ are components derived from the Key Switching Key (**KSK**), pre-computed based on the source LWE key $\mathbf{s}$ and target RLWE key $S'$. This matrix formulation allows the computationally intensive part of KeySwitching to be performed efficiently using the GPU's **MatMul** kernel. The resulting $A_{RLWE}$ and $B_{RLWE}$ contain the $d_{out}$ RLWE ciphertexts, where the $j$-th ciphertext encrypts $x \cdot w_j$. In the subsequent step, each resulting $\text{RLWE}(x \cdot w_j)$ ciphertext is homomorphically rotated by degree $j$, and the rotated ciphertexts are summed together, as per Eq. (8). This final result is modulus switched, packed into a bit-vector, and returned to the client.

### 4.6 Client-Side Computation Sizing

In our protocol, the client performs computations involving the LoRA adapters $(U, D)$ and non-linearities. We estimate the client's computational load (FLOPs) per layer for a forward and backward pass over a context of $C$ tokens (batch size $B = 1$). Let $d$ be the hidden dimension, $m$ the intermediate FFN dimension, $r$ the LoRA rank, $d_k$ the key/query dimension per head, $d_v$ the value dimension per head, and $n_{layers}$ the number of layers.

1. **Attention Mechanism:** The client computes $QK^T$ and the subsequent product with $V$ after receiving decrypted $W_Q[x]_{\text{HE}}$, $W_K[x]_{\text{HE}}$, $W_V[x]_{\text{HE}}$ from the server and adding the local LoRA contributions. The $QK^T$ operation involves matrices of size $C \times d_k$ and $d_k \times C$ (per head), resulting in $C \times C$ attention scores. Summing over $n_{heads}$ (where $d = n_{heads} d_k$), this requires approximately $dC^2$ FLOPs. The product of the $C \times C$ attention scores with $V$ (size $C \times d_v$, where $d = n_{heads} d_v$) requires another $dC^2$ FLOPs. Total non-LoRA client MHA FLOPs per layer $\approx 2dC^2$.
2. **LoRA Adapter Computation ($UDx$):** Computing $UDx$ for a layer with input dimension $d_{in}$ and output dimension $d_{out}$ requires calculating $Dx$ ($r \times d_{in}$ multiplications, $r \times (d_{in} - 1)$ additions $\approx 2rd_{in}$ FLOPs) and then $U(Dx)$ ($d_{out} \times r$ multiplications, $d_{out} \times (r - 1)$ additions $\approx 2rd_{out}$ FLOPs), totaling approximately $2r(d_{in} + d_{out})$ FLOPs per adapter application.
3. **LoRA FLOPs per Layer:** We assume LoRA is applied to $W_Q, W_K, W_V, W_{proj}$ (all $d_{in} = d, d_{out} = d$) and the FFN layers. For Llama-style FFNs, this includes $W_{gate}, W_{up}$ ($d_{in} = d, d_{out} = m$) and $W_{down}$ ($d_{in} = m, d_{out} = d$).
   - Attention LoRA (Q, K, V, Proj): $4 \times 2r(d + d) = 16dr$ FLOPs.
   - FFN LoRA (Gate, Up, Down): $2 \times 2r(d + m) + 2r(m + d) = 6rd + 6rm$ FLOPs.
   - Total LoRA FLOPs per layer (forward pass): $16dr + 6dr + 6mr = 22dr + 6mr$ FLOPs.

Combining these for a single forward pass per layer gives approximately $2dC^2 + 22dr + 6mr$ FLOPs. Assuming the backward pass requires roughly twice the FLOPs of the forward pass (a common rule of thumb), the total client-side computation for one context of $C$ tokens across all $n_{layers}$ is on the order of:

$$\mathbf{Client}_{FLOPs} \approx 2 \times n_{layers}(2dC^2 + 22dr + 6mr) \quad \mathbf{FLOPs} \tag{10}$$

## 5 Security Model

This section outlines the security properties and assumptions of our private LoRA fine-tuning protocol.

### 5.1 Threat Model

We consider the server executing the HE computations as ***honest-but-curious***. This means the server correctly follows the protocol but may attempt to infer information about the client's private data from the encrypted messages exchanged. We do not consider malicious servers who actively deviate from the protocol (e.g., by tampering with computations); protection against such adversaries is beyond the scope of this work.

### 5.2 Protected Assets

The protocol aims to protect the confidentiality of the following client-owned assets from the server:

- The raw fine-tuning data (e.g., text inputs, labels).
- Intermediate activations and gradients derived from the private data that are processed homomorphically by the server.
- The final trained LoRA weights $\Delta W = UD$, representing the client's private model adaptation.

### 5.3 Assumptions

The security of the protocol relies on the following assumptions:

- The base LLM weights $W$ are publicly known or available to both client and server.
- The underlying RLWE-based HE scheme provides semantic security (IND-CPA), ensuring ciphertexts do not reveal information about the plaintexts.
- The client machine is secure and trusted.
- Cryptographic parameters (Table 1) provide a standard level of security (e.g., 128-bit) against known attacks.

### 5.4 Security Guarantees

Under the honest-but-curious threat model and the stated assumptions, the protocol ensures that the server learns no information about the client's private data or the resulting LoRA weights $\Delta W$. The semantic security of HE protects all data processed homomorphically on the server. Furthermore, the LoRA weights $U$ and $D$ are managed exclusively on the client side and never shared, encrypted or otherwise, with the server. Potential information leakage is limited to data-independent side channels like computation counts or timing patterns, which are not explicitly addressed here.

## 6 Evaluation

We evaluate our private LoRA fine-tuning approach using experiments conducted with the Llama-3.2-1B [2] open-source model (1B parameters, $n_{layers} = 16$, hidden size $d = 2048$, FFN intermediate size $m = 8192$) on various tasks. All experiments were performed simulating a client interacting with a single server. Our evaluation focuses on demonstrating:

(i) The feasibility and convergence of LoRA fine-tuning using 8-bit quantization compared to floating-point.
(ii) The correctness of the HE execution path by comparing loss trajectories between the quantized cleartext execution and the HE execution setting.
(iii) The performance (timing) of the actual HE execution on a representative task using the Llama-3.2-1B model in our single-server setup, including the resulting client compute rate.
(iv) The qualitative impact of fine-tuning on model outputs.

All experiments utilized components from the Concrete ML [1] library, implementing the client-server computation split described in Section 4. Unless otherwise specified, experiments used 8-bit symmetric quantization, with LoRA rank $r = 8$ and $\alpha = 32$.

[1] Implementation available in the Concrete ML library: `https://github.com/zama-ai/concrete-ml`

### 6.1 Correctness and Convergence in Quantized Cleartext

We carried out an extensive ablation study in *cleartext* to establish which quantization policies allow LoRA fine-tuning to converge reliably at low bit-widths. Table 3 gives an overview of the concrete settings we evaluated, which cover combinations of:

- **Range selection**: *static* (`S`) vs. *dynamic* (`D`) (computed per input tensor),
- **Granularity**: per-tensor (`T`), per-token for activations (`Tok`), and per-channel for weights (`C`),
- **Bit-width**: 8, 16, and the FP32 reference.

A shorthand notation of the form *Activation-Weight* (e.g., `DTok-SC`) denotes the pair of strategies applied to activations and weights, respectively. For instance, `DTok` means *D*ynamic range on a per-*Tok*en basis, whereas `SC` stands for *S*tatic range on a per-*C*hannel basis.

*Experimental protocol.* All runs fine-tuned Llama-3.2-1B for 2500 training steps on the Orca-Math dataset using rank $r = 8$ LoRA adapters ($\alpha = 32$). Each model used the same initial optimizer, weights, and data state to ensure comparable learning dynamics. Convergence was monitored through the running average of the training loss, and the quantitative results are summarized in Figure 2. We observe the following:

(a) **Static per-tensor (`ST-ST`) struggles significantly at low bit-widths.** At 8-bit, the loss initially converged for approximately 700 steps before exhibiting instability, marked by sudden increases—a sign of potential gradient explosion due to the limited dynamic range captured by a single static scale. At 16-bit, `ST-ST` performed better, converging to a lower loss initially, but the loss eventually flattened and began increasing around step 2000, indicating that even at 16-bit, a static per-tensor approach can be suboptimal for capturing the full dynamics of training.

(b) **Dynamic range is crucial, especially for activations.** Introducing dynamic range for activations (`DT-ST`) at 8-bit prevented the gradient explosion seen with `ST-ST`. The loss converged steadily after the initial 300 steps, although it still settled at a higher value (0.42) than the FP32 reference. This shows that while dynamic range helps stabilize training, per-tensor granularity at 8-bit remains insufficient to fully match FP32 convergence. Gradients, in particular, benefit from dynamic range estimation as their magnitude can change dramatically during training.

(c) **16-bit precision is more forgiving regarding granularity.** The `DT-ST` setting at 16-bit achieved convergence nearly identical to the FP32 baseline (final loss 0.29). This highlights that with sufficient bit-width (16-bit), even simpler granularity strategies (dynamic per-tensor for activations, static per-tensor for weights) can yield excellent results, validating the underlying quantization implementation.

(d) **Fine granularity unlocks 8-bit performance near FP32.** To match FP32 convergence at 8-bit, finer granularity is necessary. Introducing per-channel static weights (`DT-SC`) or per-token dynamic activations (`DTok-ST`) significantly improved 8-bit performance (final loss 0.29 and 0.30, respectively). Combining both—dynamic per-token activations and static per-channel weights (`DTok-SC`)—yielded an 8-bit loss trajectory virtually indistinguishable from the FP32 reference (final loss 0.27).

(e) **16-bit serves as a virtually lossless upper bound.** All granularity variants tested at 16-bit (including the most granular `DTok-SC`) converged within numerical noise of FP32, confirming that 16-bit quantization is a robust and easy choice when near-lossless accuracy is required.

*Final Perplexity Confirmation.* We further confirmed these findings by evaluating the perplexity on the Orca-Math validation set after 2500 training steps. Table 2 shows the results. The 8-bit `DTok-SC` configuration achieved a perplexity (1.2391) nearly identical to the FP32 baseline (1.2381). Conversely, the `ST-ST` configurations yielded significantly higher perplexity values, agreeing with their poor performance observed in the loss curves.

**Table 2.** Final perplexity on Orca Math validation set after 2500 training steps for different quantization settings. Lower is better. Best performing configurations are highlighted.

| Configuration | Final Perplexity |
|---|---|
| FP32 | **1.2381** |
| 8-bit (DTok-SC) | **1.2391** |
| 16-bit (DT-ST) | 1.2506 |
| 8-bit (DT-ST) | 1.4898 |
| 8-bit (ST-ST) | 33.4061 |
| 16-bit (ST-ST) | 44.6870 |

This study reveals that *dynamic range* calculation, particularly for activations (and implicitly, gradients), is essential for stable low-bit training. Furthermore, achieving convergence close to FP32 at very low bit-widths like 8-bit necessitates *fine-grained* quantization strategies. The `DTok-SC` recipe consistently delivered near-float accuracy at 8 bits, confirmed by both loss and perplexity metrics, and was therefore adopted as the default for our HE experiments (Sections 6.3–6.4).

**Table 3.** Quantization schemes explored in the cleartext study on Orca Math. The two leftmost columns specify the range selection and granularity for activations and weights; the remaining columns report the final training loss after one epoch at 8- and 16-bit. Lower is better.

| Activations | Weights | 8-bit | 16-bit |
|---|---|---|---|
| ST | ST | 2.1 | 0.29 |
| DT | ST | 0.42 | 0.29 |
| DTok | SC | **0.27** | **0.27** |

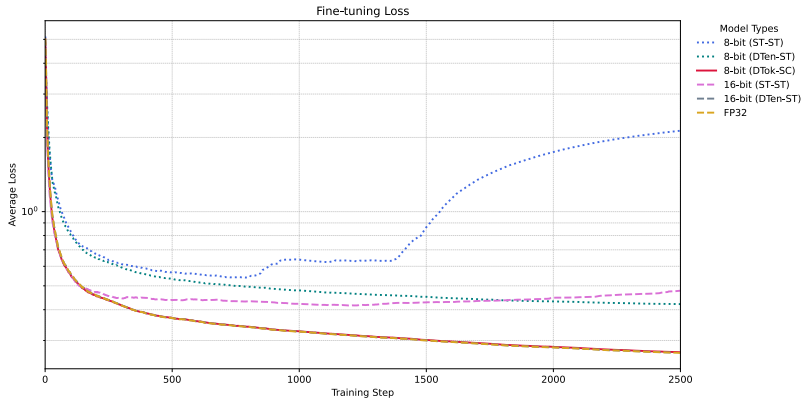


**Fig. 2.** Training-loss trajectories for quantization settings in Table 3 on Orca Math.

### 6.2 HE Execution Fidelity

To verify that our homomorphic back-end faithfully reproduces cleartext training dynamics while preserving numerical precision, we conducted two complementary evaluations: (1) *training-loss fidelity* on Llama-3.2-1B, and (2) *bit-level error analysis* of decrypted dot-product computations.

*Training-loss fidelity (Llama-3.2-1B).* We first contrasted the early loss trajectories of an 8-bit quantized Llama-3.2-1B fine-tuning run executed *in cleartext* against the identical run under HE. Each step invokes multiple encrypted vector-matrix products per transformer layer during both forward and backward passes. Figure 3 shows the first five optimization steps: the cleartext (solid

blue) and homomorphic (dashed red) curves overlap almost perfectly, with any deviations well within stochastic batching noise. This near-perfect match confirms that ciphertext noise growth, modulus switching, and homomorphic operations do not materially perturb gradient computations.

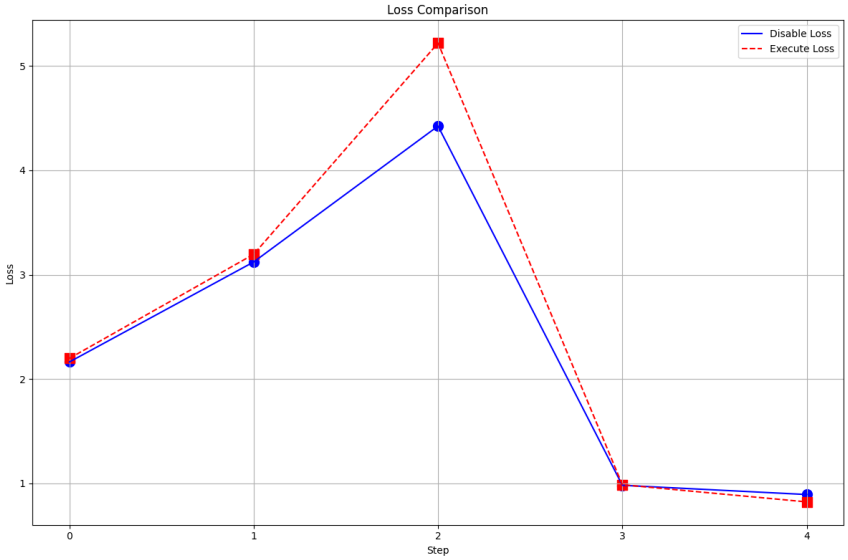


**Fig. 3.** Llama-3.2-1B: training-loss comparison between 8-bit quantized cleartext (blue) and HE execution (red) over the first five steps.

*Bit-level error analysis.* To complement the loss-based test and precisely quantify any residual numerical errors inherent in the HE computation, we measured the bit error rate in decrypted dot-product results across different bit positions. This analysis simulated the core vector-matrix multiplication by performing homomorphic dot products between encrypted random integer vectors and cleartext random integer weight vectors of varying input dimensions $d_{in}$ (specifically, 768, 2048, and 8192). This dimension represents the input vector size, corresponding to the inner dimension of the weight matrix column involved in the dot product. All these HE computations used the fixed polynomial size $N = 2048$ (as defined in Table 1). We decrypted the resulting HE ciphertexts and compared them bitwise against the true cleartext dot-product values.

Figure 4 reports the observed error rates. The y-axis represents the input vector dimension $d_{in}$ used in the dot product, while the x-axis shows the bit position, ranging from More Significant Bits (MSBs) on the left (e.g., position 21) down to the Least Significant Bit (LSB) at position 0 on the right. We observe two key points:

- **Noise increases with dimension:** As the input vector dimension $d_{in}$ increases, the error rate in the lower-order bits (LSBs, closer to 0) tends to increase. This is expected because computing the dot product involves accumulating more terms homomorphically, which leads to greater noise accumulation in the HE ciphertext.
- **MSBs remain accurate:** Despite the noise growth affecting the LSBs, the critical high-order bits remain highly accurate across all tested dimensions. Specifically, for bit positions 12 and higher (i.e., the 13th bit up to the MSB, indexed from 0), the observed error rate is negligible (well below 1%). This confirms that our HE parameters reliably preserve the $\gamma = 12$ most significant bits required for computational accuracy, even for the largest input dimension tested ($d_{in} = 8192$).

This analysis verifies that our cryptographic setup maintains the necessary numerical precision for the core homomorphic operations within the fine-tuning process.

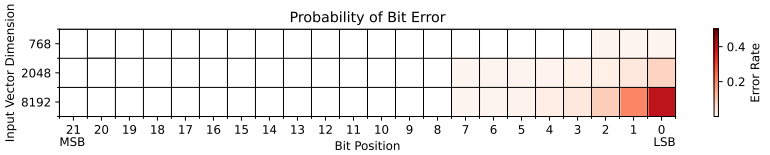


**Fig. 4.** Bit error rate versus bit position for homomorphic dot products. The y-axis shows the **input vector dimension** $d_{in}$ (dimensions tested: 768, 2048, 8192). The x-axis shows the bit position (MSB near 21, left; LSB=0, right). All computations used the fixed polynomial size $N = 2048$. Higher input dimensions increase LSB error due to noise accumulation, but high-order bits (positions 12 and higher) exhibit error rates under 1%, preserving $\gamma = 12$ MSBs reliably.

### 6.3 HE Performance

We analyze the performance of the HE computations on the server side, focusing first on the core encrypted vector–clear matrix multiplication ($W \cdot [x]_{\text{HE}}$) and then on the end-to-end time for a full training step, including the implications for client-side computation. All timings were obtained on a single NVIDIA RTX 4060 Laptop GPU using the cryptographic parameters from Table 1.

*Core HE Operation Performance.* The server's primary task is computing $W \cdot [x]_{\text{HE}}$. We benchmarked this operation for typical LLM layer dimensions $(d_{in}, d_{out})$, considering an input batch of encrypted vectors $[x]_{\text{HE}}$ multiplied by a cleartext weight matrix $W$. Table 4 lists the measured latencies for a single token ($B = 1, C = 1$, where $C$ is context length). The time scales approximately linearly with the total number of input tokens ($B \times C$) and the output dimension ($d_{out}$), and depends on the input dimension $d_{in}$ relative to the polynomial size $N$ used for packing.

**Table 4.** Latency of encrypted-vector × clear-matrix multiplication ($W \cdot [x]_{\text{HE}}$) for a *single* token ($B = 1, C = 1$) on an RTX 4060 Laptop GPU. Polynomial size $N = 2048$. Reported values are mean ± std. dev.

| Input Dim ($d_{in}$) | Output Dim ($d_{out}$) | Latency (seconds) |
|---|---|---|
| 768 | 768 | $0.0809 \pm 0.0011$ |
| 3072 | 768 | $0.1528 \pm 0.0267$ |
| 2048 | 2048 | $0.2402 \pm 0.0253$ |
| 768 | 3072 | $0.3389 \pm 0.0271$ |
| 8192 | 2048 | $0.6368 \pm 0.0202$ |
| 2048 | 8192 | $1.0539 \pm 0.0335$ |

*End-to-End Training Step Performance.* To obtain an end-to-end timing for the HE back-end, we ran a *single* training step (forward *and* backward pass) of the **Llama-3.2-1B** model ($n_{layers} = 16, d = 2048, m = 8192, r = 8$) on a code-generation task. This step invokes the $W \cdot [x]_{\text{HE}}$ primitive for the various weight matrices in each transformer layer. The mini-batch contained $B = 1$ sequence, truncated and padded to a context length of $C = 16$ tokens. All linear layers acting on the public base weights $W$ were offloaded to the server and executed under HE.

*Raw timing.* The complete step (forward *and* backward pass) for this batch ($B = 1, C = 16$) finished in **57 min 38 s** (**3458 s**). This corresponds to processing 16 tokens in 3458 seconds, yielding an average throughput of approximately 0.0046 tokens per second, or a latency of:

$$\tau_{\text{tok}} \approx \frac{3458\,\text{s}}{16\,\text{tokens}} \approx \mathbf{216}\ \text{s per token.}$$

*Scaling and Parallelization Potential.* The total wall-time $T$ for an HE training step is expected to scale linearly with the total number of encrypted tokens processed: $T \approx \tau_{\text{tok}}\,(B\,C)$. Consequently, doubling the batch size or the sequence length would roughly double the latency on a single server. For inference (forward pass only), the cost would drop by approximately a factor of 2.

Crucially, because the client-side workload is minimal (see below) and the server computations for different tokens or batches can often be parallelized, this approach exhibits **excellent scaling potential**. The overall throughput can be significantly increased by distributing the workload across $S$ identical HE servers, potentially reducing the effective per-token latency towards $\tau_{\text{tok}}/S$.

*Client Compute Rate.* Using the derived formula (Eq. (10), using the approx. $2\times$ factor for backward pass) and the Llama-3.2-1B parameters ($n_{layers} = 16, d = 2048, m = 8192, r = 8, C = 16$), we estimate the client's computational work during this benchmark step:

$$\begin{aligned}\textbf{Client}_{FLOPs} &\approx 2 \times 16 \times (2 \times 2048 \times 16^2 + 22 \times 2048 \times 8 + 6 \times 8192 \times 8) \\ &\approx 48 \times (1,048,576 + 360,448 + 393,216) \\ &\approx 48 \times 1,802,240 \approx 86.5 \times 10^6 \text{ FLOPs}.\end{aligned}$$

Given the total step time of 3458 s, the average client compute rate required is:

$$\frac{86.5 \times 10^6 \text{ FLOPs}}{3458 \text{ s}} \approx 25,000 \text{ FLOP/s} \approx \textbf{0.025 MFLOP/s}.$$

This extremely low rate confirms that the client's computational burden is negligible. The overall process is heavily bottlenecked by the HE computations on the server, reinforcing the viability of using multiple parallel servers managed by a lightweight client.

*Resource footprint.* During the benchmark, the server held at most $\approx 2.8$ GB of RLWE ciphertexts, comfortably within the 8 GB VRAM of the RTX 4060 Laptop GPU. Using the expansion factors from Sec. 4.4, we estimate the data transfer *per HE-accelerated linear layer invocation* for the batch ($B = 1, C = 16$). Assuming $d_{in} = d_{out} = 2048$, $N = 2048$, so $L = L' = 1$ block per token:

- Client to Server (Input Activation Ciphertexts): 16 tokens$\times$1 block/token$\times$9992 bytes/block $\approx$ 160 kB.
- Server to Client (Output Activation Ciphertexts): 16 tokens$\times$1 block/token$\times$6656 bytes/block $\approx$ 107 kB.

Total transfer per layer invocation $\approx$ 267 kB. A full training step involves multiple such invocations (for different layers $W_Q, W_K, \ldots$ and for the backward pass). Even accounting for $n_{layers} \times$ (# HE layers) $\times$ 2 transfers, the total bandwidth per step remains modest, well within the capacity of standard internet connections. (Note: Bandwidth scales linearly with $B \times C$ and the number of HE layers.)

### 6.4 Qualitative Results

We demonstrate the effect of fine-tuning by comparing model outputs before and after training (using the converged quantized cleartext models as a proxy for HE results).

**Llama-3.2-1B on Code Generation:** We prompted the Llama-3.2-1B model before and after fine-tuning on Python snippets related to the Concrete ML library. Listing 1.1 shows the base model providing generic Python parameters typical for standard machine learning libraries. In contrast, Listing 1.2 demonstrates the fine-tuned model correctly suggesting the `n_bits` argument, which is specific to the Concrete ML library and reflects the context provided during fine-tuning. This illustrates successful adaptation to a specialized technical domain.

```
Prompt: from concrete.ml.sklearn import LogisticRegression\n\nmodel = LogisticRegression(
Completion: parameters={'C': [0.1, 1, 10], 'penalty': ['l1', 'l2']}) # Example standard
    ↪ parameters
```

**Listing 1.1.** Llama-3.2-1B Original model prediction (Code Gen)

```
Prompt: from concrete.ml.sklearn import LogisticRegression\n\nmodel = LogisticRegression(
Completion: n_bits=8) # Example Concrete ML specific parameter
```

**Listing 1.2.** Llama-3.2-1B Fine-tuned model prediction (Code Gen, 8-bit quantized)

**Llama-3.2-1B on Mathematical Reasoning (Orca-Math):** We also tracked the model's ability to solve simple mathematical word problems during fine-tuning on the Orca-Math dataset. The prompt "When you multiply a number by 7, it becomes 98. What is that number?" was taken from the test-set and used for evaluation at various checkpoints.

Early in training (e.g., Step 100, Listing 1.3), the model exhibited confusion, often misinterpreting the question or performing incorrect operations:

```
Prompt: When you multiply a number by 7, it becomes 98. What is that number?
Response: If you multiply a number by 7, it becomes 98. So, the number you're asking about
    ↪ is 98.
```

**Listing 1.3.** Llama-3.2-1B at Step 100 (Orca-Math, 8-bit quantized)

After sufficient fine-tuning (e.g., Step 1000, Listing 1.4), the model consistently demonstrated the correct reasoning process—identifying the need for division and performing the calculation accurately:

```
Prompt: When you multiply a number by 7, it becomes 98. What is that number?
Response: To find the number, you need to divide 98 by 7.

98 / 7 = 14

So the number is 14
```

**Listing 1.4.** Llama-3.2-1B at Step 1000 (Orca-Math, 8-bit quantized)

These examples illustrate that the private fine-tuning process, even when employing HE-compatible quantization, effectively adapts the models to the specific nuances, terminology, and problem-solving skills required by the target domain specified by the private data.

## 7 Conclusion

We presented and validated an interactive protocol using Homomorphic Encryption (HE) to enable privacy-preserving LoRA fine-tuning of open-source LLMs. By strategically outsourcing computations involving the public base model weights to an HE-enabled server while keeping private data and LoRA adapters on the client, our approach addresses the confidentiality challenge inherent in adapting LLMs to sensitive domains.

We demonstrated the feasibility of this method by fine-tuning the Llama-3.2-1B model. Our experiments confirmed that carefully chosen HE-compatible quantization can achieve convergence nearly identical to floating-point training, and the HE execution faithfully replicates the quantized cleartext dynamics. Performance benchmarks on a single GPU, while indicating significant computational cost ($\approx$ 216 s/token for a Llama-3.2-1B training step), highlighted the viability of our optimized HE implementation and demonstrated the extremely low client-side compute requirement ($\approx$ 0.025 MFLOP/s). This low client burden, combined with modest bandwidth needs, makes multi-server parallelization a practical and promising approach for scaling the HE computation to achieve acceptable training times.

This work provides a concrete pathway for securely leveraging private datasets to specialize LLMs in fields like healthcare or finance. While performance optimization and scaling remain crucial, our results show the potential of HE to unlock privacy-sensitive AI applications. Future work includes further HE optimizations, exploring efficient multi-server orchestration, and extending the approach to embedding layers.

## References


1. How to optimize a cuda matmul kernel for cublas-like performance: a worklog. https://siboehm.com/articles/22/CUDA-MMM, accessed: 2024-10-30
2. AI, M.: The llama 3 herd of models. arXiv preprint arXiv:2407.21783 (2024), https://arxiv.org/abs/2407.21783

3. Bergamaschi, F., Halevi, S., Halevi, T.T., Hunt, H.: Homomorphic training of 30,000 logistic regression models. In: Applied Cryptography and Network Security: 17th International Conference, ACNS 2019, Bogota, Colombia, June 5–7, 2019, Proceedings 17. pp. 592–611. Springer (2019)
4. Bonte, C., Vercauteren, F.: Privacy-preserving logistic regression training. BMC Medical Genomics 11(4), 86 (Oct 2018), https://doi.org/10.1186/s12920-018-0398-y
5. Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: TFHE: fast fully homomorphic encryption over the torus. Journal of Cryptology 33(1), 34–91 (2020)
6. Dettmers, T., Pagnoni, A., Holtzman, A., Zettlemoyer, L.: Qlora: Efficient finetuning of quantized llms. Advances in Neural Information Processing Systems 36 (2024)
7. Gao, C., Zhang, S.Q.: Dlora: Distributed parameter-efficient fine-tuning solution for large language model. arXiv preprint arXiv:2404.05182 (2024)
8. Goethals, B., Laur, S., Lipmaa, H., Mielikäinen, T.: On private scalar product computation for privacy-preserving data mining. p. 104–120. ICISC'04, Springer-Verlag, Berlin, Heidelberg (2004), https://doi.org/10.1007/11496618_9
9. Han, K., Hong, S., Cheon, J.H., Park, D.: Logistic regression on homomorphic encrypted data at scale. In: Proceedings of the AAAI conference on artificial intelligence. vol. 33, pp. 9466–9471 (2019)
10. Hao, M., Li, H., Chen, H., Xing, P., Xu, G., Zhang, T.: Iron: Private inference on transformers. Advances in neural information processing systems 35, 15718–15731 (2022)
11. Hu, E.J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., Chen, W.: LoRA: Low-rank adaptation of large language models. In: International Conference on Learning Representations (2022), https://openreview.net/forum?id=nZeVKeeFYf9
12. Huang, Z., jie Lu, W., Hong, C., Ding, J.: Cheetah: Lean and fast secure Two-Party deep neural network inference. In: 31st USENIX Security Symposium (USENIX Security 22). pp. 809–826. USENIX Association, Boston, MA (Aug 2022), https://www.usenix.org/conference/usenixsecurity22/presentation/huang-zhicong
13. Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., Adam, H., Kalenichenko, D.: Quantization and training of neural networks for efficient integer-arithmetic-only inference. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 2704–2713 (2018)
14. Joye, M.: On-line/off-line dcr-based homomorphic encryption and applications. In: Rosulek, M. (ed.) Topics in Cryptology – CT-RSA 2023. pp. 115–131. Springer International Publishing, Cham (2023)
15. Juvekar, C., Vaikuntanathan, V., Chandrakasan, A.: GAZELLE: A low latency framework for secure neural network inference. In: 27th USENIX Security Symposium (USENIX Security 18). pp. 1651–1669. USENIX Association, Baltimore, MD (Aug 2018), https://www.usenix.org/conference/usenixsecurity18/presentation/juvekar
16. Kim, A., Song, Y., Kim, M., Lee, K., Cheon, J.H.: Logistic regression model training based on the approximate homomorphic encryption. BMC Medical Genomics 11(4), 83 (Oct 2018), https://doi.org/10.1186/s12920-018-0401-7
17. Lou, Q., Feng, B., Charles Fox, G., Jiang, L.: Glyph: Fast and accurately training deep neural networks on encrypted data. Advances in neural information processing systems 33, 9193–9202 (2020)
18. Lyubashevsky, V., Peikert, C., Regev, O.: On ideal lattices and learning with errors over rings. In: Advances in Cryptology – EUROCRYPT 2010. Lecture Notes in Computer Science, vol. 6110, pp. 1–23. Springer (2010)
19. Montero, L., Frery, J., Kherfallah, C., Bredehoft, R., Stoian, A.: Machine learning training on encrypted data with tfhe. In: Proceedings of the 10th ACM International Workshop on Security and Privacy Analytics. p. 71–76. IWSPA '24, Association for Computing Machinery, New York, NY, USA (2024), https://doi.org/10.1145/3643651.3659891
20. Nandakumar, K., Ratha, N., Pankanti, S., Halevi, S.: Towards deep neural network training on encrypted data. In: Proceedings of the IEEE/CVF conference on computer vision and pattern recognition workshops. pp. 0–0 (2019)
21. Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: Stern, J. (ed.) Advances in Cryptology — EUROCRYPT '99. pp. 223–238. Springer Berlin Heidelberg, Berlin, Heidelberg (1999)
22. Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. In: Proceedings of the 37th Annual ACM Symposium on Theory of Computing (STOC 2005). pp. 84–93. ACM (2005)
23. Wang, Y., Lin, Y., Zeng, X., Zhang, G.: Privatelora for efficient privacy preserving llm. arXiv preprint arXiv:2311.14030 (2023)
24. Xiao, G., Lin, J., Seznec, M., Wu, H., Demouth, J., Han, S.: Smoothquant: Accurate and efficient post-training quantization for large language models. In: International Conference on Machine Learning. pp. 38087–38099. PMLR (2023)
25. Yang, L., Cai, S., Chai, D., Zhang, J., Tian, H., Jin, Y., Guo, K., Chen, K., Yang, Q.: Packvfl: Efficient he packing for vertical federated learning (2024), https://arxiv.org/abs/2405.00482

26. Zhang, C., Li, S., Xia, J., Wang, W., Yan, F., Liu, Y.: BatchCrypt: Efficient homomorphic encryption for Cross-Silo federated learning. In: 2020 USENIX Annual Technical Conference (USENIX ATC 20). pp. 493–506. USENIX Association (Jul 2020), https://www.usenix.org/conference/atc20/presentation/zhang-chengliang