Towards High Performance Homomorphic Encryption for Inference Tasks on CPU: An MPI approach

***Abstract*—Fully homomorphic encryption (FHE) is a powerful tool enabling secure outsourcing to untrusted third-parties. As AI technologies mature and companies start offering AI-as-a-service (AIaaS), the privacy of their customers' data needs to be considered but the high overhead of existing FHE schemes is a major limitation. Prior work tried to address this but suffered accuracy degradation, lack of scalability, ciphertext expansion and impractical performance issues. In this paper, we explore the utility of High Performance Computing systems for accelerating the secure evaluation of deep convolutional neural networks (CNN) with FHE. We examine previous encrypted neural network architectures and propose a holistic re-design for the HPC environment, which introduces a new consideration for the latency of inter-node communication. The new architecture was implemented with MPI and OpenMP and its performance benchmarked against state-of-the-art FHE-based CNN evaluation methods. We achieved improvements of between $6 - 34\times$ for various CIFAR10 CNN architectures and more than $1500\times$ for AlexNet, bringing FHE-based evaluation of AlexNet down from several hundred days to within 14 hours. We believe that practical inference on encrypted medium resolution images is around the corner with the use of HPC systems.**

***Index Terms*—Fully Homomorphic Encryption, Convolutional Neural Networks, High Performance Computing, Message Passing interface**

I. Introduction

Research on Artificial Neural Networks (ANNs) has advanced significantly in recent years, fueled by deep learning which has proven to be data-greedy. The success of ANNs has spanned a wide array of fields and industries, including but not limited to: healthcare, finance, manufacturing, autonomous vehicles, smart homes and cybersecurity [1], [2], [3], [4]. A commonality across these applications is the challenges faced in managing the data pipeline in a privacy-preserving manner. Data can be proprietary and too valuable to risk leaking to outsiders, preventing them from being given to third party service providers or contractors for analysis or inference. Data can also be intrinsically sensitive and protected by governments laws and regulations such as the GDPR, HIPPA and PDPC. Hence, there is a rising demand for practical privacy. To address some of these issues, technologies such as Fully Homomorphic Encryption (FHE), Secure Multi-Party Computations (SMPC) and Federated Learning (FL) are becoming increasingly popular. The latter two are especially useful for collaborative learning problems, whilst FHE provides a platform for secure computation outsourcing, enabling a wide range of services including AIaaS.

With over a decade of research and development since its introduction in 2009 [5], FHE still suffers from many pitfalls, chief of which is its slow processing time. This hinders its scalability to applications requiring deep neural networks such as image recognition for medium to high resolution images, with ImageNet being the representative dataset for such problems. Model architectures for ImageNet notoriously pose the challenge of high multiplicative depth, which translates to impractical runtimes. Previous attempts on CPU only managed to extrapolate results from running the first 2-3 layers of ImageNet model architectures [6], or relied on accuracy-degrading techniques such as polynomial approximations for activation functions such as [7], which have not been shown to work with models for ImageNet.

The Message Passing Interface (MPI) standard is a widely used programming interface for distributed memory systems. Hybrid parallel programming on many-core systems most often combine MPI with OpenMP. This hybrid approach uses an MPI model for communicating between nodes while utilizing groups of threads running on each computing node in order to take advantage of multi-core/many-core architectures.

Currently, High Performance Computing (HPC) systems scale to millions of processing cores with a peak performance of 125 PFlops (125×10^{15} floating-point operations per second) [8], paving the way to new designs that favor heavy parallelization to the benefit of runtime.

Related Work. Lei et al. [9] proposed heterogeneous HPC acceleration of basic FHE operations such as bootstrapping, key generation as well as arithmetic operations. Their design leveraged FHEW [10] into a multi-core CPU/GPU implementation that achieved an overall speedup of slightly more than 2-fold for homomorphic additions and multiplications. Roy et al. [11] turned to configurable hardware, the FPGA, and implemented a custom co-processor for the Fan-Vercauteren (FV) FHE scheme [12] achieving a $13\times$ speedup with respect to an Intel i5 processor. However, compared to more recent libraries such as Microsoft SEAL, FV implementations were observed to be slower [13]. Poppelmanm et al. [14] worked on the somewhat homomorphic encryption scheme YASHE [15]. Although it showed strong performance, YASHE was broken in 2016 by Kircher and Fouque [16].

Efforts of accelerating FHE computation generally focused on using more specialized hardware such as GPU [17], [18] and FPGAs [11], [13]. These work are orthogonal to our efforts since improving the computational power of each HPC node would also benefit the methods in this paper. SparkFHE [19] applied Apache Spark develop a distributed dataflow framework for computing on encrypted data, however, they only considered basic operations such as inner product of vectors.

Prior work on evaluating deep CNNs with FHE can be categorized into two groups, fast, approximated operations tailored for high-precision integer or approximate arithmetic and much slower, exact low-precision arithmetic from Boolean circuits. The former introduced new networks with traditional activation functions approximated with polynomial functions, these introduced some accuracy loss and no results on ImageNet was achieved yet; Whilst with the latter, existing approaches to compressing and quantizing deep neural network for hardware inference were adapted for FHE, building Boolean circuits that exactly translated operations on neural networks to work on encrypted bits. Work in this direction could achieve the same accuracy levels as conventional neural networks but suffered in performance even with high-powered servers and workstations due to the high overhead of FHE for evaluating Boolean circuits.

Our Contributions. In this work, we leverage IBM's HElib [20], based on the Brakerski-Gentry-Vaikuntanathan (BGV) scheme [21], to exploit the parallelization potential of CNN inference on data encrypted with FHE. Compared to previous works, our acceleration-focused redesign effort does not leverage optimizations on the fine-grained operations but rather the model architecture in its bigger picture. To that end, we re-examine the architecture of DOReN [22], a depth-optimized, space-efficient evaluation circuit for exact quantized CNNs, and improve it to efficiently execute in HPC environments. As such, we propose a hybrid MPI/OpenMP CPU implementation that reduces synchronization overhead and load imbalance, and minimizes communication and computational overheads. Through a combination of optimization techniques, we significantly reduce the runtime cost of private CNN inference tasks. Experiments were performed to compare our architecture with DOReN and SHE.

Outline of the Paper. In Section II, we describe the preliminaries used throughout the paper: fully homomorphic encryption, convolutional neural networks with encrypted data and parallel programming with MPI. Then, in Section III, we detail the architecture of our system as well as the different techniques leveraged in its design and implementation. Next, we present experiment results in Section IV on how our system performs on various neural network architectures for CIFAR10 and ImageNet. We finally conclude in Section V.

II. Preliminaries

A. Fully Homomorphic Encryption

A (leveled) fully homomorphic encryption scheme, at a high level, consists of four algorithms.

- $(pk, evk, sk) \leftarrow \mathsf{KeyGen}(1^(\lambda), \alpha)$: Takes as inputs the security parameter λ and level parameter α, which denotes the maximum circuit depth that can be evaluated, and outputs the encryption key pk, decryption key sk and evaluation key evk.
- $c \leftarrow \mathsf{Encrypt}(pk, m)$: Takes as inputs the encryption key pk and plaintext message m, and outputs a ciphertext c that encrypts m under pk.
- $m \leftarrow \mathsf{Decrypt}(sk, c)$: Takes as inputs the decryption key sk and ciphertext c, and outputs the message m that was encrypted within it.
- $c' \leftarrow \mathsf{Eval}(evk, f, \overline{m_1}, ..., \overline{m_k})$: Takes as inputs the evaluation key evk, a description of the function f (usually expressed as a circuit) and k input ciphertexts $\overline{m_1}, ..., \overline{m_k}$, and outputs a ciphertext c such that decrypting it yields $f(m_1, ..., m_k)$, i.e. $\mathsf{Decrypt}(sk, c) = f(m_1, ..., m_k)$.

For the FHE schemes considered in this work, BGV [21] and BFV [23], [12], Eval is performed by expressing the function f as a Boolean circuit over the inputs $m_1, ..., m_k$ using AND and XOR gates, which translated to homomorphic multiplication and addition respectively on encrypted bits. The natural plaintext space for these two FHE scheme (for encrypting bits) is denoted with $R_2 = \mathbb{Z}_2[X]/\langle\Phi_m(X)\rangle$, where $\Phi_m(X)$ is the m-th cyclotomic polynomial which has degree $\phi(m)$ with $\phi(\cdot)$ denoting the Euler Totient function.

Boostrapping. Existing FHE schemes so far are all based on adding noise to masked plaintexts. The mask can be removed with the secret key but homomorphic operations increase the noise levels in these ciphertexts until decryption can no longer filter out the plaintext from the noise and fails. Multiplication is especially expensive, in terms of the noise increase, and thus typical FHE schemes are designed to support circuits of some maximum depth.

To enable arbitrary computation on encrypted data, Gentry [5] introduced bootstrapping, homomorphically evaluating the decryption function using an encrypted secret key to "refresh" noisy ciphertexts. The process yields a refreshed ciphertext encrypting the same plaintext with a moderate amount of noise that was suitable for further use.

Single Instruction Multiple Data for FHE. For the abovementioned schemes, Smart and Vercauteren [24] showed that a vector of bits could be encrypted in each ciphertext by exploiting the isomorphism from Chinese Remainder Theorem:

$$R_2 = \mathbb{Z}_2[X]/\langle\Phi_m(X)\rangle \cong \prod_{i=1}^{\ell} \mathbb{Z}_2[X]/\langle F_i(X)\rangle \cong \prod_{i=1}^{\ell} \mathbb{F}_{2^d},$$

where $\Phi_m(X) \equiv \prod_{i=1}^{\ell} F_i(X) \bmod 2$. This allows ℓ bits to be encrypted in each ciphertext since $\mathbb{F}_2 \subset \mathbb{F}_{2^d}$ and enjoy homomorphic component-wise addition and multiplication. More generally, the plaintext space of each slot is $\mathbb{F}_{2^d} \cong \{\sum_{i=0}^{d-1} a_i Y^i \mid a_i \in \mathbb{F}_2\}$. On top of that, Gentry, Halevi and Smart [25] describe how data encrypted in such vectors can be shifted or rotation through some homomorphic operations. Halevi and Shoup [26] combine this with Benes networks to achieve arbitrary homomorphic permutations on the encrypted vectors. Bootstrapping with the BGV/BFV SIMD-enabled schemes are typically computationally expensive. However, Chen and Han [27] proposed a much faster method for bootstrapping ciphertexts that only contained plaintexts of the form $(\mathbb{F}_2)^\ell$ rather than the whole plaintext space $(\mathbb{F}_{2^d})^\ell$, shaving down the bootstrapping time by about d times.

Data Compression with Finite Extension Fields. Due to the nature of HE schemes, it is very difficult to find parameters that yield large number of SIMD slots when working with bits. Tan

et al. [28] proposed a technique to pack integers represented in base-Q, for prime Q, in FHE ciphertexts during transit or storage and extract their digits into separate ciphertexts when it is time for computation.

The key idea is that the plaintext algebra in each slot is a $\mathbb{F}_2$-vector space and support depth-free evaluation of linear maps on vectors in this space. Extracting the i-th component of a vector $X = (X_0, ..., X_{d-1})$ is an $\mathbb{F}_2$-linear map, $T_i(X) = (X_i, 0, ..., 0)$. Concretely, applying T_i is achieved by evaluating an associated 2-linearized polynomials, i.e. $f_{T_i}(X) = \sum_{i=0}^{d-1} a_i X^{2^i}$, requiring constant multiplication $a_i X^{2^i}$ and automorphism evaluation ($X \mapsto X^{2^i}$) which is realized with key-switching.

B. Neural Networks with Encrypted Data

Convolutional Neural Networks (CNNs). These are built up from layers, such as convolution, fully connected and pooling layers. The first two comprise of neurons that evaluate formulas of the form $F(b + \sum_{i=1}^{k} w_i x_i)$, where F is a non-linear activation function, k inputs $x_1, ..., x_k$ which are each associated to some weight w_i and bias term b; typical activation functions include sigmoid ($\sigma(x) = 1/(1+e^x)$) and ReLU ($\mathsf{ReLU}(x) = \max\{x, 0\}$). Their main difference being convolution takes a subset of neurons in the previous layer as input while fully connected neurons take all neurons in the previous layers. Neurons in the pooling layer have the form $P(x_1, ..., x_k)$, where P is a pooling function and k inputs $x_1, ..., x_k$; typical P include the maximum and average functions. Pooling layer neurons also generally take a subset of neurons in the previous layer.

For efficiency on hardware and homomorphic encryption, CNNs can be quantized, restricting their inputs, outputs and parameters to a finite set. As an example, neuron input and output may be quantized to 8-bit integers, while weights/biases are further restricted to powers of two and zero, e.g. $\{0, \pm 2^0, \pm 2^1, \pm 2^2, \pm 2^3, \pm 2^4\}$, to replace integer multiplication with even more efficient bit-shifts. Quantization adds one more step to scale down and then truncate the output of the traditional neuron.

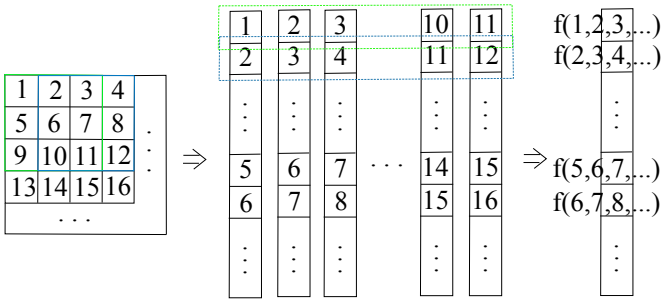

Fig. 1: The Encoding of an Input Image

Adapting CNNs to Encrypted Data. Meftah et al. [22] proposed a system called DOReN to evaluate neural networks on encrypted data using the BGV FHE scheme. An important difference using the BGV FHE scheme compared to other systems is the need to optimize the depth of the computation, instead of the number of gates required. SIMD was also exploited to increase the number of neurons that was evaluated with each operation and drastically improved running times of convolutional neural networks with FHE.

The different layers used in convolutional neural networks were re-designed to work on encrypted data with a view of optimizing the computation to the characteristics of the BGV FHE scheme. To that end, they proposed a depth-optimized method of evaluating neurons in the convolution and fully connected layers, which we refer to as *linear-then-activate* layers, exploiting the constant-depth 3:2 compressor to reduce the depth of such neurons to $O(\log m + \log n)$, where m is the number of inputs to the neuron and n is the maximum bit-width of those inputs.

On top of that, the capability of SIMD to work with vectors of bits in BGV ciphertexts was exploited to evaluate many neurons simultaneously. Unlike conventional SIMD use, the focus was to maximize the number of neurons evaluated. As illustrated in Figure 1, the inputs to convolution layers were encoded in a different vector formatting that favors SIMD. This introduced some complexity because there was a difference in the output data layout in ciphertexts and expected data layout for inputs, which was addressed by proposing transformation layers that convert the layout of outputs of previous layers (activations) to the layout needed in the next layer. We improve on that by replacing the computationally-expensive benes-network-based transformations with simple rotation operations that encode the input to fully connected layers as shown in Figure 2. Such encoding enabled efficient use of the accumulation circuit designed, whilst improving the performance.

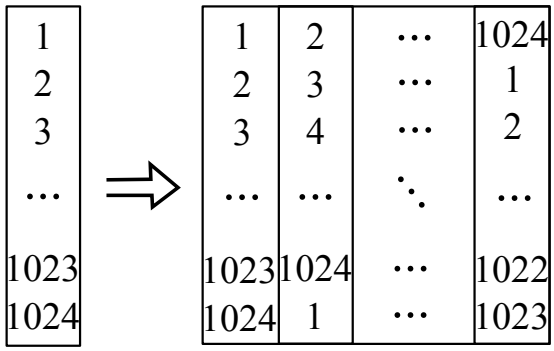

Fig. 2: Input to FC Layer Transformed using Rotations

C. Parallel Programming with MPI.

MPI Processes When distributing a program over several machines, MPI spawns several instances of the same code running independently, able to interact only through messages. Each process is assigned a consistent unique identifying integer number that we label as the $world\_rank$. The master process is conventionally assigned the $world\_rank$ 0. It serves as a central node that orchestrates the communication amongst all processes. Figure 3 illustrates the baseline workflow of an MPI process execution. As opposed to OpenMP which was designed for shared memory (i.e. for a single

system with multiple cores), MPI was designed for distributed memory (i.e. multiple systems exchanging messages). Hence, the interface provides a wide array of routines that enable complex communication scenarios amongst processes. For this work, we achieve our goals with only the basic MPI routines abstracted in Figure 3.

Hybrid Parallel Programming. The hybrid MPI and OpenMP paradigm denotes a style of programming in which multiple OpenMP threads are running within MPI processes. The masteronly flavor of this paradigm assumes calling MPI routines only outside of the parallel regions and is found to be faster than pure MPI when thread overhead is properly handled [29]. Symmetric Multiprocessing, also known as shared-memory multiprocessing (SMP) nodes are tightly coupled multiprocessing systems with a shared memory pool. OpenMP is independently ran on each SMP node, whilst the collection of the nodes are interconnected through MPI messages. This hybrid setting begs an optimal usage of the hardware including minimizing synchronization/idle time as well as properly managin the communication bandwisth of the hardware.

III. Architecture

In this section, we describe our system architecture for the distributed evaluation of quantized convolutional neural networks on encrypted data. First, we present an optimization for communication between the nodes in the distributed environment and its impact on the system design. Then, we introduce a distributed architecture for evaluating linear-then-activate layers when they have many (several hundred to thousands) inputs. Following that, we present the complete system which includes smaller convolution, pooling and transformation layers.

A. Optimizing Communication with Ciphertext Compression

In a distributed computing environment, messages will be exchanged between the nodes so that they have the inputs to the functions being processed. For privacy-preserving computing with FHE, especially with schemes that support SIMD packing, these messages are ciphertext arrays that can be between several hundred to thousands of megabytes in size. To minimize the impact of latency in the distributed system, we apply the ciphertext compression techniques of Tan et al. [28] and pack all the bits of an activation/input (as a polynomial) in a single ciphertext before sending them out. For an integer $x = \sum_{i=0}^{7} x_i 2^i$, the vector of encrypted bits, $(\overline{x_0}, ..., \overline{x_7})$, is compressed into a single polynomial $\overline{\mathsf{Compress}(x)} = \overline{\sum_{i=0}^{7} x_i Y^i}$. Although this reduces the overhead that a distributed system incurs due to the need to communicate, we now have to adjust the computation layers. A step to decompress the received ciphertexts to recover the vector of encrypted bits has to be done before actual neuron evaluation can begin.

Although that is the case for the linear-then-activate and pooling layers, the transformation layer proposed by Meftah et al. [22] is actually improved by this change. Previously, A set of 8 ciphertexts packed inputs in a bit-sliced manner and thus needed 8 invocations of the same transformation. With ciphertext compression, the transformation only has to be applied to a single packed ciphertext, resulting in an 8-fold improvement for the transformation layer.

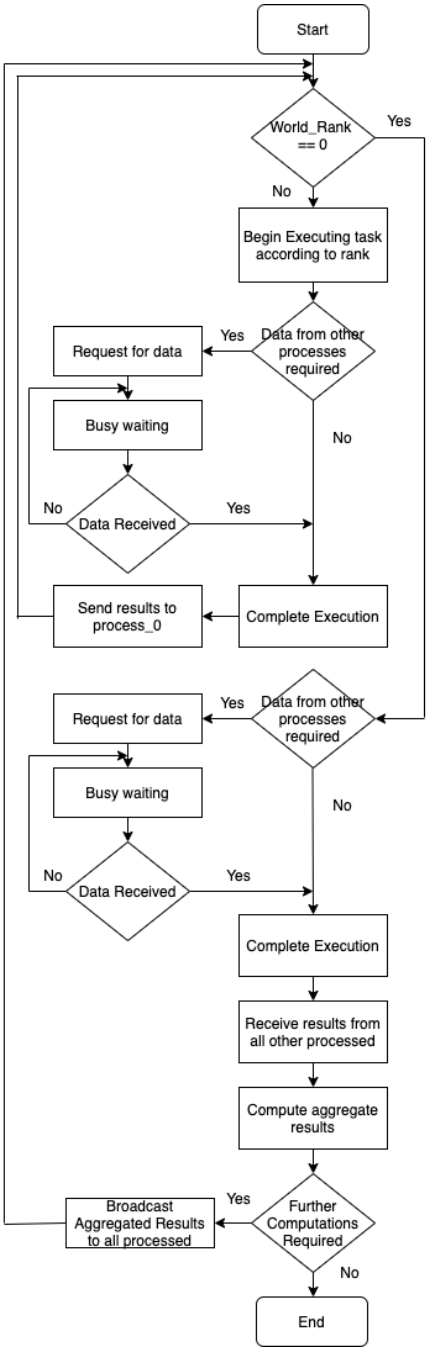

Fig. 3: MPI Process Flow Chart

B. An Alternate Linear-then-Activate Layer

A key component of quantized CNNs, convolution and fully connected layers, consist of neurons that evaluate the following function:

$$output = \mathsf{ReLU}'\left(b + \sum_{i=1}^{k} w_i x_i\right). \tag{1}$$

We use ReLU' to denote the ReLU-then-quantize function for quantized CNNs. Meftah et al. [22] proposed a 3-step structure (Steps 2-4) for neuron evaluation, with the ciphertext compression technique introducing one step before and after that.

1) Decompress inputs into bit-sliced encrypted integers;
2) Apply weights w_i to their respective inputs x_i;
3) Sum the weighted inputs and bias into a result, x;
4) Apply $\mathsf{ReLU}'(x)$ to obtain the result;
5) Compress the result into a single ciphertext.

Combining Bit-Shifting and Decompression. With the changes to reduce communication overhead, neuron evaluation is now slightly more complicated. However, we introduce a new weight application technique to overcome this situation, exploiting the decompression process to directly create a set of ciphertexts encrypting weight-applied inputs. First, weighted inputs may no longer be 8 bits wide. Assuming bit-shifts of up to 4 positions, weighted inputs could require up to $8 + 4 = 12$ bits. Instead of decompressing first and then shifting to obtain 12 bit weighted inputs, we directly use the bit-extraction procedure to place individual bits of the inputs into location based on the weight to apply.

Decompression uses a set of extraction maps $f_{T_0}, ..., f_{T_7}$ to get a vector of ciphertexts $(\overline{x_0} = f_{T_0}(\overline{x}), ..., \overline{x_7} = f_{T_7}(\overline{x}))$ from an encrypted integer $\overline{x}$, with $x = \sum_{i=0}^{7} x_i 2^i$ encoded as $x = \sum_{i=0}^{7} x_i Y^i$. With power-of-two weights, inputs are simply bit-shifted, e.g. $\overline{2^2 \cdot x} = (\overline{0}, \overline{0}, \overline{x_0} = f_{T_0}(\overline{x}), ..., \overline{x_7} = f_{T_7}(\overline{x}))$, a 10-bit long vector. The lower order positions can be populated with zeroes by evaluating a zero map using the same methods as the f_{T_i}'s, i.e. a map Z such that $Z(\overline{x}) = 0$.

In general, let $y = \sum_{i=0}^{11} y_i 2^i$ denote the result of applying the weight w to the input x. Then, we can write, for weight $w = 2^j$, where $0 \leq j \leq 4$,

$$\overline{y_i} = \begin{cases} f_{T_{i-j}}(\overline{x}), & \text{if } 0 \leq i - j < 8; \\ Z(\overline{x}) = 0, & \text{otherwise.} \end{cases}$$

Combining weight application with decompression needs more linear map evaluations. But practically, its effect on runtime is slight because the bulk of computation costs can be amortized over multiple maps.

Modified Negation. Besides combining bit-shifts with the decompression process, we propose another modification to weight application. Here, our goal is to eliminate the need for a full-adder in the weight application step for converting negative-weighted inputs into their two's complement form, reducing the circuit depth of weight application to 0. Multiplying negative weights for powers-of-two quantized CNNs can be done by first shifting the bits and then applying the two's complement negation formula, i.e. for 8-bit x_i,

$$-|2^k|x_i = 2^{k+8} - |2^k|x_i = (2^{k+8} - 1) \oplus |2^k|x_i + 1.$$

A full adder is used to add 1 and obtain the two's complement representation of $-|2^k|x_i$, which requires $O(\log 8 + k)$ depth. However, we propose to modify the convolution operation within Equation (1) so that all the "+1"s from two's complement negation are folded into the bias. Assuming 4-bit weights, we interpret all products as 12-bit integers and obtain

$$\begin{aligned} b + \sum_{i=1}^{k} w_i x_i &= b + \sum_{w_i<0} -|w_i|x_i + \sum_{w_i \geq 0} |w_i|x_i \\ &= b' + \sum_{w_i<0} (2^{12} - 1) \oplus |w_i|x_i + \sum_{w_i \geq 0} |w_i|x_i, \end{aligned}$$

where $b' = b + W$ with W is the number of negative weights.

Packing Neurons with Different Weights. With many neurons to evaluate in any given layer, the focus is on maximizing the number of simultaneous neuron evaluations with SIMD. In early layers, there are sufficiently many features per channel to fill a single ciphertext but this is turned around at deeper layers. Previous methods of bit-shifting inputs work best when the same weight is applied to all slots, but deeper convolutional layers and fully connected layers do not allow that easily since they have fewer/one neuron in each "channel".

In this re-design, different shift values can be performed on each slot by evaluating the appropriate 2-linearized polynomial on them. Next, we introduce a bit-mask, $z = (z_0, ..., z_{\ell-1}) \in \{0, 1\}^\ell$, to perform one's complement negation for slots with negative weights without affecting the other slots. For a ciphertext that can pack ℓ-length inputs,

$$z_j = \begin{cases} 1, & \text{if weight for the slot is negative;} \\ 0, & \text{otherwise.} \end{cases}$$

Then, after the merged shift and decompress operation, we obtain a set of 12 ciphertexts $\{\overline{y_k}\}_{k=0}^{11}$, where $\overline{y_k} = \overline{(y_{k,0}, ..., y_{k,\ell-1})}$ denotes the ciphertext encrypting the k-th bit of the packed shifted inputs, we simply compute $\overline{y_k} \oplus z$. Effectively, the slots with negative weights would become their one's complement ($\oplus 1$) and those with positive weights remain unchanged ($\oplus 0$).

Algorithm 1: Pseudo-code for an MPI distribution

```
call MPI_Init();
call MPI_Comm_rank();
call MPI_Comm_size();
//Client sends encrypted data to server
if world_rank==0 then
    output_stream ← Serialize(inputs);
    string ← output_stream.str();
    length ← string.length();
end
call MPI_Bcast(&Length);
Declare char cstr[Length];
if world_rank==0 then
    cstr ← string.c_str();//
end
call MPI_Bcast(&cstr);
inputs ← Deserialize(input_stream);
```

Distributing Linear-then-Activate Layers with Many Inputs. In the new architecture of linear-then-activate layers, it can be broken down into two phases, individually processing each input (applying weights), then reducing the set of weighted inputs to produce the final output. Therefore, the

most natural part to distribute is the weight application since it is completely independent for each ciphertext input.

However, the bulk of the computation as shown by Meftah et al. [22] lie in the accumulation of weighted inputs. Fortunately, this is hierarchical and can be decomposed into two parts. First, we partition the accumulation inputs among several MPI processes and separately accumulate these partitions into partial sums. Finally, these partial sums are gathered, then accumulated to the final value and activated with ReLU'.

C. The System Implementation

Distribution of input data and intermediate results. The distributed architecture relies on the communication functionality presented by the Message Passing Interface. The objects and variables that we wish to communicate across processes are converted into streams of byte-sized characters through serialization. The length of the string holding the stream is communicated to all processes for allocating proper memory, before the massage itself is broadcast as illustrated in Algorithm 1.

Algorithm 2: Pseudo-code for a distributed convolution

```
for i ← 0 to Number of Filters -1 do
    if world_rank==i+1 then
        Shift_Inputs_by_Weight_Values;
        VFE_Accumulate;
        Quantized_Activate;
        call MPI_Send(Result → Process_0);
    end
end
if world_rank==0 then
    for i ← 1 to Number of Filters do
        call MPI_Probe(&Status ← i);
        call MPI_Get_Count(l ← &Status);
        call MPI_Receive([l] ← Result);
        Concatenate_Results;
    end
end
call MPI_Bcast(&Length);
call MPI_Bcast(&Results);
```

For convolution and max-pooling layers, the hybrid parallelization architecture lent itself naturally to the algorithm. Since the same computations for both layer types are repeated across filters, each filter can be independently processed by a different MPI process in a non-blocking manner and without major changes to the code base as shown in Algorithm 2. The OpenMP multi-threading is then used within each process to accelerate addition, multiplication and shift operations applied uniformly to vectors of ciphertexts. "Omp for" directives were also utilised to distribute loop iterations within the team of threads that encounters this work-sharing construct, for any loop whose logic supports it.

IV. Experiments

For our experimental setup, we adopt clusters in the Amazon Web Services (AWS). The machines run the Ubuntu Server 20.04 LTS (HVM), SSD Volume Type AMI. All machines instantiated in our clusters are of type r5.24xlarge (-ECUS, 96 vCPUs, 3.1GHz, -, 768 GiB memory, EBS only), with a network performance of 25 Gigabit.

A. CIFAR10 Results

As explained in section III-A, the ciphertext compression techniques allowed to significantly reduce the MPI communicaiton overhead. For instance. The initial broadcast of an encoded private CIFAR10 image resulted in a stream size of 1,422,663,449 bytes. This was effectively reduced to a size of 117,931,538 bytes, which is a 12-fold improvement on the communication cost. Not only does it reduce the strain on the network resources, but it also benefits the overall runtime performance of the system as shown in Figure 4. The improvement factor on the communication cost carries over throughout the layers of all networks architectures and ranges between 28 to 33 folds depending on the number of MPI processes across which the output is distributed.

The number of machines deployed to run each of the network architectures was chosen based on both CPU and RAM requirements. Naturally, wider networks had support for more aggressive parallelization due to the depth of filters in each layer as well as the large volume of vector operations to be carried out. As summarized in Table I, the improvement factor ranged from 6.45 to 34.48 folds in runtime compared to DOREN [22], bringing down the inference time of homomorphically encrypted CNNs to production-friendly figures.

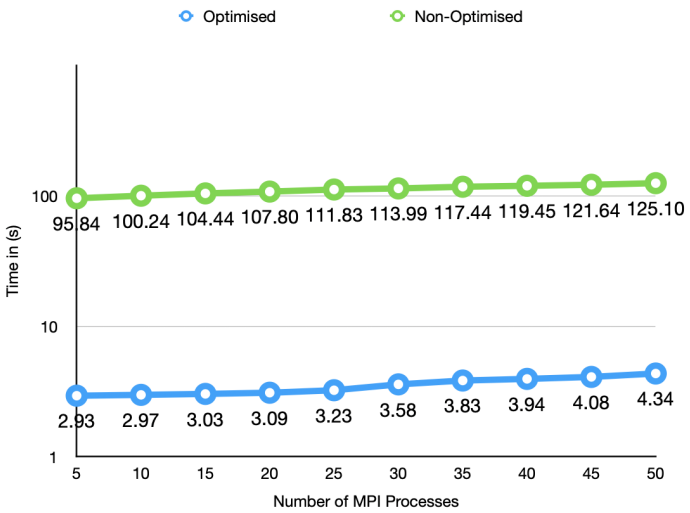

Fig. 4: Communication time necessary to distribute an encoded, encrypted CIFAR10 image over MPI processes

B. ImageNet Results

To the best of our knowledge, no prior works ever managed to run a fully homomorphically encrypted network architecture based on the ImageNet dataset on CPU. Previous attempts were only able to extrapolate the runtime required through running the first 2-3 layers of the architectures. In fact, even thanks to our highly optimized and heavily parallelized design,

Network Architecture	Non-Distributed Runtime*	Number of Cores	Number of Machines	Total RAM Available (Gb)	Accuracy (%)	Distributed Runtime*	Gain in Runtime
VGG	5483.3	513	16	12.16k	84.76	159	34.48x
Resnet 20	3333.3	65	3	2.28k	87.84	517	6.45x
LoLa	1341.6	164	7	5.32k	81.60	69	19.4x
GHE	585	81	4	3.04k	73.85	80	7.31x
SHE	2200	257	9	6.84k	92.32	217	10.13x

*Timings are given in minutes.

TABLE I: CIFAR10 Results (Non-Distributed Runtime: Reported in [22]; Number of Cores: Number of MPI processes spawned over which the network is parallelized; Number of Machines: All machines used for parallelization have the same specificaitons described eariler in this section; Gain inn Runtime: Expressed in folds of improvement with the non-distributed runtime as a baseline)

we still required the use of no less than 32 machines to be able to run AlexNet within a reasonable time-frame. Nonetheless, the design retains its flexibility in trading off operational cost represented by the recommended number of machines with the runtime performance of the CNN inference.

We summarize our findings in Table II, where we run the AlexNet architecture, achieving a runtime of under 800 minutes and top-1 accuracy of 55.9% achieving a 1518 folds improvement over Lou and Jiang [6]. Our results for the ResNet-18 architecture ran in under 2 days with an top-1 accuracy of 70.74%, achieving a runtime improvement of 1196 folds. Both networks followed a 4-bits weights and activations quantization.

V. Conclusion and Future Works

As we are headed towards the availability, affordability and popularity of high performance computing, we took the first step to recognise its big potential in solving complex encryption tasks such as CNN inference using FHE. Our experiments demonstrated up to 35 folds of improvement on the runtime of several CIFAR10 architectures, and up to 33 folds of improvement on communication overhead, as well as a first of its kind support for ImageNet architectures. Our design also reduced synchronization overhead and load imbalance leaving memory consumption as the only trade-off to a secure, private and fast CNN inference.

As future works, we are exploring the support for floating point model parameters which would slightly boost the accuracy of the networks as well as lay ground for numerous applications constrained by such support.

References

[1] O. Kocabas, T. Soyata, J. Couderc, M. Aktas, J. Xia, and M. Huang, "Assessment of cloud-based health monitoring using homomorphic encryption," in *2013 IEEE 31st International Conference on Computer Design (ICCD)*, 2013, pp. 443–446.

[2] Q. Li, Y. Yue, and Z. Wang, "Deep robust cramer shoup delay optimized fully homomorphic for iiot secured transmission in cloud computing," *Computer Communications*, 2020.

[3] M. Chraibi, S. Meftah, A. Maach, and H. Harroud, "Policy based context aware service level agreement (sla) management in the cloud," in *CLOUD COMPUTING 2017 : The Eighth International Conference on Cloud Computing, GRIDs, and Virtualization*, 2017, pp. 122–128.

[4] S. Meftah, T. Rachidi, and A. Nasser, "Network based intrusion detection using the unsw-nb15 dataset," *International Journal of Computing and Digital Systems*, 2019.

[5] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, ser. STOC '09. Association for Computing Machinery, 2009, p. 169–178.

[6] Q. Lou and L. Jiang, "SHE: A fast and accurate deep neural network for encrypted data," in *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*, 2019, pp. 10 035–10 043.

[7] N. Dowlin, R. Gilad-Bachrach, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing, "Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy," Microsoft, Tech. Rep., February 2016.

[8] Y. Zheng and P. Marguinaud, "Simulation of the performance and scalability of message passing interface (mpi) communications of atmospheric models running on exascale supercomputers," *Geoscientific Model Development*, vol. 11, no. 8, pp. 3409–3426, 2018. [Online]. Available: https://gmd.copernicus.org/articles/11/3409/2018/

[9] X. Lei, R. Guo, F. Zhang, L. Wang, R. Xu, and G. Qu, "Optimizing fhew with heterogeneous high-performance computing," *IEEE Transactions on Industrial Informatics*, vol. 16, no. 8, pp. 5335–5344, 2020.

[10] L. Ducas and D. Micciancio, "Fhew: Bootstrapping homomorphic encryption in less than a second," in *Advances in Cryptology – EUROCRYPT 2015*, E. Oswald and M. Fischlin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 617–640.

[11] S. Sinha Roy, F. Turan, K. Jarvinen, F. Vercauteren, and I. Verbauwhede, "Fpga-based high-performance parallel architecture for homomorphic computing on encrypted data," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 387–398.

[12] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," Cryptology ePrint Archive, Report 2012/144, 2012.

[13] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "Heax: An architecture for computing on encrypted data," 2020.

[14] T. Pöppelmann, M. Naehrig, A. Putnam, and A. Macias, "Accelerating homomorphic evaluation on reconfigurable hardware," in *Cryptographic Hardware and Embedded Systems – CHES 2015*, T. Güneysu and H. Handschuh, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 143–163.

[15] T. Lepoint and M. Naehrig, "A comparison of the homomorphic encryption schemes fv and yashe," in *Progress in Cryptology – AFRICACRYPT 2014*, D. Pointcheval and D. Vergnaud, Eds. Cham: Springer International Publishing, 2014, pp. 318–335.

[16] P. Kirchner and P.-A. Fouque, "Revisiting lattice attacks on overstretched NTRU parameters," in *EUROCRYPT 2017, Part I*, ser. LNCS, vol. 10210. Springer, Heidelberg, Apr. / May 2017, pp. 3–26.

[17] A. A. Badawi, B. Veeravalli, C. F. Mun, and K. M. M. Aung, "High-performance FV somewhat homomorphic encryption on GPUs: An implementation using CUDA," *IACR TCHES*, vol. 2018, no. 2, pp. 70–95, 2018.

[18] A. Al Badawi, B. Veeravalli, J. Lin, N. Xiao, M. Kazuaki, and A. Khin Mi Mi, "Multi-gpu design and performance evaluation of homomorphic encryption on gpu clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 2, pp. 379–391, 2021.

[19] P. Hu, A. Aloufi, A. Caulfield, K. Laine, and K. Lauter, "Sparkfhe:

Network Architecture	Non-Distributed Runtime*	Number of Cores	Number of Machines	Accuracy	Total RAM Available (Gb)	Distributed Runtime*	Gain in Runtime
AlexNet	1.2M	385	32	55.9%	24.57k	790.5	$\sim$1518x
ResNet18	3.25M	513	48	70.74%	36.85k	2716.5	$\sim$1196x

*Timings are given in minutes.

TABLE II: ImageNet Results: Accuracy refers to Top-1 accuracy; Non-Distributed Runtime is Reported in [6]); Number of Cores: Number of MPI processes spawned over which the network is parallelized; Number of Machines: All machines used for parallelization have the same specificaitons described eariler in this section; Gain inn Runtime: Expressed in folds of improvement with the non-distributed runtime as a baseline)

Distributed dataflow framework with fully homomorphic encryption," in *PPML-PriML 2020 workshop co-located with NeurIPS 2020*, 2020.

[20] S. Halevi and V. Shoup, "Algorithms in helib," in *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I*, 2014, pp. 554–571.

[21] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," in *ITCS 2012*. ACM, 2012, pp. 309–325.

[22] S. Meftah, B. H. M. Tan, C. F. Mun, K. M. M. Aung, B. Veeravalli, and V. Chandrasekhar, "Doren: Towards efficient deep convolutional neural networks with fully homomorphic encryption," *IEEE Transactions on Information Forensics and Security*, pp. 1–1, 2021.

[23] Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical GapSVP," in *CRYPTO 2012*, ser. LNCS, vol. 7417. Springer, Heidelberg, 2012, pp. 868–886.

[24] N. P. Smart and F. Vercauteren, "Fully homomorphic simd operations," *Designs, Codes and Cryptography*, vol. 71, no. 1, pp. 57–81, 2014.

[25] C. Gentry, S. Halevi, and N. P. Smart, "Fully homomorphic encryption with polylog overhead," in *EUROCRYPT 2012*, ser. LNCS, vol. 7237. Springer, Heidelberg, 2012, pp. 465–482.

[26] S. Halevi and V. Shoup, "Bootstrapping for helib," in *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, 2015, pp. 641–670.

[27] H. Chen and K. Han, "Homomorphic lower digits removal and improved FHE bootstrapping," in *EUROCRYPT 2018, Part I*, ser. LNCS, vol. 10820. Springer, Heidelberg, 2018, pp. 315–337.

[28] B. H. M. Tan, H. T. Lee, H. Wang, S. Q. Ren, and K. M. M. Aung, "Efficient private comparison queries over encrypted databases using fully homomorphic encryption with finite fields," *IEEE Transactions on Dependable and Secure Computing*, pp. 1–1, 2020.

[29] R. Rabenseifner, "Hybrid parallel programming: Performance problems and chances," 2003.