nGraph-HE2: A High-Throughput Framework for Neural Network Inference on Encrypted Data

Fabian Boemer
fabian.boemer@intel.com
Intel AI Research
San Diego, California, USA

Anamaria Costache
anamaria.costache@intel.com
Intel AI Research
San Diego, California, USA

Rosario Cammarota
rosario.cammarota@intel.com
Intel AI Research
San Diego, California, USA

Casimir Wierzynski
casimir.wierzynski@intel.com
Intel AI Research
San Diego, California, USA

ABSTRACT

In previous work, Boemer et al. introduced nGraph-HE, an extension to the Intel nGraph deep learning (DL) compiler, that enables data scientists to deploy models with popular frameworks such as TensorFlow and PyTorch with minimal code changes. However, the class of supported models was limited to relatively shallow networks with polynomial activations. Here, we introduce nGraph-HE2, which extends nGraph-HE to enable privacy-preserving inference on standard, pre-trained models using their native activation functions and number fields (typically real numbers). The proposed framework leverages the CKKS scheme, whose support for real numbers is friendly to data science, and a client-aided model using a two-party approach to compute activation functions.

We first present CKKS-specific optimizations, enabling a 3x-88x runtime speedup for scalar encoding, and doubling the throughput through a novel use of CKKS plaintext packing into complex numbers. Second, we optimize ciphertext-plaintext addition and multiplication, yielding 2.6x-4.2x runtime speedup. Third, we exploit two graph-level optimizations: *lazy rescaling* and *depth-aware encoding*, which allow us to significantly improve performance.

Together, these optimizations enable state-of-the-art throughput of 1,998 images/s on the CryptoNets network. Using the client-aided model, we also present homomorphic evaluation of (to our knowledge) the largest network to date, namely, pre-trained MobileNetV2 models on the ImageNet dataset, with 60.4%/82.7% top-1/top-5 accuracy and an amortized runtime of 381 ms/image.

KEYWORDS

Privacy-Preserving Machine Learning; Deep Learning; Graph Compilers; Homomorphic Encryption

WAHC'19, November 11, 2019, London, United Kingdom

ACM ISBN 978-1-4503-6829-2/19/11…$15.00
https://doi.org/10.1145/3338469.3358944

ACM Reference Format:
Fabian Boemer, Anamaria Costache, Rosario Cammarota, and Casimir Wierzynski. 2019. nGraph-HE2: A High-Throughput Framework for Neural Network Inference on Encrypted Data. In *7th Workshop on Encrypted Computing & Applied Homomorphic Cryptography (WAHC'19), November 11, 2019, London, United Kingdom.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3338469.3358944

1 INTRODUCTION

The proliferation of machine learning inference as a service raises privacy questions and concerns. For example, a data owner may be concerned about allowing an external party access to her data.

Homomorphic encryption (HE) is an elegant cryptographic technology which can solve the data owner's concern about data exposure. HE is a form of encryption with the ability to perform computation on encrypted data, without ever decrypting it. In particular, HE allows for a data owner to encrypt her data, send it to the model owner to perform inference, and then receive the encrypted inference result. The data owner accesses the result of the inference by decrypting the response from the server.

The class of HE schemes known as leveled HE schemes or somewhat HE (SHE) schemes supports a limited number of additions and multiplications. As such, these schemes are attractive solutions to the DL based inference, whose core workload is multiplications and additions in the form of convolutions and generalized matrix multiplications (GEMM). One challenge in enabling HE for DL using SHE schemes is that we cannot compute non-linear functions, common in deep neural networks activations.

Another challenge in enabling HE for DL is the lack of support in existing frameworks. While popular DL frameworks such as TensorFlow [2] and PyTorch [35] have greatly simplified the development of novel DL methods, they do not support HE. Meanwhile, existing HE libraries such as Microsoft SEAL [39], HElib [25], and Palisade [37] are typically written at a level far lower than the primitive operations of DL. As a result, implementing DL models in HE libraries requires a significant engineering overhead.

nGraph-HE [6] introduced the first industry-class, open-source DL graph compiler which supports the execution of DL models through popular frameworks such as TensorFlow, MXNet, and PyTorch. Graph compilers represent DL models using a graph-based intermediate representation (IR), upon which hardware-dependent

and hardware-agnostic graph optimizations are performed. By treating HE as a virtual hardware target, nGraph-HE takes advantage of the graph compiler toolchain to create a framework for DL with HE. nGraph-HE uses Microsoft SEAL [39] for the underlying HE evaluation (with a framework for additional HE schemes), and nGraph [17] for the graph compiler IR. nGraph-HE enabled data scientists to use familiar DL frameworks; however, nGraph-HE supported only a limited class of models, restricted to polynomial activations.

In this work, we present nGraph-HE2[1], which introduces a number of optimizations in the graph compiler and the HE library. nGraph-HE2 utilizes a client-aided model, i.e. a hybrid approach using two-party computation, to execute a much wider class of pre-trained deep neural networks including non-polynomial activations with a focus on maximizing throughput. Our optimizations focus on inference on encrypted data with a plaintext model. We use batch-axis packing (Section 2.3.3) to enable a simple implementation of the Reshape operation and significantly increase throughput.

This setting is friendly to data scientists. It supports standard DL models,including non-polynomial activations. Since we do not rely on HE-specific training models, the training phase is HE-independent. Thus, data scientists can perform HE inference on standard DL models without cryptographic expertise.

A challenge specific to this data-scientist-friendly setting is that neural networks typically contain operations not suitable to all HE schemes, particularly in the activation functions. For instance, computing ReLU or MaxPool requires the comparison operator, which is not supported in the CKKS HE scheme. To this end, we use a protocol in which the server interacts with a client to perform non-polynomial operations such as ReLU (Section 4.1). Nevertheless, the CKKS scheme has several advantages, including support for floating-point numbers, plaintext packing, and faster runtime.

We present three main contributions. First, we describe optimizations to the CKKS encoding operations in SEAL (Section 3.1). We demonstrate a 3x-88x improvement in scalar encoding, and introduce *complex packing*, an optimization which doubles the inference throughput in networks without ciphertext-ciphertext multiplication (Section 3.1.2). Second, we introduce optimizations to ciphertext-plaintext addition, and ciphertext-plaintext multiplication, which apply in the batch-axis plaintext packing setting (Section 3.2). Third, we exploit two graph-level optimizations (Section 3.3). The first graph-level optimization, *lazy rescaling*, improves the runtime of higher-level operations such as Dot and Convolution[2] by delaying, hence minimizing the runtime spent on, the expensive rescaling operation. The second graph-level optimization, *depth-aware encoding*, minimizes the memory usage of the encoded model by encoding at the appropriate coefficient modulus level. Our *just-in-time encoding* implementation of depth-aware encoding encodes the values as late as possible.

We evaluate our contributions on both small, single-operation tests (Section 4.2), and on larger neural networks (Section 4.3). In particular, we demonstrate state-of-the-art performance on the CryptoNets network (Section 4.3.1), with a throughput of 1,998 images/s. Our contributions also enable the first, to our knowledge, homomorphic evaluation of a network on the ImageNet dataset, MobileNetV2, with 60.4%/82.7% top-1/top-5 accuracy and amortized runtime of 381 ms/image (Section 4.3.2). This is the first work showing the privacy-preserving execution of a full production-level deep neural network.

[1] nGraph-HE2 is available under the Apache 2.0 license at https://ngra.ph/he.

[2] Dot is a generalized dot product operation and Convolution is a batched convolution operation; see https://www.ngraph.ai/documentation/ops for more information.

2 BACKGROUND

2.1 Homomorphic Encryption

Homomorphic encryption (HE) enables computations to be carried out on encrypted data. We will focus on the FV scheme (sometimes referred to as BFV) [9, 22], as implemented in SEAL version 3.3 [39], with the CKKS optimizations [13, 14] and the relinearization technique with the special prime [12]. This is a somewhat HE (SHE) scheme, meaning that it supports a limited (and pre-determined) number of additions and multiplications. In contrast, fully homomorphic encryption (FHE) schemes support an unlimited number of additions and multiplications, typically by modifying an SHE scheme with an expensive bootstrapping step.

More concretely, if $\mathtt{ct}_1$ and $\mathtt{ct}_2$ are encryptions of m_1 and m_2, respectively, then

$$\mathtt{Dec}(\mathtt{ct}_1 + \mathtt{ct}_2) \approx m_1 + m_2, \quad \mathtt{Dec}(\mathtt{ct}_1 \cdot \mathtt{ct}_2) \approx m_1 \cdot m_2 \qquad (1)$$

The imprecision in the arithmetic is due to noise introduced during the computation, and can be controlled by setting encryption parameters appropriately. CKKS also offers ciphertext-plaintext operations, which are typically faster than ciphertext-ciphertext operations in Equation 1. That is, if $\mathtt{pt}_1$ is an encoding of m_1, then

$$\mathtt{Dec}(\mathtt{pt}_1 + \mathtt{ct}_2) \approx m_1 + m_2, \quad \mathtt{Dec}(\mathtt{pt}_1 \cdot \mathtt{ct}_2) \approx m_1 \cdot m_2$$

2.2 Mathematical Background

Many homomorphic encryption schemes, including CKKS, are based on the ring learning with error (RLWE) problem. A full description of the CKKS scheme and the RLWE problem is outside the scope of this paper. Instead, we provide a brief introduction to the CKKS scheme and refer the reader to [14, 33] for additional details.

Let $\Phi_M(X)$ be the M^{th} cyclotomic polynomial of degree $N = \phi(M)$. The plaintext space is the ring $\mathcal{R} = \mathbb{Z}[X]/(\Phi_M(X))$. We always take to be $\deg(\Phi_M(X))$ a power of two, typically 2,048 or 4,096. This is for both performance and security reasons.

2.2.1 Rescaling. In most HE schemes, a message is encrypted by adding noise. This noise grows with each homomorphic operation, especially multiplication. To manage this noise growth, CKKS introduces a rescaling operation which lowers the noise, and is typically performed after every multiplication.

We can only perform a (predetermined) limited number of such rescaling operations; therefore we can perform a (predetermined) number of multiplications. We let L be this number. Each multiplication represents a 'level' in our ciphertext space, and a rescaling operation lowers the level. To implement this, we have a 'layered' ciphertext space, where each layer has a different ciphertext modulus. We construct this space as follows. Let $p_1, \ldots, p_L$ be primes, and let p_{sp} be a 'special prime.' The ciphertext modulus is $q_L = \prod_{i=1}^{L} p_i$, yielding ciphertext space $\mathcal{R}_{q_L} = \mathcal{R}/(q_L\mathcal{R})$. Ciphertexts in the CKKS scheme are typically pairs of polynomials, i.e., $\mathtt{ct} \in \mathcal{R}_{q_L}^2$. The relinearization step (also referred to as the key-switching step) is

performed using the raise-the-modulus idea from [23] and the special modulus p_{sp}.

Encryption is performed using the special prime; this means a fresh ciphertext will be modulo $q_L \cdot p_{sp}$. We immediately perform a scale operation to reduce the level to that of q_L, so that the encryption algorithm's final output is an element of $\mathcal{R}_{q_L}$.

The rescaling algorithm is the homomorphic equivalent to the removing inaccurate LSBs as a rounding step in approximate arithmetic. More formally, we bring a ciphertext $\mathtt{ct}$ from level ℓ to ℓ' by computing

$$\mathtt{ct}' \leftarrow \left\lfloor \mathtt{ct} \frac{q_\ell}{q'_\ell} \right\rceil \tag{2}$$

where $q_\ell = \prod_{i=1}^{\ell} p_i$. Typically, rescaling is performed with $\ell' = \ell - 1$ after each multiplication to minimize noise growth. As such, the encryption parameter L is typically set to be at least L_f, i.e. $L \geq L_f$, the multiplicative depth of the function to compute.

2.2.2 Double-CRT Representation. To enable fast modular arithmetic modulo large integers, SEAL uses the residue number system (RNS) to represent the integers. To use this, we choose the factors q_i in $q_\ell = \prod_{i=1}^{\ell} p_i$ to be pairwise coprime, roughly of the same size and 64-bit unsigned integers (they are typically chosen to be of size 30-60 bits). Then, using the Chinese remainder theorem, we can write an element x in its RNS representation (also referred to as the CRT representation.) $(x \pmod{q_i})_i$. Each operation on x can be implemented by applying the operation on each element x_i. In particular, addition and multiplication of two numbers in RNS form are performed element-wise in $O(L)$ time, rather than $O(L \log L)$ time for multiplication, as would be required in a naive representation.

SEAL also implements the number-theoretic transform (NTT) for fast polynomial multiplication. Together, the CRT and NTT representation is known as the 'double-CRT' form. However, the NTT representation is incompatible with the rescaling operation. SEAL's rescaling operation requires performing an NTT^{-1}, the rescaling operation (2), then an NTT. The NTT and its inverse are relatively expensive computations, hence we will describe optimizations for avoiding them where possible (Section 3.3). A full description of the NTT is beyond the scope of this paper; see for example [32] for a cryptographer's perspective of the NTT.

2.2.3 Plaintext Packing. An extremely useful feature of the CKKS scheme is plaintext packing, also referred to as batching. This allows us to "pack" $N/2$ complex scalar values into one plaintext or ciphertext, where N is the cyclotomic polynomial degree. It works by defining an encoding map $\mathbb{C}^{N/2} \rightarrow \mathcal{R}$, where $\mathcal{R}$ is the plaintext space. An operation (addition or multiplication) performed on an element in $\mathcal{R}$ corresponds to the same operation performed on $N/2$ elements in $\mathbb{C}^{N/2}$. The number $N/2$ elements in the packing is also known as the number of *slots* in the plaintext.

Let $\mathcal{P} = \mathcal{R}$ refer to the plaintext space, and $C = \mathcal{R}^*_{q_L}$ refer to the ciphertext space.

2.3 HE for Deep Learning

The ability of HE to perform addition and multiplication makes it attractive to DL, whose core workloads are multiplication and addition in the form of convolutions and GEMM operations. However, neural networks commonly contain operations not suitable to all HE schemes, particularly in the activation functions. For instance, computing ReLU or MaxPool requires the comparison operator, which is not supported in all SHE schemes. At a high level, therefore, there are two broad approaches to enabling homomorphic evaluation of a given DL model:

(1) *HE-friendly networks*: Modify the network to be HE-friendly, and re-train.
(2) *Pre-trained networks*: Modify the HE scheme or protocol to accommodate the network as is, ideally with no retraining.

2.3.1 HE-friendly Networks. In this setting, we assume (and require) that the data scientist has access to the entire DL workflow, including training. Here, the network is re-trained with polynomial activations, and max-pooling is typically replace with average pooling. Low-degree polynomials may be used, as high-degree polynomials result in prohibitively large encryption parameters due to the large multiplicative depth. The CryptoNets network [24] is the seminal HE-friendly network, using the $f(x) = x^2$ activation function to achieve ≈99% accuracy on the MNIST [30] handwritten digits dataset. However, on larger datasets, the accuracy of HE-friendly networks suffers. CHET [19] adopts a similar approach on the CIFAR10 [29] dataset, instead using activation functions $f(x) = ax^2 + bx$, with $a, b, \in \mathbb{R}$. This approach results in 81.5% accuracy, down from 84% accuracy in the original model with ReLU activations. Hesamifard et al. [26] see a similar drop-off in accuracy from 94.2% to 91.5% on the CIFAR10 dataset. Depending on the use case, such a drop in accuracy may not be acceptable.

From a practical viewpoint, HE-friendly networks tend to be more difficult to train than their native counterparts. In particular, polynomial activations are unbounded and grow more quickly than standard activation functions such as ReLU or sigmoid, resulting in numerical overflow during training. Possible workarounds include weight and activation initialization and gradient clipping.

Sparsification methods, such as in SEALion [40] and Faster CryptoNets [16] improve latency by reducing the number of homomorphic additions or multiplications. This is an optimization mostly independent of HE.

2.3.2 Pre-trained Networks. In this setting, we assume a network has been trained, and no modifications are possible. This setting results in independent training and inference tasks. In particular, data scientists need not be familiar with HE to train privacy-preserving models. Additionally, this setting preserves the accuracy of the existing models, which tend to be higher than models built with HE-friendly constraints. Two solutions to the pre-trained network setting are FHE schemes and hybrid schemes.

FHE schemes. FHE schemes enable an unlimited number of additions and multiplications, allowing for arbitrary-precision polynomial approximations of non-polynomial activations. However, due to the expensive bootstrapping step used to cope with increasing the computational depth, this approach is typically much slower than alternatives. Some FHE schemes, such as TFHE [15] operate

on boolean circuits which support low-depth circuits for exact computation of ReLU. However, performance on arithmetic circuits, such as GEMM operations, suffers.

Wang, et al. [42] propose using Intel Software Guard Extensions (SGX) to implement a bootstrapping procedure in a trusted execution environment (TEE). In effect, this approach turns a SHE scheme to a FHE scheme with a lower performance penalty than FHE bootstrapping. However, it loosens the security model, as the TEE must be trusted.

Hybrid schemes. Hybrid schemes combine privacy-preserving primitives, such as HE and multi-party computation (MPC). In MPC, several parties follow a communication protocol to jointly perform the computation. MPC techniques, such as garbled circuits (GCs), typically support a broader range of operations than HE, while introducing a communication cost between the parties. Hybrid HE-MPC schemes therefore provide an elegant solution to the pre-trained network setting by using MPC to perform non-polynomial activations, and HE to perform the FC and Convolution layers.

This approach has two important benefits. First, it enables exact computation, mitigating the performance drop-off in HE-friendly networks. Second, it enables smaller encryption parameters. The HE-MPC interface involves refreshing the ciphertext at each non-polynomial activation, i.e. resetting the noise budget and coefficient modulus to the highest level L. This resetting reduces the effective multiplicative depth of the computation to the number of multiplications between non-polynomial activations. As a result, L is quite small, even for large networks. For instance, $L = 3$ suffices for the MobileNetV2 network [38] (Section 4.3.2). Smaller L also enables choice of smaller polynomial modulus degree, which greatly reduces the runtime (see Appendix A.4) and memory usage.

Several hybrid schemes have been developed. Chimera [7] is a hybrid HE-HE scheme which performs ReLU in TFHE, and affine transformations in an arithmetic-circuit HE scheme such as FV or CKKS. However, the translation between TFHE and FV/CKKS is potentially expensive. MiniONN [31] is a hybrid HE-MPC scheme which uses an additive HE scheme to generate multiplication triples, which are used in an MPC-based evaluation of the network. Gazelle [27] uses HE to perform the polynomial functions and GCs to perform the non-polynomial activations.

Other schemes. A third solution to the pre-trained network setting is pure MPC schemes. ABY [20] supports switching between arithmetic, boolean, and Yao's GCs. ABY3 [34] increases the performance of ABY by introducing a third party. SecureNN [41] likewise increases performance at the cost of a third party. Some two-party MPC schemes also have shortcomings, such as requiring binarizing the network [1]. Our work, in contrast, supports full-precision networks using standard data types.

2.3.3 *Challenges in deploying DL on HE.*

Software Frameworks. One difficulty in enabling HE for DL is the lack of support in existing frameworks. While popular DL frameworks such as TensorFlow [2] and PyTorch [35] have greatly simplified the development of novel DL methods, they do not support HE. Existing HE libraries such as Microsoft SEAL [39], HElib [25], and Palisade [37] are typically written at a low level. As such, implementing DL models requires a significant engineering overhead. nGraph-HE [6] introduces a DL graph compiler which supports execution of DL models through popular frameworks such as TensorFlow, MXNet, and PyTorch.

Performance Considerations. One of the primary shortcomings of HE is the large computational and memory overhead compared to unencrypted computation, which can be several orders of magnitude. The choice of encryption parameters, N and the coefficient moduli q_i, has a large impact on this overhead, as well as the security level (see Appendix A.4). As such, parameter selection, which remains a largely hand-tuned process, is vital for performance.

Mapping to DL Functions. Another difficulty in enabling HE for DL is the mapping from HE operations to DL operations. While HE addition and multiplication map naturally to plaintext addition and multiplication, there are various choices for plaintext packing (see Section 2.2). Both CryptoNets [24] and nGraph-HE [6] use plaintext packing along the batch axis (*batch-axis packing*) to store a 4D tensor of shape (S, C, H, W) (batch size, channels, height, width) as a 3D tensor of shape (C, H, W), with each ciphertext packing S values. Each model weight is stored as a plaintext with the same value in each slot (encoded using scalar encoding, see Section 3.1.1). Since HE addition and multiplication are performed element-wise on each slot, this enables inference on up to S data items simultaneously, where the runtime for one data item is the same as for S data items (for $S \leq N/2$, the slot count). As a result, this use of plaintext packing greatly increases throughput for a given latency.

Other approaches such as Gazelle [27] and LoLa [10] use *inter-axis packing*, a choice of plaintext packing which encrypts multiple scalars from the same inference data item or weight matrix to the same ciphertext. Inter-axis packing optimizes inference on one data item at a time, with latency scaling linearly with the batch size. However, DL workloads on inter-axis packing often use HE rotations, which are relatively expensive (see Appendix A.4). The optimal packing approach depends on the workload, and can be determined by graph compilers. nGraph-HE2 uses batch-axis packing.

2.4 Graph Compilers

Graph compilers represent DL models with a graph-based intermediate representation (IR). The primary advantage to a graph-based IR is the enabling of graph-based optimizations, which can be either hardware agnostic or hardware dependent. Intel nGraph [17] is a DL graph compiler which optimizes the inference graph for several hardware targets. A second advantage to graph-based IR is the ability to represent models from different DL frameworks in a common IR; thus, the graph-based optimizations are framework-agnostic. nGraph-HE [6] introduces the first DL framework for HE. nGraph-HE treats HE as a virtual hardware target and uses Microsoft SEAL [39] for the underlying HE evaluation, as well as a simple structure for adding other HE libraries. In addition to graph-based optimizations, nGraph-HE provides run-time based optimizations based on the values of the plaintext model.

CHET [19] is another graph-based compiler for HE. It uses inter-axis packing to optimize the layout of each tensor, as opposed to using batch-axis packing for every tensor, as in nGraph-HE. SEALion [40] uses a graph compiler for automatic parameter selection, while lacking packing and value-based runtime optimizations.

3 CONTRIBUTIONS

We introduce the following contributions, which apply in the batch-axis packing setting:

- CKKS encoding optimizations;
- CKKS arithmetic optimizations;
- graph-level optimizations.

The CKKS encoding optimizations include faster scalar encoding, and *complex packing*, which doubles the throughput by taking advantage of the complex components of the plaintext encoding map. Our arithmetic optimizations apply to ciphertext-plaintext addition, and ciphertext-plaintext multiplication. The graph-level optimizations include *lazy rescaling* and *depth-aware encoding*, which reduce the runtime spent rescaling and encoding, respectively.

3.1 CKKS Encoding Optimizations

3.1.1 Scalar Encoding. Plaintext packing enables the encoding of $N/2$ complex scalars into a single plaintext. For more efficient addition and multiplication, SEAL stores each plaintext in double-CRT form (performing an NTT on the polynomial, and storing each coefficient in RNS form with respect to the p_i). At the top level, (with L coefficient moduli), encoding requires $O(LN)$ memory and $O(LN \log N)$ runtime. Algorithm 1 shows the pseudocode for general encoding, including the NTT.

Algorithm 1 General CKKS Encoding

function EncodeVector($c \in \mathbb{C}^{N/2}, q \in \mathbb{Z}, s \in \mathbb{R}$)
  $p \in \mathbb{C}^N$
  $p[0 : N/2] \leftarrow c$
  $p[N/2 + 1 : N] \leftarrow c^*$
  $p \leftarrow \text{DFT}^{-1}(p \cdot s)$
  $p \leftarrow [p]_q$
  $p \leftarrow \text{NegacyclicNTT}(p)$
end function

SEAL additionally provides an optimized encoding algorithm in the setting where the $N/2$ scalars are the same real-valued number. This setting yields a simplified DFT^{-1} and NTT, resulting in an implementation requiring $O(LN)$ runtime and memory. Both of SEAL's encoding implementations are general, that is they allow arbitrary operations on the resulting plaintext.

Here, we optimize for the more restrictive case in which $N/2$ identical real-valued scalars are stored in the plaintext, for the entire lifetime of the plaintext. Our use of batch-axis packing (see Section 3.1.2) maintains this property on the plaintexts, since they are used only for plaintext addition and multiplication. Other plaintext packing schemes, such as inter-axis packing (see Section 2.3.3), however, do not maintain this property. Thus, scalar encoding applies only in specific use-cases, including batch-axis packing.

Our optimization takes advantage of the fact that the general CKKS encoding algorithm of $N/2$ identical real-valued scalars will result in a plaintext with N identical values across the slots. See Appendix A.3 for the proof of this property. So, rather than store N copies of the same scalar, we modify the plaintext to store just a single copy. This improves the memory usage and runtime each by a factor of N, yielding $O(L)$ runtime and memory usage. Algorithm 2 shows the pseudocode for the scalar-optimized encoding algorithm.

Algorithm 2 CKKS Scalar encoding of c with respect to modulus q at scale s

function EncodeReal($c \in \mathbb{R}, q \in \mathbb{Z}, s \in \mathbb{R}$)
  $y \in \mathbb{R}$
  $y \leftarrow [s \cdot c]_q$
  return y
end function

Note, SEAL implements a variant of Algorithm 2 for scalar encoding; however it computes $y \in \mathbb{R}^N$, with $y_i \leftarrow [s \cdot c]_q \ \forall i$, requiring memory and runtime $O(LN)$. For comparison, Algorithm 2 avoids the expensive copy of size N, decreasing the runtime compared to SEAL's implementation.

3.1.2 Complex packing. We introduce *complex packing*, an optimization which doubles the inference throughput in cases without ciphertext-ciphertext multiplication. One of the primary ways to combat the large runtime and memory overhead of HE is to use plaintext packing in the CKKS encoding mapping $\mathbb{C}^{N/2} \to \mathcal{R}$ (Section 2.2.3). Neural network models, however, typically operate on real numbers. As such, packing real-valued model weights or data values utilizes at most half of the available computation. Complex packing, on the other hand, utilizes the entire computational capacity of plaintext packing.

For simplicity, let $N = 4$, so each plaintext and ciphertext encodes two complex scalars. Given scalars $a, b, c, d, f, g, h, k \in \mathbb{R}$, let:

- $REnc(a, b) = p(a, b)$ represent the plaintext encoding with a in the first slot and b in the second slot.
- $CEnc(a, b, c, d) = p(a + bi, c + di)$ encode $a + bi$ in the first slot, and $c + di$ in the second slot.
- $RDec(p(a, b)) = (a, b)$
- $CDec(p(a + bi, c + di)) = (a, b, c, d)$

Let *real packing* refer to the $REnc/RDec$ representation, and *complex packing* refer to the $CEnc/CDec$ representation. Then, let

	$p(a + bi,$	$c + di)$
	$p(f + gi,$	$h + ki)$
$\xrightarrow{\pm}$	$p(a \pm f + (b \pm g)i,$	$c \pm h + (d \pm k)i)$
$\xrightarrow{\times}$	$p(af - bg + (ag + bf)i,$	$ch - dk + (ck + dh)i)$

represent element-wise (real) addition/subtraction and multiplication, respectively. Note, a given implementation of a plaintext may not represent the plaintext slots internally as complex numbers. SEAL, for instance, uses 64-bit unsigned integers. Instead, our presentation serves to illustrate the concept, which is independent of the HE library's specific implementation. Though we only consider plaintexts here, the same logic also holds for ciphertexts.

Now, we consider the following element-wise computations:

- Add/Subtract:

$$\begin{aligned}(a, b, c, d) \pm (f, g, h, k) &= (a \pm f, b \pm g, c \pm h, d \pm k) \\ &= CDec(CEnc(a, b, c, d) \pm CEnc(f, g, h, k))\end{aligned}$$

- Broadcast-Multiply:

$$(a,b,c,d) \times f = (af, bf, cf, df)$$
$$= CDec(CEnc(a,b,c,d) \times CEnc(f,0,f,0))$$

- Multiply:

$$(a,b,c,d) \times (f,g,h,k) = (af, bg, ch, dk))$$
$$\neq CDec(CEnc(a,b,c,d) \times CEnc(f,g,h,k))$$

So, we observe each operation except Multiply[3] can be represented using complex packing. Furthermore, we can compose any number of Add, Subtract, and Broadcast-Multiply, operations represented using complex packing. Note, real packing supports these operations, as well as Multiply. However, real packing requires twice the number of slots, i.e. two plaintexts, or doubling N.

These properties easily generalize to larger N. In essence, complex packing can perform the same computation (as long as it does not include a ciphertext-ciphertext Multiply operation) as the real packing representation on twice as many slots.

Now, following nGraph-HE, nGraph-HE2 uses batch-axis plaintext packing (Section 2.3.3) during inference to store a 4D inference tensor of shape (S, C, H, W) (batch size, channels, height, width) a 3D tensor of shape (C, H, W), with each ciphertext packing S values. Each model weight is stored as a plaintext with the same value in each slot (encoded using scalar encoding, see Section 3.1.1). Hence, in a neural network, the FC and Convolution layers consist of only Add, Subtract, and Broadcast-Multiply operations, suitable for complex packing. Polynomial activations such as $f(x) = x^2$, in contrast, are not suited for complex packing since they require ciphertext-ciphertext multiplications. However, ciphertext-ciphertext multiplications are absent in many neural networks with ReLU activations. For these networks, complex packing doubles the throughput.

Kim and Song [28] also propose complex packing, by modifying the underlying HE scheme. Bergamaschi et al. [4] use a similar complex packing idea to train logistic models in a genome-wide association study (GWAS), with limited speedup due to the requirement of additional conjugation operations. Our use of complex packing, on the other hand, applies to neural network inference, and nearly doubles the throughput.

3.2 CKKS Arithmetic Optimizations

We introduce optimizations to ciphertext-plaintext addition and multiplication in CKKS, which apply in the special case of batch-axis packing. A further ciphertext-plaintext multiplication optimization applies when the coefficient modulus is less than 32 bits.

3.2.1 Ciphertext-plaintext Addition. Ciphertext-plaintext addition in RNS form requires element-wise addition of two polynomials in which each sum is reduced with respect to the coefficient modulus p_ℓ. With our scalar encoding approach, we instead perform summation of the same scalar with each element of a polynomial. Algorithm 3 shows the ciphertext-plaintext vector algorithm, compared to Algorithm 4, which shows the optimized ciphertext-plaintext scalar addition algorithm. Both implementations require $O(LN)$ memory and runtime, however Algorithm 4 is more cache-friendly.

Algorithm 3 Ciphertext-Plaintext Vector Addition

```
1: function Add Cipher-Plain Vector(ct ∈ C, pt ∈ P)
2:     for ℓ = 1 to L do
3:         for n = 1 to N do
4:             ct[ℓ][n] ← (ct[ℓ][n] + pt[ℓ][n]) mod p_ℓ
5:         end for
6:     end for
7: end function
```

Algorithm 4 Ciphertext-Plaintext Scalar Addition

```
1: function Add Cipher-Plain Scalar(ct ∈ C, pt ∈ P)
2:     for ℓ = 1 to L do
3:         tmp ← pt[ℓ]
4:         for n = 1 to N do
5:             ct[ℓ][n] ← (ct[ℓ][n] + tmp) mod p_ℓ
6:         end for
7:     end for
8: end function
```

Note that the same optimization works for ciphertext-plaintext subtraction, and we expect similar improvements.

3.2.2 Ciphertext-plaintext Multiplication. Ciphertext-plaintext multiplication in RNS form requires element-wise multiplication of two polynomials in which each product is reduced with respect to the coefficient modulus q_l. The modulus reduction is performed with Barrett reduction [3]. We present two optimizations.

First, our scalar encoding allows us to perform multiplication between a scalar and each element of the polynomial, rather than between two polynomials. This is the same optimization as in ciphertext-plaintext addition.

Second, we provide an optimization for the case in which the coefficient modulus is 32 bits, rather than 64 bits. The benefit arises from a simpler implementation of Barrett reduction which requires fewer additions and multiplications. In SEAL, ciphertext and plaintext elements are stored at 64-bit unsigned integers, with a maximum modulus of 62 bits [39]. As a result, performing the multiplication may overflow to 128 bits. Then, performing Barrett reduction requires 5 multiplications, 6 additions, and 2 subtractions (including the conditional subtraction). See Algorithm 5 for the pseudocode, which closely follows SEAL's implementation[4]. We store an unsigned 128-bit number z as two unsigned 64-bit numbers with $z[0]$ containing the 64 low bits and $z[1]$ containing the 64 high bits. The *add*64 function will return the carry bits of the addition.

In the case where q is a 32-bit modulus, the Barrett reduction becomes much simpler, requiring just 2 multiplications and 2 subtractions (including the conditional subtraction). Algorithm 6 shows the pseudocode for the more efficient Barrett reduction, which closely follows SEAL's implementation[5]. Algorithm 7 shows the general,

[3] due to the cross-terms in $(a+bi)\times(f+gi) = af - bg + (ag+bf)i \neq af + bgi$. Note, it is an open problem to compute and add the correction term $bg + (bg - ag - bf)i$. This is non-trivial because in this setting a, b are encrypted, and we can only use complex multiplication to compute the cross-term.

[4] https://github.com/microsoft/SEAL/blob/3.3.0/native/src/seal/util/uintarithsmallmod.h#L146-L187

[5] https://github.com/microsoft/SEAL/blob/3.3.0/native/src/seal/util/uintarithsmallmod.h#L189-L217

Algorithm 5 BarrettReduction128

function BarrettReduction128(128-bit number z, 64-bit modulus q, 128-bit Barrett ratio r)
  $uint64\ tmp1, tmp2[2], tmp3, carry$
  $carry \leftarrow mult\_hw64(z[0], r[0])$ ▷ Multiply low bits
  $tmp2 \leftarrow z[0] * r[1]$
  ▷ Compute high bits of $z[0] * r$
  $tmp3 \leftarrow tmp2[1] + add64(tmp2[0], carry, \&tmp1)$
  $tmp2 \leftarrow z[1] * r[0]$
  $carry \leftarrow tmp2[1] + add64(tmp1, tmp2[0], \&tmp1)$
  $tmp1 \leftarrow z[1] * r[1] + tmp3 + carry$ ▷ Compute $[z * r]_{2^{128}}$
  $tmp3 \leftarrow z[0] - tmp1 * q$ ▷ Barrett subtraction
  if $tmp3 \geq q$ **then** ▷ Conditional Barrett subtraction
    $result \leftarrow tmp3 - q$
  else
    $result \leftarrow tmp3$
  end if
  return $result$
end function

64-bit modulus implementation of ciphertext-plaintext multiplication. Note, SEAL uses Barrett64 reduction for rescaling, whereas we use it for optimized ciphertext-plaintext multiplication.

Algorithm 8 shows the optimized 32-bit modulus implementation of multiplication with a scalar plaintext. Note, the plaintext `pt` contains only L entries, rather than $N \cdot L$ entries. Algorithm 7 and Algorithm 8 both require $O(LN)$ runtime; however, Algorithm 8 is more cache-friendly.

Algorithm 6 BarrettReduction64

function BarrettReduction64(64-bit number z, 32-bit modulus q, 64-bit Barrett ratio r)
  $uint64\ carry$
  $carry \leftarrow mult\_hw64(z, r)$ ▷ Compute $[z \cdot q]_{2^{64}}$
  $carry \leftarrow z - carry * q$ ▷ Barrett subtraction
  if $carry \geq q$ **then** ▷ Conditional Barrett subtraction
    $result \leftarrow carry - q$
  else
    $result \leftarrow carry$
  end if
  return $result$
end function

3.3 Graph-level Optimizations

In addition to the above low-level CKKS optimizations, we present two graph-level optimizations.

3.3.1 Lazy Rescaling. Rescaling in CKKS can be thought of as a procedure which homomorphically removes the inaccurate LSBs in the (encrypted) message. See Section 2.2 for a longer description, or [14] for full details. Due to the NTT and NTT^{-1}, rescaling is ≈9x more expensive than ciphertext-plaintext multiplication in SEAL (see Appendix A.4). The *naive rescaling* approach rescales after every multiplication. *Lazy rescaling*, on the other hand, minimizes the number of rescaling operations by:

Algorithm 7 Ciphertext-Plaintext 64-bit Multiplication

function Multiply Cipher-Plain 64-bit(`ct` $\in C$, `pt` $\in \mathbb{Z}^{L \times N}$, 128-bit Barrett ratio r)
  for $\ell = 1$ to L **do**
    for $n = 1$ to N **do**
      $uint64\ z[2];$
      $z \leftarrow \texttt{ct}[\ell][n] * \texttt{pt}[\ell][n]$ ▷ Perform multiplication
      $\texttt{ct}[\ell][n] \leftarrow BarrettReduction128(z, q_\ell, r)$
    end for
  end for
end function

Algorithm 8 Ciphertext-Plaintext Scalar 32-bit Multiplication

function Multiply Cipher-Plain 32-bit(`ct` $\in C$, `pt` $\in \mathbb{Z}^{L}$, 64bit Barrett ratio r)
  for $\ell = 1$ to L **do**
    $tmp \leftarrow \texttt{pt}[\ell]$
    for $n = 1$ to N **do**
      $uint64\ z;$
      $z \leftarrow \texttt{ct}[\ell][n] * tmp$ ▷ Perform multiplication
      $\texttt{ct}[\ell][n] \leftarrow BarrettReduction64(z, q_\ell, r)$
    end for
  end for
end function

- rescaling only after a Fully-Connected (FC) or Convolution layer, rather than after every multiplication therein;
- skipping rescaling if there are no subsequent multiplications before the ciphertext is decrypted.

Since FC and Convolution layers each contain several multiplications per output element, the first optimization reduces the number of rescaling operations performed by a factor of the inner dimension (for FC layers) or window size (for Convolution layers).

The second optimization ensures rescaling happens only when reducing the message scale is necessary. In particular, addition is allowed post-multiplication, pre-rescaling. In the case where the last two layers of a network (or before a ReLU activation in a hybrid MPC-HE scheme, see Section 2.3.2) are FC-Add, Convolution-Add or Multiply-Add, this ensures the rescaling is omitted entirely. Note, for a choice of parameters with $L = L_f$, where L_f is the multiplicative depth of the function, this optimization is equivalent to skipping rescaling to level 0. Table 1 shows an example where the second optimization results in a skipped rescaling. Note, lazy rescaling applies 'Use rescaling sparingly' from [5], to neural network inference instead of a genome-wide association study (GWAS). The GWAS setting has a closed-form semi-parallel logistic regression model, whereas our setting involves long sequences of linear and non-linear operations on tensors, e.g. convolutions, and pooling operations.

3.3.2 Depth-aware Encoding. The runtime complexity and memory usage of encoding a scalar at level ℓ in SEAL are both $O(N\ell)$ (see Section 3.1.1). Throughout the execution of HE operations, the level ℓ decreases due to the rescaling operation (see Section 2.2.1). When multiplying or adding a ciphertext `ct` at level $\ell < L$ with a

Table 1: Benefit of lazy rescaling at level 0. Lazy rescaling skips the rescaling, whereas naive rescaling performs an unnecessary rescaling.

Operation	Number of rescalings	
	Naive Rescaling	Lazy Rescaling
Constant	$L-1$	$L-1$
Multiply	L	$L-1$
Add	L	$L-1$

plaintext pt, it is therefore advantageous to encode pt at level ℓ rather than level L, as noted by the 'Harnessing the CRT ladder' technique in [5]. This will reduce both the runtime and memory usage of the encoding step. In practice, this implementation can have two forms:

(1) *Compile-time encoding.* An optimization pass through the computation graph can identify the level at which each plaintext is encoded. This compilation step requires a larger initial memory overhead, for the benefit of increased runtime.
(2) *Lazy encoding.* In this implementation, the plaintext model weights are stored in native scalar (i.e. floating-point) format, and encoding is delayed until immediately preceding multiplication or addition with a ciphertext ct. The level ℓ at which to encode the model weight is determined by observing the level of ct.

If encoding is expensive compared to addition/multiplication (as in SEAL, see Appendix A.4), compile-time encoding yields the fastest runtime. However, due to the choice batch-axis packing, nGraph-HE2's scalar encoding (Section 3.1.1) is significantly cheaper than addition/multiplication, requiring runtime and memory $O(\ell)$, compared to $O(\ell N \log N)$ runtime and $O(\ell N)$ memory usage of general encoding. Hence, performing lazy encoding at runtime results in little slowdown, and allows for a simpler implementation.

Here, we introduced CKKS-specific optimizations to scalar encoding, ciphertext-plaintext addition, and ciphertext-plaintext multiplication in the batch-axis packing case. We also introduced graph-level optimizations of complex packing and depth-aware encoding.

4 EVALUATION

We evaluate our optimizations on small, single-operation tests (Section 4.2), as well as on larger neural networks (Section 4.3). All results are computed on Intel Xeon® Platinum 8180 2.5GHz systems with 376GB of RAM and 112 cores, running Ubuntu 16.04. The localhost bandwidth is 46.2 Gbit/s, and the local area network (LAN) bandwidth is 9.4 Gbit/s. We use GCC 7.4 with -O2 optimization.

4.1 Client-aided Model

To mitigate the classification accuracy degradation of HE-friendly networks (Section 2.3), we implement a simple two-party computation approach. Specifically, evaluate a non-polynomial function f on a ciphertext ct, the server sends ct to the client, which decrypts $Dec(\texttt{ct}) \rightarrow \texttt{pt}$, computes $f(\texttt{pt})$, and sends a fresh encryption of the result, $Enc(f(\texttt{pt}))$ to the server. This approach accomplishes two tasks: first, it enables the computation of non-polynomial functions; second, it refreshes the ciphertext, i.e. resets the noise budget and coefficient modulus to the highest level L. However, this approach can leak information about the model to the client, as it provides the pre-activation values pt to the client, as well as the activation function itself. One possible improvement is performing the non-polynomial function using additive masking and garbled circuits, as in Gazelle [27]. Another approach is to perform the decryption, non-polynomial activation, and encryption in a trusted execution environment (TEE) attested to the user, such as Intel's Software Guard Extensions (SGX) [11]. For instance, Wang et. al [42] use Intel's SGX for bootstrapping only, though this approach is easily adapted to perform the non-polynomial activation as well.

Our client-aided approach, therefore, represents a placeholder for more secure implementations in future work. Since the client-aided model refreshes ciphertexts at each non-polynomial layer, the effective multiplicative depth is reduced to the multiplicative depth between non-polynomial layers. This enables the computation of arbitrarily-deep neural networks with much smaller encryption parameters, and therefore much faster runtimes.

Table 2: Runtime and memory usage when encoding a scalar, with and without optimization (Opt.). Runtimes are averaged across 1000 trials.

N	L	Opt.	Memory		Runtime	
			Usage (bytes)	Improv.	Time (ns)	Speedup
2^{12}	1	✗	32,768		605	
2^{12}	1	✓	8	4,096	177	3.4
2^{13}	3	✗	196,608		2,951	
2^{13}	3	✓	24	8,192	202	14.6
2^{14}	8	✗	1,048,576		38,938	
2^{14}	8	✓	64	16,384	443	87.9

4.2 Low-level Operations

4.2.1 Scalar Encoding. We implement the scalar encoding optimization from Section 3.1.1. Table 2 shows the speedup of several parameter choices, each satisfying $\lambda = 128$-bit security. As expected, the memory improvement is a factor of N. The runtime improvement of scalar encoding increases with N, due to the $O(L)$ runtime, compared to $O(LN \log N)$ in the general encoding.

4.2.2 Ciphertext-plaintext Addition. We implement the scalar addition optimization from Section 3.2.1. Table 3 shows a speedup of 2.6x-4.2x, with more speedup for larger encryption parameters. Algorithm 3 and Algorithm 4 both have $O(LN)$ runtime complexity. The primary source of speedup, therefore, is due to the fact that one of the two operands, the scalar, is kept into the processor registers, and the other operand does not have to compete for its placement and retrieval from the cache memory.

4.2.3 Ciphertext-plaintext Multiplication. We implement the scalar multiplication optimization from Section 3.2.2. Table 4 shows a speedup of 2.6x for different parameter choices. Notably, the parameters uses 30-bit coefficient moduli, so our Barrett reduction optimization applies.

Table 3: Runtime improvement in ciphertext-plaintext scalar addition. Parameter choices satisfy $\lambda = 128$-bit security. Runtimes are averaged across 1000 trials.

N	L	Runtime (µs)		
		General	Scalar	Speedup
2^{12}	1	2.3	0.9	2.6
2^{13}	3	12.6	4.5	2.8
2^{14}	8	124.5	30.0	4.2

Table 4: Runtime improvement in ciphertext-plaintext scalar multiplication. Parameter choices satisfy $\lambda = 128$-bit security. Runtimes are averaged across 1000 trials.

N	L	Runtime (µs)		
		General	Scalar	Speedup
2^{13}	3	181.7	71.1	2.6
2^{14}	8	966.6	377.6	2.6

Table 5: Impact of lazy rescaling on CryptoNets runtime using $N = 2^{13}, L = 6$, with accuracy 98.95%.

Thread Count	Lazy Rescaling	Runtime	
		Amortized (ms)	Total (s)
1	✗	59.21	242.51 ± 3.69
1	✓	7.23	29.62 ± 0.63
24	✓	0.50	2.05 ± 0.11

4.3 Neural Network Workloads

To evaluate our graph-level optimizations and complex packing, we evaluate two neural networks: the standard CryptoNets [24] model, and MobileNetV2 [38]. To the best of our knowledge, this is the largest network whose linear layers have been homomorphically evaluated, as well as the first homomorphic evaluation of a network on the ImageNet dataset.

4.3.1 CryptoNets. The CryptoNets network [24] is the seminal HE-friendly DL model for the MNIST handwritten digits dataset [30]. The architecture uses $f(x) = x^2$ for the activations, and has a multiplicative depth of 5. See Appendix A.1 for the full architecture. As in [24], we achieve 98.95% accuracy. Table 5 shows lazy rescaling reduces the runtime of the CryptoNets network by ≈8x. Multi-threading further improves the performance (see Appendix A.2).

In order to show the benefits of complex packing (Section 3.1.2), we implement the client-aided model (Section 4.1). We train the CryptoNets network with ReLU activations rather than x^2 activations, and add bias terms. See Appendix A.1 for the complete architecture. The use of ReLU activations effectively decreases the multiplicative depth to 1, since the client-aided computation of ReLU refreshes the ciphertexts. This lower multiplicative depth enables much smaller encryption parameters, $N = 2^{11}, L = 1$, with a single 54-bit coefficient modulus.

Table 6: Impact of complex packing on CryptoNets with ReLU activations using $N = 2^{11}, L = 1$, and 98.62% accuracy. Results are averaged over 10 trials. Amt. times are amortized over the largest batch size supported.

Thread Count	Complex packing	Network setting	Batch size	Runtime	
				Amt. (ms)	Total (s)
1	✗	localhost	1,024	2.72	2.79 ± 0.06
1	✓	localhost	2,048	1.44	2.94 ± 0.04
24	✓	localhost	2,048	0.24	0.50 ± 0.04
24	✓	LAN	2,048	0.34	0.69 ± 0.04

Table 7: CryptoNets performance comparison, including accuracy (Acc.), latency (Lat.), and throughput (Thput.). For hybrid protocols, latency is reported in the LAN setting and communication (Comm.) includes only the interactive part of the protocol.

Method	Acc. (%)	Lat. (s)	Thput. (im/s)	Protocol	Comm. (MB/im)
LoLa [10]	98.95	2.2	0.5	HE	
FHE-DiNN100 [8]	96.35	1.65	0.6	HE	
CryptoNets [24]	98.95	250	16.4	HE	
Faster CryptoNets [16]	98.7	39.1	210	HE	
nGraph-HE [6]	98.95	16.7	245	HE	
CryptoNets 3.2 [10]	98.95	25.6	320	HE	
nGraph-HE2	98.95	2.05	1,998	HE	
Chameleon [36]	99	2.24	1.0	HE-MPC	5.1
MiniONN [31]	98.95	1.28	2.4	HE-MPC	44
Gazelle [27]	98.95	0.03	33.3	HE-MPC	0.5
nGraph-HE2-ReLU	98.62	0.69	2,959	HE-MPC	0.03

Table 6 shows the improvement due to complex packing. Complex packing does not take advantage of our scalar encoding optimization (Section 3.1.1), slightly increasing the runtime from the real packing case. Nevertheless, complex packing roughly halves the amortized runtime by doubling the capacity.

The total runtime is much smaller than the runtime in Table 5, due to the use of much smaller encryption parameters. The amortized runtime is also improved, though less dramatically. Note, the communication overhead between the server and client accounts for roughly 27% of the runtime in the LAN setting. Optimizing the communication leaves room for future improvement.

Table 7 shows the performance of nGraph-HE2 on the CryptoNets network compared to existing methods. Lola and Gazelle optimize for latency at the cost of reduced throughput. Other methods, such as CryptoNets, Faster CryptoNets, and nGraph-HE adopt the same batch-axis packing as we do, thereby optimizing for throughput. Our method achieves the highest throughput of 1,998 images/s on the CryptoNets network. Furthermore, the client-aided model enables an even higher throughput of 2,959 images/s. Notably, the latency of our approach is much smaller than previous batch-axis packing approaches, and has similar runtime as the latency-optimized LoLa, while achieving much larger throughput.

Table 8: MobileNetV2 results on localhost and LAN settings using complex packing, batch size 4096, 56 threads, and encryption parameters $N = 2^{12}, L = 3$ at $\lambda = 128$-bit security. Runtimes are averaged across 10 trials. Encrypting the data reduces the top-1 accuracy by an average of 0.0136%, ≈7 images in 50,000.

MobileNetV2 Model	Unencrypted Accuracy (%)		Encrypted Accuracy (%)		Runtime				Communication (MB/image)	Memory (GB)	
					Localhost		LAN				
	Top-1	Top-5	Top-1	Top-5	Amt. (ms)	Total (s)	Amt. (ms)	Total (s)		Client	Server
0.35-96	42.370	67.106	42.356 (−0.014)	67.114 (+0.008)	27	112 ± 5	71	292 ± 5	38.4	8.6	60.3
0.35-128	50.032	74.382	49.982 (−0.050)	74.358 (−0.024)	46	187 ± 4	116	475 ± 10	63.7	12.6	100.4
0.35-160	56.202	79.730	56.184 (−0.018)	79.716 (−0.014)	71	290 ± 7	197	807 ± 19	107.5	17.9	161.0
0.35-192	58.582	81.252	58.586 (+0.004)	81.252 (−0.000)	103	422 ± 23	278	1,141 ± 22	152.2	24.2	239.2
0.35-224	60.384	82.750	60.394 (+0.010)	82.768 (+0.018)	129	529 ± 18	381	1,559 ± 27	206.9	56.9	324.3

4.3.2 MobileNetV2. ImageNet [21] is a dataset used for image recognition, consisting of colored images, ≈1.2 million for training, and 50,000 for validation, classified into 1,000 categories. The images vary in size, though they are commonly rescaled to shape $224 \times 224 \times 3$. The large number of categories and large image resolution make ImageNet a much more difficult task than MNIST or CIFAR10 [29]. MobileNetV2 [38] is a lightweight network architecture which achieves high accuracy on ImageNet with a small number of parameters and multiply-accumulate operations. MobileNets are parameterized by an expansion factor, which can be used to reduce the model size, resulting in a faster runtime at the expense of lower accuracy. The ReLU activations reduce the effective multiplicative depth, enabling use of small encryption parameters, $N = 2^{12}$ and $L = 3$ coefficient moduli at $\lambda = 128$-bit security. Furthermore, the lack of ciphertext-ciphertext multiplications enables use of complex packing. We demonstrate nGraph-HE2 on MobileNetV2 with expansion factor 0.35, and image ranging from size 96×96 to the full size, 224×224.

Table 8 shows the results from MobileNetV2 inference on a variety of image sizes. The large increase in runtime from the localhost setting to the LAN setting is due to the communication overhead. The localhost setting therefore represents a lower-bound to the timings possible in the LAN setting. Notably, the accuracy degradation due to HE is ≈0.01%, less than 7 images in 50,000. Figure 1 shows the increase in runtime with larger images sizes, and the significant latency introduced by the LAN setting.

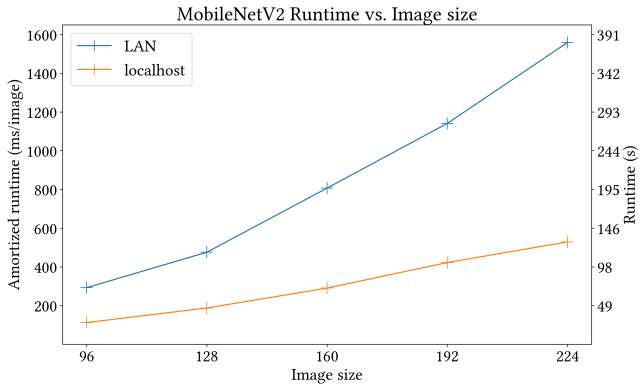

Figure 1: Runtime vs. Image size of LAN and localhost MobileNetV2 models. Table 8 shows the corresponding accuracies.

5 CONCLUSION AND FUTURE WORK

Homomorphic encryption is a promising solution to preserving privacy of user data during DL inference. Current DL solutions using HE induce significant slowdown and memory overhead compared to performing inference on unencrypted data. One potential solution to this overhead is the use of plaintext packing, which enables storing multiple scalars in a single plaintext or ciphertext. The choice of how to use plaintext packing typically either increases throughput, via batch-axis plaintext packing, or reduces latency, via inter-axis plaintext packing.

In this work, we presented nGraph-HE2, which introduced several optimizations to SEAL's implementation of the CKKS encryption scheme, for batch-axis plaintext packing. Our optimizations result in a 3x-88x improvement in scalar encoding, a 2.6x-4.2x speedup in ciphertext-plaintext scalar addition, and a 2.6x speedup in ciphertext-plaintext multiplication.

We also introduced lazy rescaling, a CKKS-specific graph-based optimization which reduces the latency by 8x on the CryptoNets network. Additionally, we introduced complex packing, which doubles the throughput with minimal effect on runtime.

Together, these optimizations enable state-of-the art throughput of 1,998 images/s for the CryptoNets network for the MNIST dataset. Furthermore, the integration of our approach with nGraph-HE enables inference on pre-trained DL models without modification. To demonstrate this capability, we presented the first evaluation of MobileNetV2, the largest DL model with linear layers evaluated homomorphically, with 60.4%/82.7% top-1/top-5 accuracy, and amortized runtime of 381 ms/image. To our knowledge, this is also the first evaluation of a model with encrypted ImageNet data.

One avenue for future work involves performing non-polynomial activations securely. In our approach, a client computes activations such as MaxPool and ReLU by first decrypting, the computing the non-linearity in plaintext, then encrypting the result. In the near future, we plan to add support for other privacy-preserving primitives, e.g., Yao's Garbled Circuit, to provide a provably privacy-preserving solution. Other directions for future work include further optimization of scalar encoding for complex numbers, and optimizing plaintext-ciphertext addition and multiplication with Intel Advanced Vector Extensions (AVX).

REFERENCES

[1] 2019. XONN: XNOR-based Oblivious Deep Neural Network Inference. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA. https://www.usenix.org/conference/usenixsecurity19/presentation/riazi

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: a system for large-scale machine learning. In *OSDI*, Vol. 16. 265–283.

[3] Paul Barrett. 1986. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *Conference on the Theory and Application of Cryptographic Techniques*. Springer, 311–323.

[4] Flavio Bergamaschi, Shai Halevi, Tzipora T Halevi, and Hamish Hunt. 2019. Homomorphic Training of 30,000 Logistic Regression Models. In *International Conference on Applied Cryptography and Network Security*. Springer, 592–611.

[5] Marcelo Blatt, Alexander Gusev, Yuriy Polyakov, Kurt Rohloff, and Vinod Vaikuntanathan. 2019. Optimized Homomorphic Encryption Solution for Secure Genome-Wide Association Studies. (2019).

[6] Fabian Boemer, Yixing Lao, Rosario Cammarota, and Casimir Wierzynski. 2019. nGraph-HE: a graph compiler for deep learning on homomorphically encrypted data. In *Proceedings of the 16th ACM International Conference on Computing Frontiers*. ACM, 3–13.

[7] Christina Boura, Nicolas Gama, Mariya Georgieva, and Dimitar Jetchev. 2018. CHIMERA: Combining Ring-LWE-based Fully Homomorphic Encryption Schemes. Cryptology ePrint Archive, Report 2018/758. https://eprint.iacr.org/2018/758.

[8] Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. 2018. Fast homomorphic evaluation of deep discretized neural networks. In *Annual International Cryptology Conference*. Springer, 483–512.

[9] Zvika Brakerski. 2012. Fully homomorphic encryption without modulus switching from classical GapSVP. In *Annual Cryptology Conference*. Springer, 868–886.

[10] Alon Brutzkus, Oren Elisha, and Ran Gilad-Bachrach. 2019. Low Latency Privacy Preserving Inference. In *International Conference on Machine Learning*. https://github.com/microsoft/CryptoNets/tree/6db77e36c4103385f0a621284d0c3609f0308e74#cryptonets

[11] Chia che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 645–658. https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai

[12] Hao Chen, Wei Dai, Miran Kim, and Yongsoo Song. [n. d.]. Efficient Multi-Key Homomorphic Encryption with Packed Ciphertexts with Application to Oblivious Neural Network Inference. ([n. d.]).

[13] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. 2018. A full RNS variant of approximate homomorphic encryption. In *International Conference on Selected Areas in Cryptography*. Springer, 347–368.

[14] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic encryption for arithmetic of approximate numbers. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 409–437.

[15] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachene. 2016. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 3–33.

[16] Edward Chou, Josh Beal, Daniel Levy, Serena Yeung, Albert Haque, and Li Fei-Fei. 2018. Faster CryptoNets: Leveraging Sparsity for Real-World Encrypted Inference. *arXiv preprint arXiv:1811.09953* (2018).

[17] Scott Cyphers, Arjun K. Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, William Constable, Christian Convey, Leona Cook, Omar Kanawi, Robert Kimball, Jason Knight, Nikolay Korovaiko, Varun Kumar, Yixing Lao, Christopher R. Lishka, Jaikrishnan Menon, Jennifer Myers, Sandeep Aswath Narayana, Adam Procter, and Tristan J. Webb. 2018. Intel nGraph: An Intermediate Representation, Compiler, and Executor for Deep Learning. *CoRR* abs/1801.08058 (2018). arXiv:1801.08058 http://arxiv.org/abs/1801.08058

[18] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering* 5, 1 (1998), 46–55.

[19] Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. 2019. CHET: an optimizing compiler for fully-homomorphic neural-network inferencing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 142–156.

[20] Daniel Demmler, Thomas Schneider, and Michael Zohner. 2015. ABY-A Framework for Efficient Mixed-Protocol Secure Two-Party Computation.. In *NDSS*.

[21] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 248–255.

[22] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat Practical Fully Homomorphic Encryption. Cryptology ePrint Archive, Report 2012/144. https://eprint.iacr.org/2012/144.

[23] Craig Gentry, Shai Halevi, and Nigel P Smart. 2012. Homomorphic evaluation of the AES circuit. In *Annual Cryptology Conference*. Springer, 850–867.

[24] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. 2016. CryptoNets: Applying neural networks to encrypted data with high throughput and accuracy. In *International Conference on Machine Learning*. 201–210.

[25] Shai Halevi and Victor Shoup. 2018. Faster homomorphic linear transformations in helib. In *Annual International Cryptology Conference*. Springer, 93–120.

[26] Ehsan Hesamifard, Hassan Takabi, and Mehdi Ghasemi. 2019. Deep Neural Networks Classification over Encrypted Data. In *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy (CODASPY '19)*. ACM, New York, NY, USA, 97–108. https://doi.org/10.1145/3292006.3300044

[27] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. 2018. GAZELLE: A Low Latency Framework for Secure Neural Network Inference. In *27th (USENIX) Security Symposium (USENIX) Security 18)*. 1651–1669.

[28] Duhyeong Kim and Yongsoo Song. 2018. Approximate Homomorphic Encryption over the Conjugate-invariant Ring. Cryptology ePrint Archive, Report 2018/952. https://eprint.iacr.org/2018/952.

[29] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. 2014. The CIFAR-10 dataset. *online: http://www. cs. toronto. edu/kriz/cifar. html* (2014).

[30] Yann LeCun. 1998. The MNIST database of handwritten digits. *http://yann. lecun. com/exdb/mnist/* (1998).

[31] Jian Liu, Mika Juuti, Yao Lu, and Nadarajah Asokan. 2017. Oblivious neural network predictions via miniONN transformations. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 619–631.

[32] Patrick Longa and Michael Naehrig. 2016. Speeding up the Number Theoretic Transform for Faster Ideal Lattice-Based Cryptography. Cryptology ePrint Archive, Report 2016/504. https://eprint.iacr.org/2016/504.

[33] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. 2013. On Ideal Lattices and Learning with Errors over Rings. *J. ACM* 60, 6, Article 43 (Nov. 2013), 35 pages. https://doi.org/10.1145/2535925

[34] Payman Mohassel and Peter Rindal. 2018. ABY 3: a mixed protocol framework for machine learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 35–52.

[35] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. In *NIPS-W*.

[36] M Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M Songhori, Thomas Schneider, and Farinaz Koushanfar. 2018. Chameleon: A hybrid secure computation framework for machine learning applications. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*. ACM, 707–721.

[37] Kurt Rohloff. 2018. The PALISADE Lattice Cryptography Library. Retrieved 2019-03-25 from https://git.njit.edu/palisade/PALISADE

[38] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 4510–4520.

[39] SEAL 2019. Microsoft SEAL (release 3.3). https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA.

[40] Tim van Elsloo, Giorgio Patrini, and Hamish Ivey-Law. 2019. SEALion: a Framework for Neural Network Inference on Encrypted Data. *arXiv preprint arXiv:1904.12840* (2019).

[41] Sameer Wagh, Divya Gupta, and Nishanth Chandran. 2019. SecureNN: 3-Party Secure Computation for Neural Network Training. *Proceedings on Privacy Enhancing Technologies* 1 (2019), 24.

[42] Wenhao Wang, Yichen Jiang, Qintao Shen, Weihao Huang, Hao Chen, Shuang Wang, XiaoFeng Wang, Haixu Tang, Kai Chen, Kristin Lauter, et al. 2019. Toward Scalable Fully Homomorphic Encryption Through Light Trusted Computing Assistance. *arXiv preprint arXiv:1905.07766* (2019).

A APPENDIX

A.1 Network Architectures

For each architecture, n indicates the batch size.

- CryptoNets, with activation $Act(x) = x^2$.
  (1) *Conv.* [Input: $n \times 28 \times 28$; stride: 2; window: 5×5; filters: 5, output: $n \times 845$] + *Act.*
  (2) *FC.* [Input: $n \times 845$; output: $n \times 100$] + *Act.*
  (3) *FC.* [Input: $n \times 100$; output: $n \times 10$].
- CryptoNets-ReLU, with activation $Act(x) = ReLU(x)$.
  (1) *Conv with bias.* [Input: $n \times 28 \times 28$; stride: 2; window: 5×5; filters: 5, output: $n \times 845$] + *Act.*
  (2) *FC with bias.* [Input: $n \times 845$; output: $n \times 100$] + *Act.*
  (3) *FC with bias.* [Input: $n \times 100$; output: $n \times 10$].

A.2 Parallel Scaling

nGraph-HE [6] uses OpenMP [18] to parallelize high-level operations such as Dot and Convolution. As such, the runtime depends heavily on the number of threads. For the CryptoNets network with $N = 2^{13}, L = 6$, Figure 2 shows the latency decreases linearly with the number of threads up to thread count 16. Best performance is achieved with 88 threads. However, the performance with 24 threads is just 9% slower (1.87s vs. 2.05s) than with 88 threads, representing a better runtime-resource tradeoff. In general, the optimal number of threads will depend on the network.

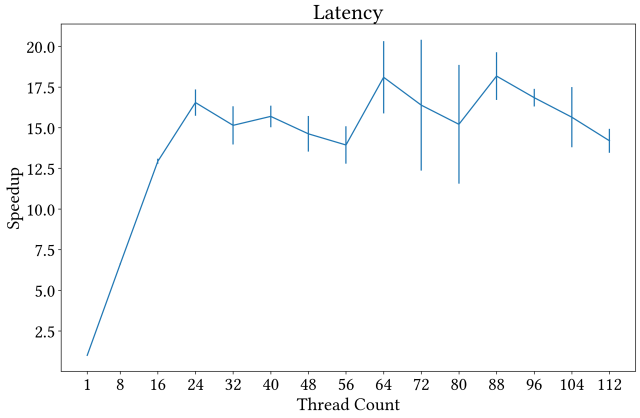

Figure 2: Runtimes on CryptoNets network with different number of threads. Runtimes are averaged across 10 trials.

A.3 Scalar Encoding

Lemma 1. *Refer to Algorithm 1 for the general CKKS encoding algorithm. If the input vector c consists of the same real number r in each slot, then the output plaintext p will contain the same real number in each slot.*

Proof. We refer to the notation in Algorithm 1. Since $c \in \mathbb{R}^{N/2}$, $c = c^*$, and so line 3 and line 4 yield $p \leftarrow (r, r, \ldots, r)$, the same value in every slot. Now, we show

$$\mathrm{DFT}^{-1}(r, r, \ldots, r) = (r, 0, 0, \ldots, 0).$$

The DFT^{-1} can be represented by a matrix transformation $W \in \mathbb{C}^{N \times N}$ with $W = \left(w_{jk}\right)_{0 \le j,k \le N-1}$ for $w_{jk} = \frac{\omega^{-jk}}{N}$ where $\omega = e^{-2\pi i/N}$ is a primitive N^{th} root of unity. In particular, the first row of W consists of all ones, and the sum of every j^{th} row for $j \neq 0$ is 0, since

$$\sum_{k=0}^{N-1} \frac{\omega^{-jk}}{N} = \frac{1}{N}\left(\frac{\omega^j(1-\omega^{-jN})}{\omega^j - 1}\right) = 0$$

where the last equality uses that ω is a root of unity. Now, since p has all the same values,

$$\mathrm{DFT}^{-1}(r, \ldots, r) = \Big(\sum_i p_i/N, 0, \ldots, 0\Big) = (r, 0, \ldots, 0).$$

Scaling by s yields $(rs, 0, \ldots, 0)$. The modulus reduction (line 6) yields $([rs]_q, 0, \ldots, 0)$. Finally, the negacyclic NTT (line 7) can also be represented by a matrix transformation in the finite field $\mathbb{Z}/q\mathbb{Z}$, the integers modulo q. As with the DFT^{-1} matrix, the first row is all ones, hence

$$\mathrm{NegacyclicNTT}(([rs]_q, 0, \ldots, 0)) = [rs]_q(1, 1, \ldots, 1).$$

Thus, the CKKS encoding has the same scalar, $[rs]_q$, in each slot. □

A.4 SEAL Performance Test

Table 9 shows the runtimes from SEAL's CKKS performance tests. The runtime increases with N and L. In general, larger L supports more multiplications. However, to maintain the same security level, N must be increased accordingly.

Table 9: SEAL CKKS performance test. Parameters satisfy $\lambda = 128$-bit security. Runtimes averaged across 1000 trials.

Operation	Runtime (µs)		
	$N = 2^{12}$ $L = 2$	$N = 2^{13}$ $L = 4$	$N = 2^{14}$ $L = 8$
Add	16	59	237
Multiply plain	58	234	936
Decrypt	54	214	883
Square	105	476	2,411
Multiply	155	709	3,482
Rescale	440	2,224	10,189
Encode	1,654	4,029	10,989
Encrypt	1,514	4,808	16,941
Relinearize	936	4,636	27,681
Decode	2,153	7,372	30,175
Rotate one step	1,098	5,294	30,338
Rotate random	4,683	25,270	158,905