

HeLayers: A Tile Tensors Framework for Large Neural Networks on Encrypted Data

Ehud Aharoni
IBM Research, Israel

Allon Adir
IBM Research, Israel

Moran Baruch
IBM Research, Israel
Bar Ilan University
Israel

Nir Drucker
IBM Research, Israel

Gilad Ezov
IBM Research, Israel

Ariel Farkash
IBM Research, Israel

Lev Greenberg
IBM Research, Israel

Ramy Masalha
IBM Research, Israel

Guy Moshkovich
IBM Research, Israel

Dov Murik
IBM Research, Israel

Hayim Shaul
IBM Research, Israel

Omri Soceanu
IBM Research, Israel

ABSTRACT

Privacy-preserving solutions enable companies to offload confidential data to third-party services while fulfilling their government regulations. To accomplish this, they leverage various cryptographic techniques such as Homomorphic Encryption (HE), which allows performing computation on encrypted data. Most HE schemes work in a SIMD fashion, and the data packing method can dramatically affect the running time and memory costs. Finding a packing method that leads to an optimal performant implementation is a hard task. We present a simple and intuitive framework that abstracts the packing decision for the user. We explain its underlying data structures and optimizer, and propose a novel algorithm for performing 2D convolution operations. We used this framework to implement an inference operation over an encrypted HE-friendly AlexNet neural network with large inputs, which runs in around five minutes, several orders of magnitude faster than other state-of-the-art non-interactive HE solutions.

KEYWORDS

homomorphic encryption, packing optimization, privacy preserving machine learning, neural networks, non-interactive homomorphic encryption, convolutional layers

1 INTRODUCTION


Homomorphic Encryption (HE) schemes allow computations to be performed over encrypted data while providing data confidentiality for the input. Specifically, they allow the evaluation of functions on encrypted input, which is useful when outsourcing sensitive data to a third-party cloud environment. For example, a hospital that provides an X-ray classification service (e.g., COVID-19 versus pneumonia) can encrypt the images using HE, express the classification algorithm as a function, and ask a cloud service to evaluate it over the encrypted data without decrypting it. In this way, the hospital can use the cloud service while possibly complying with regulations such as HIPAA [9] and GDPR [18].

The proliferation of HE solutions in the last decade [20] shows that customers are eager to use them and that companies and organizations strive to provide secure and efficient solutions such as HEBench [2]. Nevertheless, it turns out that running large Neural Networks (NNs) using HE only is still considered an expensive task. For example, the best implementations [19] of AlexNet [31] with large inputs of $224 \times 224 \times 3$ before this paper, was measured as taking more than 3 hours using only 80 security bits. This barrier forces users to search for other secure alternatives instead of enjoying the advantage of solutions that rely only on HE. Our proposed framework aims to narrow down this barrier, allowing the users to better utilize cloud capabilities while operating on their confidential data.

Some HE schemes, such as CKKS [13], operate on ciphertexts in a homomorphic Single Instruction Multiple Data (SIMD) fashion. This means that a single ciphertext encrypts a fixed size vector, and the homomorphic operations on the ciphertext are performed slot-wise on the elements of the plaintext vector. To utilize the SIMD feature, we need to pack and encrypt more than one input element in every ciphertext. The packing method can dramatically affect the *latency* (i.e., time to perform computation), *throughput* (i.e., number of computations performed in a unit of time), communication costs, and memory requirements. The designer of an HE-based system is usually tasked with choosing the ciphertext-size. The size is bounded from above by the FHE-library implementation and from below by security parameters. Once selected, the ciphertext-size stays fixed for the entire process (with different packing options). To demonstrate the effect of different packing choices we use CryptoNets [21]. We summarize the results in Table 2, and observe that using two naïve packing solutions achieved latencies of 0.86 sec. and 11.1 sec., with memory usage of 1.58 GB and 14 GB, respectively. In comparison, a different non-trivial packing method achieved better latency of 0.56 sec. and memory usage of 0.73 GB. Section 7 provides more details about these three packing methods.

We also demonstrate why packing is hard by presenting the problems involved in encrypting and packing a matrix. A naïve packing option may keep every row or column in a different ciphertext. When the evaluated algorithm performs manipulations over rows, one packing option is clearly better than the other. However, when the matrix dimensions are small, the encrypted ciphertext

This work is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license visit <https://creativecommons.org/licenses/by/4.0/> or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.



Proceedings on Privacy Enhancing Technologies 2023(1), 325–342
© 2023 Copyright held by the owner/author(s).
<https://doi.org/10.56553/popets-2023-0020>

may include many unused ciphertext slots. With the cost of performing extra rotations, it is possible to pack more than one row in a single ciphertext.

Designing a good packing method is not straightforward (e.g., [16, 29, 44]) and the most efficient packing method may not be the trivial one (see above). Moreover, different optimization goals may lead to different packing, e.g., as shown in Table 3. As the size of the HE code increases, it becomes harder to find the optimal packing. For example, finding the best packing for a large NN inference algorithm is hard since the input is typically a four or five dimensional tensor, and the computation involves a long sequence of operations such as matrix multiplication and convolution.

Non-interactive HE. Two approaches for HE computations are client-aided and non-client-aided. In the client-aided approach, during the computation the server asks the data owner or the end-user for assistance. I.e., the user is asked to decrypt the intermediate ciphertext results, perform some minor computation tasks, and re-encrypt the data using HE. This approach was implemented in GAZELLE [29] and NGraph [8] using Multi Party Computations (MPC). It has the drawback that the client must stay online during the computation. Moreover, this setting poses some security concerns [4] or can even make it easier to perform model-extraction attacks [35]. To avoid these limitations, we focus on the non-client-aided approach where the computation is done entirely under encryption, without interaction.

Using non-client-aided designs require using HE-friendly NN architectures; these replace nonlinear layers such as ReLU and MaxPooling with other functions. Today, these conversions have become common practice (see survey [43]), and they present a time versus accuracy tradeoff that is mostly analyzed in AI-related works (e.g., [6]). Our framework can work with any polynomial activation function. Hence, our current and future users can decide how to balance time and accuracy by choosing the HE architecture that best suits their needs. We leave the accuracy discussion outside the scope of this paper. Here, we emphasize that our security-oriented framework leverages the SIMD property of HE-schemes, which can independently benefit from any AI-domain improvement in model accuracy.

Using HE-friendly models may require retraining models before using them. Nevertheless, we argue that the proliferation of the HE-domain, together with future AI improvements, will bring about more solutions that prefer training HE-friendly models directly, from day one. With our focus on non-client-aided designs, we provide customers with better security guarantees and improved client usability. The potential small cost in model accuracy is expected to get smaller over time.

Some recent HE compilers [8, 17] simplify the way users can implement NN solutions on encrypted data by allowing them to focus on the network and leaving the packing optimizations to the compilers. This is also the purpose of our tile tensor framework. It enables us to evaluate an HE-friendly version [6] of AlexNet [31] in only five minutes. To the best of our knowledge, this is the largest network to be implemented with a feasible running time, 128-bit security, in a non client-aided mode, and without bootstrapping (see [23]). In comparison, NGraph [8] reported their measurements for CryptoNets [21] or for MobileNetV2 [41] when using client-aided

design, and CHET [17] reported the results for SqueezeNet [26], which has 50× fewer parameters than AlexNet. Another experiment using NGraph and CHET was reported in [43] using Lenet-5 [32], which is also a small network compared to AlexNet. See a complete comparison at Section 8.

1.1 Our Contribution

Our contributions can be summarized as follows:

- *A tile tensor based framework.* We introduce a new packing-oblivious programming-framework that allows users to concentrate on the NN design instead of the packing decisions. This framework is simple and intuitive, and is available for non-commercial use in [39].
- *Tile tensors.* We introduce tile tensors, a new generalized packing method that intuitively covers many of the previous packing optimizations. A tile tensor offers the user the API of an ordinary tensor, and is implemented using elegant algorithms.
- *Packing optimizer.* We describe a packing optimizer that considers many different packing options. The optimizer estimates the time and memory needed to run a given function for every option, and reports the one that performs best per a given objective, whether latency, throughput, or memory.
- *A new method to compute convolution.* We provide a new packing method and a new implementation of the 2D convolutional layer, which is a popular building block in NNs. Our new packing and implementation are more efficient for large inputs than previous work. In addition, with this packing, we are able to efficiently compute a long sequence of convolution-layers.
- *Efficient HE-friendly version of AlexNet inference under encryption.* We implemented an HE-friendly version of AlexNet. To the best of our knowledge, this is the fastest non-client-aided evaluation of this network.
- *Packing notation.* We present a language for describing packing details. It covers several known packing schemes, as well as new ones, and allows easy and intuitive HE circuit design.

The rest of the paper is organized as follows. Section 2 describes the notation used in the paper, and some background terminology. Section 3 provides an overview of the tile tensor framework. We describe the tile tensors data structure in Section 4 and the packing optimizer in Section 6. Section 5 describes our novel convolution algorithm, and Section 7 reports the results of our experiments when using CryptoNets and AlexNet. In Section 8, we provide an extended comparison of our methods to the state-of-the-art methods. Section 9 concludes the paper.

2 BACKGROUND

2.1 Notation

We use the term *tensor* as synonymous with multi-dimensional array, as this is common in the AI domain. We denote the shape of a k -dimensional tensor by $[n_1, n_2, \dots, n_k]$, where $0 < n_i$ is the size of the i 'th dimension. For example, the shape of the 5×6 matrix M is $[5, 6]$. We sometimes refer to a tensor M by its name and shape $M[5, 6]$ or just by its name M when the context is clear. For a tensor

$R[n_1, \dots, n_k]$, we use $R(j_1, j_2, \dots, j_k)$ to refer to a specific element, where $0 \leq j_i < n_i$. We use uppercase letters for tensors.

We write matrix multiplication without a multiplication symbol, e.g., $M_1 M_2$ stands for the product of M_1 and M_2 . We denote the transpose operation of a matrix M by M^T and we use tags (e.g., M' , M'') to denote different objects. The operations $M_1 + M_2$ and $M_1 * M_2$ refer to element-wise addition and multiplication, respectively.

2.2 Tensor Basic Operations

2.2.1 Broadcasting and Summation. Here we define some commonly used tensor terms and functions.

DEFINITION 2.1 (COMPATIBLE SHAPES). *The tensors $A[n_1, \dots, n_k]$ and $B[m_1, \dots, m_k]$ have compatible shapes if $m_i = n_i$ or either $n_i = 1$ or $m_i = 1$, for $i \leq k$. Their mutual expanded shape is $[\max\{n_i, m_i\}]_{i \leq k}$.*

Remark 1. *When a tensor A has more dimensions than a tensor B , we can match their dimensions by expanding B with dimensions of size 1. This results in equivalent tensors up to transposition. For example, both tensors $V[b]$ and $V[b, 1]$ represent column vectors, while $V[1, b] = V^T$ represents a row vector.*

The broadcasting operation takes two tensors with compatible but different shapes and expands every one of them to their mutual expanded shape.

DEFINITION 2.2 (BROADCASTING). *For a tensor $A[n_1, \dots, n_k]$ and a tensor shape $s = [m_1, \dots, m_k]$ with $n_i \in \{1, m_i\}$ for each $i = 1, \dots, k$, the operation $C = \text{broadcast}(A, s)$ replicates the content of A along the r dimension m_r times for every $r = 1, \dots, k$ and $n_r = 1 < m_r$. The output tensor C is of shape s .*

Example 1. *The tensors $A[3, 4, 1]$ and $B[1, 4, 5]$ have compatible shapes. Their mutual expanded shape is $s = [3, 4, 5]$ and $\text{broadcast}(A, s)$ has the same shape s as $\text{broadcast}(B, s)$.*

We perform element-wise operations such as addition ($A + B$) and multiplication ($A * B$) on two tensors with compatible shapes A, B by first using broadcasting to expand them to their mutual expanded shape and then performing the relevant element-wise operation.

DEFINITION 2.3 (SUMMATION). *For a tensor $A[n_1, \dots, n_k]$, the operation $B = \text{sum}(A, t)$ sums the elements of A along the t -th dimension and the resulting tensor B has shape $[n_1, \dots, n_{t-1}, 1, \dots, n_k]$ and*

$$B(j_1, \dots, j_{t-1}, 0, \dots, j_k) = \sum_{l=0}^{n_t-1} A(j_1, \dots, j_{t-1}, l, \dots, j_k),$$

For all $j_i < n_i$ for $i \in \{1, 2, \dots, k\} \setminus \{t\}$,

Using broadcasting and summation we can define common algebraic operators. For two matrices $M_1[a, b]$, $M_2[b, c]$ and the column vector $V[b, 1]$, we compute matrix-vector multiplication using $M_1 V = \text{sum}(M_1 * V^T, 2)$, where M_1 and V^T have compatible shapes with the mutual expanded shape of $[a, b]$. We compute matrix-matrix multiplication using $M_1 M_2 = \text{sum}(M_1' * M_2', 2)$, where $M_1' = M_1[a, b, 1]$ and $M_2' = M_2[1, b, c]$.

2.2.2 Convolution. 2D convolution is a popular building block in NNs. Its input is often an images tensor $I[w_I, h_I, c, b]$ and a filters tensor $F[w_F, h_F, c, f]$ with the following shape parameters: width w_I, w_F , height h_I, h_F , and the number of image channels c (e.g., 3 for an RGB image). In addition, we compute the convolution for a batch of b images and for f filters. Informally, the convolution operator moves each filter in F as a sliding window over elements of I starting at position $(0, 0)$ and using strides of δ_w and δ_h . When the filter fits entirely in the input, the dot product is computed between the elements of the filter and the corresponding elements of I .

DEFINITION 2.4 (CONVOLUTION). *Let $I[w_I, h_I, c, b]$ and $F[w_F, h_F, c, f]$ be two input tensors for the convolution operator representing images and filters, respectively. The results of the operation $O = \text{conv2d}(I, F)$ is the tensor $O[w_O, h_O, f, b]$, where $w_O = \left\lfloor \frac{w_I - w_F + 1}{\delta_w} \right\rfloor$, $h_O = \left\lfloor \frac{h_I - h_F + 1}{\delta_h} \right\rfloor$, δ_h and δ_w are the strides and*

$$O(i, j, m, n) = \sum_{k=0}^{w_F-1} \sum_{l=0}^{h_F-1} \sum_{p=0}^{c-1} I(i \cdot \delta_w + k, j \cdot \delta_h + l, p, n) F(k, l, p, m). \quad (1)$$

In the degenerated case where $\delta_w = \delta_h = b = f = c = 1$, Equation (1) can be simplified to

$$O(i, j) = \sum_{k=0}^{w_F-1} \sum_{l=0}^{h_F-1} I(i + k, j + l) F(k, l). \quad (2)$$

2.3 Homomorphic Encryption

An HE scheme is an encryption scheme that allows us to evaluate any circuit, and any function, on encrypted data. A survey is available in [23]. Modern HE instantiations such as CKKS [13] rely on the hardness of the Ring-LWE problem, support SIMD operations, and operate over rings of polynomials. They include the following methods *Gen, Enc, Dec, Add, Mul* and *Rot* which we now briefly describe. The function *Gen* generates a secret key public key pair. The function *Enc* gets a message that is a vector $M[s]$ and returns a ciphertext. Here, s denotes the number of plaintext vector elements (*slot count*). It is determined during the key generation. The function *Dec* gets a ciphertext and returns an s -dimensional vector. For correctness we require $M = \text{Dec}(\text{Enc}(M))$. The functions *Add, Mul* and *Rot* are then defined as

$$\text{Dec}(\text{Add}(\text{Enc}(M), \text{Enc}(M'))) = M + M'$$

$$\text{Dec}(\text{Mul}(\text{Enc}(M), \text{Enc}(M'))) = M * M'$$

$$\text{Dec}(\text{Rot}(\text{Enc}(M), n))(i) = M((i + n) \bmod s)$$

An *approximation* scheme, such as CKKS [13], is correct up to some small error term, i.e., $|M(i) - \text{Dec}(\text{Enc}(m))(i)| \leq \epsilon$, for some $\epsilon > 0$ that is determined by the key.

2.4 Reducing Convolution to Matrix-matrix Multiplication

In HE settings it is sometimes useful to convert a convolution operation to a matrix-matrix multiplication by pre-processing the input before encrypting it. One such method is *image-to-column* [10],

which works as follows for the case $c = b = 1$. Given an image $I[w_I, h_I]$ and f filters $F[w_F, h_F, f]$, the operator $I', F' = im2col(I, F)$ computes a matrix $I'[w_O h_O, w_F h_F]$, where each row holds the content of a valid window location in I flattened to a row-vector, and $F'[w_F h_F, f]$ contains every filter of F flattened to a column-vector. Here, the tensor $O'[w_O h_O, f] = I'F'$ is a flattened version of the convolution result $O[w_O, h_O, f] = conv2d(I, F)$.

We propose a variant $I'', F'' = im2col'(I, F)$ that computes $I''[w_O h_O f, w_F h_F]$ by consecutively replicating f times every row of I' , and $F''[w_O h_O f, w_F h_F]$ by concatenating $w_O h_O$ times the matrix F'^T . The tensor $O''[w_O h_O f, 1] = sum(I'' * F'', 2)$ contains the convolution result $O[w_O, h_O, f]$. The advantage of this variant is that the output is fully flattened to a column vector, which is useful in situations where flattening is costly (e.g., in HE, where encrypted element permutations are required). The drawback of the *im2col* method is that it is impossible to perform two consecutive convolution operators without costly pre-processing in between. Moreover, this pre-processing increases the multiplicative-depth, deeming it impractical for networks such as AlexNet, where the multiplicative-depth was already near the underlying HE-library’s limit. In comparison, our convolution methods do not require pre-processing between two consecutive calls.

2.5 Threat Model

Our threat model involves three entities: An AI model owner, a cloud server that performs model inference on HE encrypted data using the pre-computed AI model, and users who send confidential data to the cloud for model inference. Among other, we take into account the following three models: a) the users and the model owner belong to the same organization, where both have access to the private key and are allowed to see the encrypted data; b) the user is the private-key owner. The model owner can use the public key to encrypt its model before uploading it to the cloud. This scenario involves a non-collusion assumption between the user and the cloud; c) the model-owner is the private-key holder. The user uses the public key to encrypt its samples before uploading them to the cloud. The model owner can decrypt and distribute the inference or post-processed results to the users. This scenario involves a non-collusion assumption between the model-owner and the cloud. In all cases, the cloud should learn nothing about the underlying encrypted data. In Scenarios b and c the users should not learn the model excluding privacy attacks, where the users try to extract the model training data through the inference results. We assume that communications between all entities are encrypted using a secure network protocol such as TLS 1.3, i.e., a protocol that provides confidentiality, integrity, and allows the model owner and the users to authenticate the cloud server. The model owner can send the model to the server either as plaintext or encrypted. When the model is encrypted, the server should learn nothing about the model but its structure. In both cases, we assume that the cloud is semi-honest (or honest-but-curious), i.e., that it evaluates the functions provided by the data owner and users without any deviation. We stress that our packing methods modify the data arrangement before encrypting it and thus do not affect the semantic security properties of the underlying HE scheme. Finally, in our experiments we target an HE solution with 128-bit level security.

3 OUR TILE TENSOR FRAMEWORK

HE libraries such as HELib [24] and SEAL [1] provide simple APIs for their users (e.g., encrypt, decrypt, add, multiply, and rotate). Still, writing an efficient program that involves more than a few operations is not always straightforward.

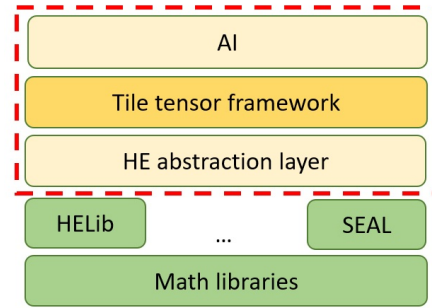


Figure 1: A simplified schematic of the layers in our library.

Providing users with the ability to develop complex and scalable HE-based programs is the motivation for developing higher-level solutions such as our library, NGraph [8], and CHET [17]. These solutions rely on the low-level HE libraries while offering additional dedicated optimizations, such as accelerating NNs inference on encrypted data.

Figure 1 provides a simplified schematic view of the layers we use in our library. The first two layers include the low-level HE libraries and their underlying software and hardware math accelerators. Our library [39] involves the three upper layers. At the bottom of these layers is the *HE abstraction layer*, which makes our library agnostic to the underlying HE library. The next layer is the *tile tensor framework layer*. It contains the tile tensor data structure (Section 4) that simplifies computation involving tensors, and the packing optimizer (Section 6) that searches for the most efficient tile tensor packing configuration for a given computation. The AI layer is built on top of these. It implements AI related functionality, such as reading NNs from standard file formats, encrypting their weights, and computing inference.

In this paper we focus on the tile tensor framework layer, and how it contributes to the optimization of NN inference computations. The optimizations this layer offers focus on packing and related algorithms, and can thus be combined with other types of optimizations in other layers. Note that our optimizer (Section 6) is simulation based, and therefore can take into account optimizations in any layer below this layer. We refer the reader to [40] for APIs and code of the HE abstraction layer.

4 TILE TENSORS

The tile tensor framework layer uses our new data structure, which named tile tensor. A tile tensor is a data structure that packs tensors in fixed size chunks, as required for HE, and allows them to be manipulated similar to regular tensors. It has an accompanying notation for describing the packing details.

We briefly and informally describe both data structure and notation. The notation is extensively used in the rest of the paper, and

it is summarized for quick reference in Table 1. For completeness, we provide formal definitions in Appendix B.

4.1 Tiling Basics

We start by describing a simple tiling process in which we take a tensor $A[n_1, n_2, \dots, n_k]$, and break it up into equal-size blocks that we call *tiles*, each having the shape $[t_1, t_2, \dots, t_k]$. In our implementation, a tile is an HE ciphertext.

Table 1: Tile tensor shape notation.

$\frac{n_i}{t_i}$	Basic tiling	n_i	Basic tiling, $t_i = 1$
$\frac{*}{t_i}$	Replication, $n_i = 1$	$\frac{n_i?}{t_i}$	Unknown values
$\frac{n_i \sim}{t_i}$	Interleaved tiling	$\frac{n_i \sim e_i}{t_i}$	Interleaved, given e_i

We construct a tensor $E[e_1, e_2, \dots, e_k]$, which we call the *external tensor*, such that each element of E is a tile, and $e_i = \left\lfloor \frac{n_i}{t_i} \right\rfloor$. Thus, $T = E(a_1, a_2, \dots, a_k)$ for $0 \leq a_i < e_i$, is a specific tile in E , and $T(b_1, b_2, \dots, b_k)$ for $0 \leq b_i < t_i$ is a specific slot inside this tile. An element of the original tensor $A(c_1, c_2, \dots, c_k)$ will be mapped to tile indices $a_i = \left\lfloor \frac{c_i}{t_i} \right\rfloor$, and indices inside the tile $b_i = c_i \bmod t_i$. All other slots in E that were not mapped to any element of A will be set to 0. Figure 2 demonstrates three examples for tiling a matrix $M[5, 6]$. In Figure 2c, for example, the shape of the external tensor is $[3, 2]$, and the tile shape is $[2, 4]$.

4.2 The Tile Tensor Data Structure

A tile tensor is a data structure containing an external tensor as described above, and *public* meta data called *tile tensor shape*. The tile tensor shape defines the shape of the tiles, the shape of the original tensor we started with, and some additional details about the organization of data inside the external tensor, which we describe later.

We use a special notation to denote tile tensor shapes. For example, $[\frac{n_1}{t_1}, \frac{n_2}{t_2}, \dots, \frac{n_k}{t_k}]$ is a tile tensor shape specifying that we started with a tensor of shape $[n_1, \dots, n_k]$ and tiled it using tiles of shape $[t_1, \dots, t_k]$. In this notation, if $t_i = 1$, then it can be omitted. For example, $[\frac{5}{1}, \frac{6}{8}]$ can be written $[5, \frac{6}{8}]$. Figure 2 shows three examples along with their shapes.

A tile tensor can be created using a *pack* operation that receives a tensor $A[n_1, \dots, n_k]$ to be packed and the desired tile tensor shape: $T_A = \text{pack}(A, [\frac{n_1}{t_1}, \dots, \frac{n_k}{t_k}])$. The *pack* operator computes the external tensor using the tiling process described above, and stores along-side it the tile tensor shape, to form the full tile tensor T_A . We can retrieve A back using the *unpack* operation: $A = \text{unpack}(T_A)$. As with regular tensors, we sometimes refer to a tile tensor T_A together with its shape: $T_A[\frac{n_1}{t_1}, \dots, \frac{n_k}{t_k}]$.

4.3 Replication

For some computations it is useful to have the tensor data replicated several times inside the tile slots. The tile tensor shape indicates this by using the $\frac{*}{t_i}$ notation. It implies that $n_i = 1$, but each element of the original tensor is replicated t_i times along the i 'th dimension. When packing a tensor $A[n_1, \dots, n_k]$ and $n_i = 1$, and

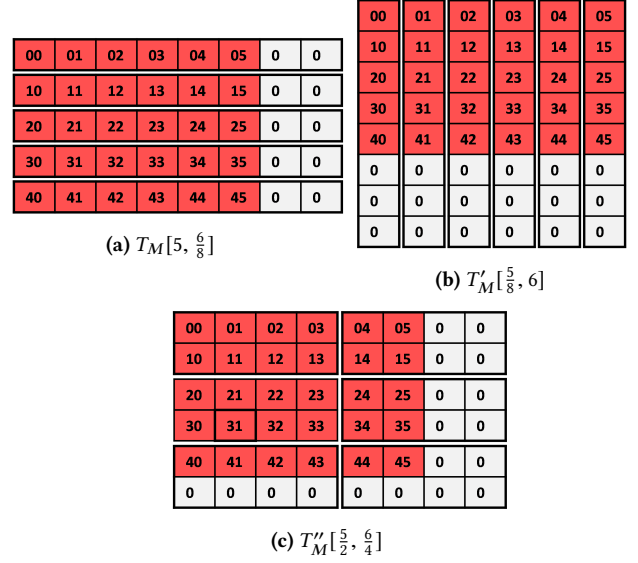


Figure 2: Packing an $M[5, 6]$ tensor into the tile tensors T_M (Panel a), T'_M (Panel b), T''_M (Panel c) with 8-slot tiles of shape $[1, 8]$, $[8, 1]$, and $[2, 4]$, respectively. For these, the external tensor shape is $[5, 1]$, $[1, 6]$, and $[3, 2]$, respectively. The value of $M[5, 6]$ in the i th row and j th column is the value ij .

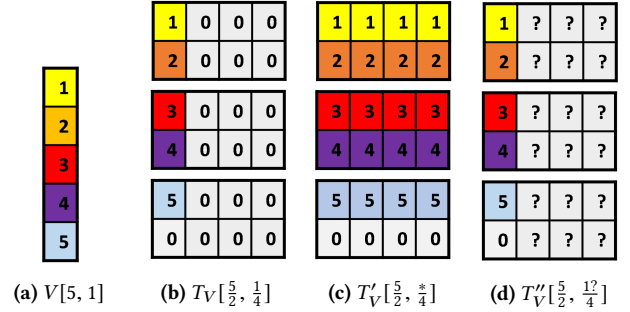


Figure 3: Packing $V[5, 1]$ into tile tensors using different tile tensor shapes. Every rectangle represents a tile. Panel (c) demonstrates tiling with replication, where the packing process computes $V' = \text{broadcast}(V, [5, 4])$ and tiles V' . Panel (d) demonstrates unknown values along the second dimension using question mark symbols.

with a tile tensor shape specifying $\frac{*}{t_i}$, then the packing operation performs $\text{broadcast}(A, [n_1, \dots, t_i, \dots, n_k])$ and tiles the result. The unpacking process shrinks the tensor back to its original size. The replications can either be ignored, or an average of them can be taken; this is useful if the data is stored in a noisy storage medium, as in approximate HE schemes. Figure 3 demonstrates packing $V[5, 1]$ with different tile tensor shapes. Note that we only replicate inside a tile. When broadcasting requires replicating whole tiles it is possible to hold only one copy of it and use it multiple times.

4.4 Unknown Values

When tensors are packed into tile tensors, unused slots are filled with zeroes, as shown in Figure 2. Section 4.5 shows how to apply operators on tile tensors, and then unused slots might get filled with arbitrary values. Although these unused slots are ignored when the tile tensor is unpacked, the presence of arbitrary values in them can still impact the validity or performance of applying additional operators. To reflect this state, the tile tensor shape contains an additional flag per dimension, denoted by the symbol “?”, indicating the presence of unknown values.

Figure 3d shows a tile tensor with the shape $[\frac{5}{2}, \frac{1?}{4}]$. The “?” in the second dimension indicates that whenever we exceed the valid range of the packed tensor along this dimension, we may encounter arbitrary unknown values. However, it still holds that $V = \text{unpack}(T_V)$, as these unused slots are ignored.

4.5 Operators

Tile tensor operators are homomorphic operations between tile tensors and the packed tensors they contain. For two tile tensors T_A and T_B , and a binary operator \odot , it holds that $\text{unpack}(T_A \odot T_B) = \text{unpack}(T_A) \odot \text{unpack}(T_B)$. Unary operators are similarly defined.

Remark 2. *The above operations involve two levels of homomorphism. One homomorphism is between direct operations on ordinary tensors A, B and operating on the tile tensors T_A, T_B that contain them. The tile tensor operators are implemented by performing operations on encrypted tiles. Due to the properties of homomorphic encryption (see Section 2.3), these are homomorphic to ordinary operations on the plaintext content of the tiles.*

Binary elementwise operators are implemented by applying the operator on the external tensors tile-wise, and the tile tensor shape is updated to reflect the shape of the result. If the inputs have identical shapes then so do the results, e.g., $T_M''[\frac{5}{2}, \frac{6}{4}]$ of Figure 2c can be multiplied with an identically packed matrix $T_N[\frac{5}{2}, \frac{6}{4}]$, resulting in $T_R[\frac{5}{2}, \frac{6}{4}]$, where $R = M * N$. As with regular tensors, the tile tensor shapes need not be identical, but compatible. Compatible tile tensor shapes have the same number of dimensions, and for each dimension specification they are either identical, or one is $\frac{*}{t_i}$ and the other is $\frac{n_i}{t_i}$. The intuition is that if the tensor is already broadcast inside the tile, it can be further broadcast to match any size by replicating the tile itself. For example, for $T_V'[\frac{5}{2}, \frac{*}{4}]$ of Figure 3c we can compute $T_M'' * T_V'$ resulting in $T_R' = [\frac{5}{2}, \frac{6}{4}]$. We can also compute $T_M'' + T_V'$, but this results in $T_R''[\frac{5}{2}, \frac{6?}{4}]$, i.e., with unknown values in unused slots along the second dimension. This occurs because in T_V' this dimension is filled with replicated values, and after the addition they fill the unused slots of the result. Computing $T_M'' * T_V$ (for T_V from Figure 3b) is illegal because their shapes are not compatible. For the full set of rules for elementwise operators see Appendix B.

The *sum* operator is also defined homomorphically: $\text{unpack}(\text{sum}(T_A, i)) = \text{sum}(\text{unpack}(T_A), i)$. It works by summing over the external tensor along the i 'th dimension, then by summing inside each tile along the i 'th dimension. In an HE environment, the latter summation requires using the rotate-and-sum algorithm [3]. Generally, the sum operator reduces the i 'th dimension and the resulting tile tensor shape changes to $\frac{1?}{t_i}$. However, there are

some useful special cases. If $t_i = 1$, then it is reduced to $\frac{1}{1}$ or simply 1. When i is the smallest i such that $t_i > 1$, the dimension reduces to $\frac{*}{t_i}$, i.e., the sum results are replicated. This is due to properties of the rotate-and-sum algorithms. It is a useful property, since this replication is sometimes needed for compatibility with another tile tensor. For example, let T_A be a tile tensor with the shape $[4, \frac{3}{8}, \frac{5}{16}]$. Then, $\text{sum}(T_A, 1)$ is of shape $[1, \frac{3}{8}, \frac{5}{16}]$; $\text{sum}(T_A, 2)$ is of shape $[4, \frac{*}{8}, \frac{5}{16}]$ and $\text{sum}(T_A, 3)$ is of shape $[4, \frac{3}{8}, \frac{1?}{16}]$.

Three other operators used in this paper do not change the packed tensor, just the external tensor and tile tensor shape. The *clear*(T_A) operator clears unknown values by multiplying with a mask containing ones for all used slots, i.e., it removes the “?” from the tile tensor shape. For example, $\text{clear}(T_V''[\frac{5}{2}, \frac{1?}{4}]) = T_V[\frac{5}{2}, \frac{1}{4}]$ (see Figure 3). The *rep*(T_A, i) operator assumes the i 'th dimension is $\frac{1}{t_i}$, and replicates it to $\frac{*}{t_i}$, using a rotate-and-sum algorithm. The *flatten*(T_A, i, j) operator flattens dimensions i through j assuming they are all replicated. This is done trivially by just changing the meta data, e.g., $\text{flatten}(T_A[\frac{3}{4}, \frac{*}{8}, \frac{*}{2}, \frac{5}{32}], 2, 3)$ results with $T_A'[\frac{3}{4}, \frac{*}{16}, \frac{5}{32}]$.

4.6 Higher Level Operators

Using elementwise operators and summation, we can perform various algebraic operations on tile tensors.

Matrix-vector multiplication. Given a matrix $M[a, b]$ and a vector $V[b]$, we reshape V to $V[1, b]$ for compatibility, and pack both tensors into tile tensors as $T_M[\frac{a}{t_1}, \frac{b}{t_2}]$, and $T_V[\frac{*}{t_1}, \frac{b}{t_2}]$, for some chosen tile shape $[t_1, t_2]$. We can multiply them using:

$$T_R[\frac{a}{t_1}, \frac{1?}{t_2}] = \text{sum}(T_M[\frac{a}{t_1}, \frac{b}{t_2}] * T_V[\frac{*}{t_1}, \frac{b}{t_2}], 2). \quad (3)$$

The above formula works for any value of a, b, t_1, t_2 . This is because the tile tensor shapes of T_M and T_V are compatible, and therefore, due to the homomorphism, this computes $R[a, 1] = \text{sum}(M[a, b] * V[1, b], 2)$, which produces the correct result as explained in Section 2.2.

A second option is to initially transpose both M and V and pack them in tile tensors $T_M[\frac{b}{t_1}, \frac{a}{t_2}]$ and $T_V[\frac{b}{t_1}, \frac{*}{t_2}]$. Now we can multiply them as:

$$T_R[\frac{*}{t_1}, \frac{a}{t_2}] = \text{sum}(T_M[\frac{b}{t_1}, \frac{a}{t_2}] * T_V[\frac{b}{t_1}, \frac{*}{t_2}], 1). \quad (4)$$

This computes the correct result using the same reasoning as before. The benefit here is that the result $T_R[\frac{*}{t_1}, \frac{a}{t_2}]$ is replicated along the first dimension due to the properties of the sum operator (Section 4.5). Thus, it is ready to play the role of T_V in Formula 3, and we can perform two matrix-vector multiplications consecutively without any processing in between. The output of Formula 3 can be processed to fit as input for Formula 4 using $\text{rep}(\text{clean}(T_R), 2)$.

Matrix-matrix multiplication. The above reasoning easily extends to matrix-matrix multiplication as follows. Given matrices $M_1[a, b]$ and $M_2[b, c]$, we can compute their product using either of the next two formulas, where in the second one we transpose M_1 prior to packing. As before, the result of the second fits as input to the first.

$$T_R[\frac{a}{t_1}, \frac{1?}{t_2}, \frac{c}{t_3}] = \text{sum}(T_{M_1}[\frac{a}{t_1}, \frac{b}{t_2}, \frac{*}{t_3}] * T_{M_2}[\frac{*}{t_1}, \frac{b}{t_2}, \frac{c}{t_3}], 2). \quad (5)$$

$$T_R\left[\frac{*}{t_1}, \frac{a}{t_2}, \frac{c}{t_3}\right] = \text{sum}(T_{M_1}\left[\frac{b}{t_1}, \frac{a}{t_2}, \frac{*}{t_3}\right] * T_{M_2}\left[\frac{b}{t_1}, \frac{*}{t_2}, \frac{c}{t_3}\right], 1). \quad (6)$$

Example 2. The product $R[100, 60] = \prod_{i=1}^4 M_i$ of the four matrices $M_1[100, 90]$, $M_2[90, 80]$, $M_3[80, 70]$, and $M_4[70, 60]$ is computed by packing the matrices in tile tensors $T_{M_1}\left[\frac{90}{t_1}, \frac{100}{t_2}, \frac{*}{t_3}\right]$, $T_{M_2}\left[\frac{90}{t_1}, \frac{80}{t_2}, \frac{*}{t_3}\right]$, $T_{M_3}\left[\frac{70}{t_1}, \frac{80}{t_2}, \frac{*}{t_3}\right]$, and $T_{M_4}\left[\frac{70}{t_1}, \frac{*}{t_2}, \frac{60}{t_3}\right]$ and computing

$$T_{X_1}\left[\frac{90}{t_1}, \frac{1?}{t_2}, \frac{60}{t_3}\right] = \text{sum}(T_{M_2} * (\text{sum}(T_{M_3} * T_{M_4}, 1)), 2)$$

$$T_R\left[\frac{*}{t_1}, \frac{100}{t_2}, \frac{60}{t_3}\right] = \text{sum}(T_{M_1} * (\text{rep}(\text{clean}(T_{X_1}), 2)), 1)$$

4.7 Interleaved Tiling

Another option for tiling is denoted by the symbol “~” in the tile tensor shape. This symbol indicates that the tiles do not cover a contiguous block of the tensor, but are spread out in equal strides. If the dimensions are interleaved, an element of the original tensor $A(c_1, c_2, \dots, c_k)$ will be mapped to tile indices $a_i = c_i \bmod e_i$, and indices inside the tile $b_i = \left\lfloor \frac{c_i}{e_i} \right\rfloor$ (where e_i is the size of the external tensor, see Section 4.1). See Figures 4a and 4b for an example.

For each dimension, we can specify separately whether it is interleaved or not. For example, in $\left[\frac{?}{2}, \frac{6?}{4}\right]$ only the second dimension is interleaved. Also, although with basic tiling it holds that $e_i = \left\lceil \frac{h_i}{t_i} \right\rceil$, for interleaved tiling it is sometimes useful to have larger values for e_i . In this case, this value can be explicitly stated using the notation: $\frac{n_i \sim e_i}{t_i}$.

Interleaved dimensions fit seamlessly with all the operators described above, and are useful for computing convolutions. See Section 5 for more details.

5 CONVOLUTION USING TILE TENSORS

In this section we describe our novel method to compute convolution. Compared to previous work (e.g. [29, 44]) our method has two advantages: (i) it is more efficient when the input is a large image and (ii) it allows efficient computation of consecutive convolution layers in an HE only system.

We first consider in Section 5.1 the convolution problem in its simplest form: a single, one-channel image, and a single filter $F[w_F, h_F]$. We extend it to convolution with strides and multiple channels, filters, and batching in Section 5.2.

5.1 Convolution with Interleaved Dimensions

We now show how interleaved dimensions (see Section 4.7) can be used to efficiently compute convolution.

Figures 4a and 4b show a matrix $M[6, 6]$ packed in the tile tensor $T_M\left[\frac{6?}{2}, \frac{6?}{4}\right]$. Here, the tile shape is $[2, 4]$ and the external tensor shape is $[3, 2]$. Every tile contains a 2×4 sub-matrix, but instead of being contiguous, it is a set of elements spaced evenly in the matrix. We use the same color to denote elements that are mapped to the same slot in different tiles. For example, the elements 00, 01, 10, 11, 20, 21 are all colored red as they are mapped to slot (0, 0) in 6 different tiles. This coloring divides the original matrix into contiguous regions.

The interleaved packing allows for a more efficient implementation of Equation 2 with respect to runtime and storage. Intuitively, we use the SIMD to compute multiple elements of the output in a single operation. The filter is packed simply as $T_F[w_F, h_F, \frac{*}{t_1}, \frac{*}{t_2}]$. I.e., it has $w_F h_F$ tiles, each containing one value of the filter in all slots. This allows multiplying each image tile with each value of the filter.

For example, Figure 4c shows a computation of the convolution output when the filter is placed at the top left position. The SIMD nature of the computation computes the output in other regions as well. The result is a single tile, where each slot contains the convolution result of the corresponding region, such that this tile is packed in the same interleaved packing scheme as the input tiles.

A more complicated example is given in Figure 4d. Here the filter is placed one pixel to the right. As a result, the filter needs to be multiplied by elements that appear in different regions, i.e. they are mapped to slots of different indices. In this case we need to rotate the tiles appropriately. For example, placing the filter with its upper left corner on pixel (0, 1), the convolution is computed using the (0, 0) slot of tiles (0, 1) and (1, 1) and slot (0, 1) of tiles (0, 0) and (1, 0). The latter two are therefore rotated to move the required value to slot (0, 0) as well.

The total cost of convolution when using this packing is summarized in the following lemma.

Lemma 1. Let s be the number of slots in a ciphertext. Then, given an input image $I[w_I, h_I]$ and a filter $F[w_F, h_F]$, packing I as $T_I\left[\frac{w_I \sim ?}{t_1}, \frac{h_I \sim ?}{t_2}\right]$ and the filter as $T_F[w_F, h_F, \frac{*}{t_1}, \frac{*}{t_2}]$, convolution can be computed using: $O(\lceil w_I h_I w_F h_F / s \rceil)$ multiplications, and $O(w_F \left\lceil \frac{w_I}{t_1} \right\rceil + h_F \left\lceil \frac{h_I}{t_2} \right\rceil + w_F h_F)$ rotations. The input is encoded in $O(w_I h_I / s)$ ciphertexts.

PROOF. See Appendix E. \square

The output of the convolution is a tile tensor $T_O\left[\frac{w_O \sim ?}{t_1}, \frac{h_O \sim ?}{t_2}\right]$. The unknown values are introduced by filter positions that extend beyond the image, as shown in Figure 4d. Note further that the external sizes $e_1 = \left\lceil \frac{w_I}{t_1} \right\rceil$ and $e_2 = \left\lceil \frac{h_I}{t_2} \right\rceil$ of the tile tensor T_I remain the same in T_O , and they may be larger than those actually required to hold the tensor $O[w_O, h_O]$. Hence, a more accurate depiction of T_O 's shape is $T_O\left[\frac{w_O \sim e_1?}{t_1}, \frac{h_O \sim e_2?}{t_2}\right]$, but we will ignore this technicality from here on.

5.2 Handling Strides, Batching and Multiple Channels and Filters

In this section we extend the simple description given in Section 5.1. We first show how our convolution algorithm extends to handle multiple channels, multiple filters, and batching. We then show how we handle striding.

Let the input be a tensor of images $I[w_I, h_I, c, b]$, where c is the number of channels and b is the batch size. Then we pack it as $T_I\left[\frac{w_I \sim ?}{t_1}, \frac{h_I \sim ?}{t_2}, \frac{c}{t_3}, \frac{b}{t_4}, \frac{*}{t_5}\right]$. Also, we pack the filters $F[w_F, h_F, c, f]$, where f is the number of filters, as $T_F[w_F, h_F, \frac{*}{t_1}, \frac{*}{t_2}, \frac{c}{t_3}, \frac{*}{t_4}, \frac{f}{t_5}]$, where $t_i \in \mathcal{N}$ and $\prod t_i = s$.

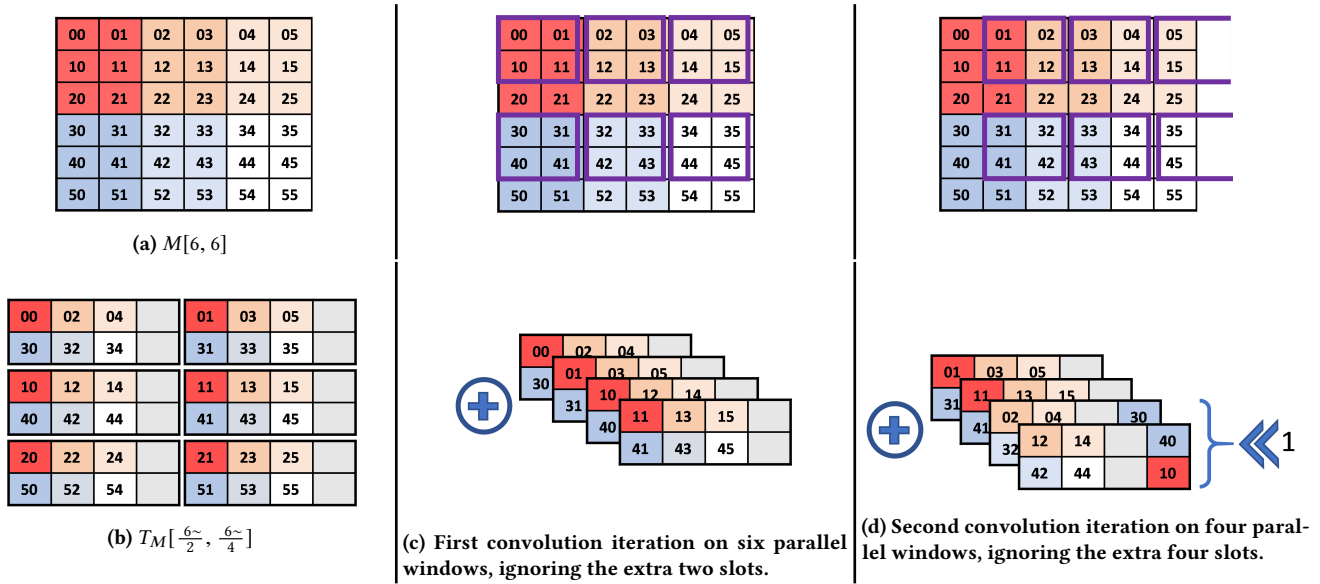


Figure 4: Packing a matrix $M[6, 6]$ into $T_M[\frac{6}{2}, \frac{6}{4}]$ and performing two 8-parallel kernel evaluations using a 2×2 filter (purple). The upper figures illustrate the convolution operator on the $M[6, 6]$ matrix. The lower figures illustrate the same operation using tiles representation. Here, \oplus denotes component-wise summation of tiles and $\ll 1$ denotes left circular rotation by 1.

The convolution is computed similarly to the description in Section 5.1, multiplying tiles of T_I with the appropriate tiles of T_F . The result is a tile tensor of shape $T_O[\frac{w_O \sim ?}{t_1}, \frac{h_O \sim ?}{t_2}, \frac{c}{t_3}, \frac{b}{t_4}, \frac{f}{t_5}]$. Summing over the channel (i.e., third) dimension using $\mathcal{O}(\lfloor \frac{w_O}{t_1} \rfloor \lfloor \frac{h_O}{t_2} \rfloor \lfloor \frac{b}{t_4} \rfloor \lfloor \frac{f}{t_5} \rfloor \log t_3)$ rotations, we obtain $T_O[\frac{w_O \sim ?}{t_1}, \frac{h_O \sim ?}{t_2}, \frac{1?}{t_3}, \frac{b}{t_4}, \frac{f}{t_5}]$, where the $\log t_3$ factor comes from the sum-and-rotate algorithm [3].

For bigger strides, $\delta_h > 1$ (resp. $\delta_w > 1$), we require that either $t_1 = 1$ (resp. $t_2 = 1$) or $\lfloor \frac{h_I}{t_2} \rfloor \bmod \delta_h = 0$ (resp. $\lfloor \frac{w_I}{t_1} \rfloor \bmod \delta_w = 0$). Then, our implementation trivially skips δ_w ciphertexts in every row and δ_h ciphertexts in every column.

5.3 A Sequence of Convolutions

In this section we discuss how to implement a sequence of multiple convolution layers. This is something that is common in neural networks. One of the advantages of our tile tensor method is that the output of one convolution layer can be easily adjusted to be the input of the next convolution layer.

Assume we are given an input batch tensor $I[w_I, h_I, c, b]$ and a sequence of convolution layers with the l 'th layer having a filter tensor $F^l[w_F^l, h_F^l, c^l, f^l]$. For the first layer we have $c^1 = c$, and for $l > 1$ we have $c^l = f^{l-1}$. As before, we pack the input tensor as $T_I[\frac{w_I \sim ?}{t_1}, \frac{h_I \sim ?}{t_2}, \frac{c}{t_3}, \frac{b}{t_4}, \frac{*}{t_5}]$. For odd layers, $l = 2\ell + 1$, we pack the filter tensor as before $T_F^l[w_F^l, h_F^l, \frac{*}{t_1}, \frac{*}{t_2}, \frac{c}{t_3}, \frac{*}{t_4}, \frac{f^l}{t_5}]$. The output is then $T_O[\frac{w_O \sim ?}{t_1}, \frac{h_O \sim ?}{t_2}, \frac{1?}{t_3}, \frac{b}{t_4}, \frac{f^l}{t_5}]$. For even layers, $l = 2\ell$, we introduce this packing for the filters: $T_F^l[w_F^l, h_F^l, \frac{*}{t_1}, \frac{*}{t_2}, \frac{f^l}{t_3}, \frac{*}{t_4}, \frac{c}{t_5}]$.

As can be seen, the shapes of the layer outputs do not match the shapes of the inputs of the subsequent layers. We now show how to solve it and thus allow for a sequence of convolution layers.

To make an output of an odd layer suitable for the next even layer, we clear the unknowns by multiplying with a mask and then replicate the channel dimension. We then get a tile tensor of this shape: $T_O[\frac{w_O^l \sim ?}{t_1}, \frac{h_O^l \sim ?}{t_2}, \frac{*}{t_3}, \frac{b}{t_4}, \frac{f^l}{t_5}]$, which matches the input format of the next layer since $f^l = c^{l+1}$. To make an output of an even layer suitable for the next odd layer, we similarly clean and replicate along the filter dimension.

We note that changing the order of the dimensions leads to a small improvement. The improvement comes because summing over the first dimension ends up with a replication over this dimension. Therefore, setting the channel dimension as the first dimension saves us the replication step when preparing the input to an even layer. We can skip cleaning as well, since the unknown values along the image width and height dimensions do no harm. Alternatively, the filter dimension can be set as first and then the replication step can be skipped when preparing the input for an odd layer.

5.4 Naïve Convolution Methods

The above method reduces to a simple method known by various names such as SIMD packing when $t_1 = t_2 = t_3 = t_5 = 1$. In this case, every element in the tensors for the images and filters is stored in a separate ciphertext, and the slots are only used for batching. In this paper, we further use the reduction to matrix multiplication as described in Section 2.4. It is applicable only for NNs with one convolutional layer.

6 THE OPTIMIZER

The packing optimizer is responsible for finding the most efficient packing arrangement for a given computation, as well as the optimal configuration of the underlying HE library. This relieves the users

from the need to handle these HE related complexities. The users only need to supply the model architecture e.g., a NN architecture, in some standard file format. The optimizer automatically converts it to an HE computation with optimal packing and optimal HE library configuration. Users can further supply constraints such as the required security level or maximal memory usage, and choose an optimization target, whether to optimize for CPU time, latency or throughput, or optimize for memory usage. Our optimizer has many similarities to the one of CHET. However, it has one principal difference, it operates over and understands our concept of tile tensors. This allows it, for example, to explore different intermediate batching levels and to suggest different tradeoffs between latency and amortized latency, which was not discussed or analyzed in CHET.

The optimizer needs to choose among different possible configurations of the HE library, as well as different packing techniques to support certain operators (see Section 5). It also chooses the tile shape, i.e., the values of t_1, t_2, \dots , in the tile tensor shapes. For example, consider an HE scheme configured to have 16,384 slots in each ciphertext. Since our convolution operator uses five-dimensional tiles, the number of possible tuples t_1, \dots, t_5 such that $\prod_i t_i = 16,384$ is $\binom{\log_2(16,384)+5-1}{5-1} = 3,060$. We use the term *configuration* to refer to a complete set of options the optimizer can choose (HE configuration parameters, tile shape, and other packing options).

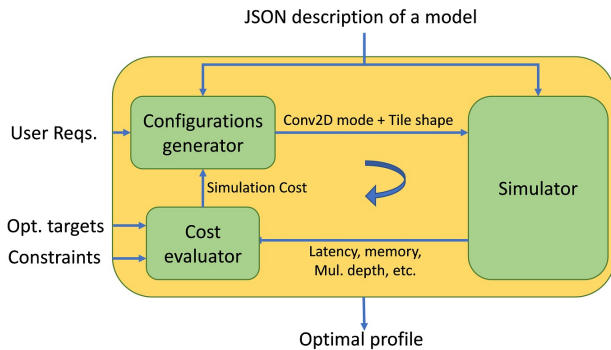


Figure 5: Packing optimizer

Figure 5 presents a schematic of the packing optimizer, and Alg. 1 in Appendix C presents the process using pseudo code. It contains three main units: the configuration generator, the cost evaluator, and the simulator. The user provides a JSON file m that contains the model architecture. The configuration generator generates a list of all possible configurations (GenerateConfs at Step 4, see also Alg 2 in Appendix C), including the packing details and HE configuration details applicable for this architecture. The simulator unit tests every such configuration and outputs the following data (*res* at Step 6) for each: the computation time of the different stages including encrypting the model and input samples, running inference, and decrypting the results; the throughput; the memory usage of the encrypted model; input; and output; and more. The optimizer passes this data to the cost evaluator for evaluation (Steps 7-16). Finally, it returns the configuration option that yields the optimal cost to the user (among the tested configurations), together with the simulation output profile (Step 17).

Configuration generator. The configuration generator unit receives the model architecture, and generates all applicable configurations for it. For example, if the model has a single convolutional layer it will generate three basic configurations with three possible convolution implementations: the *im2col* based method, and the two options of our novel method (see Section 5). If the model has multiple convolutional layers, the *im2col* based method will not be applicable. From each of these three basic configurations the generator will create multiple complete configurations by exploring all possible tile shapes. The generator explores possible tile shape using one of two strategies. The first involves brute forcing over all valid options for tile shapes (Alg. 1). Since these may be numerous, a second strategy searches using a “steepest ascent hill climbing” local search algorithm, which requires replacing Step 4 in Alg. 1 with an adaptive GenerateConf function call.

The local search starts with a balanced tile shape, where the number of slots in every dimension is of the same order. This is a heuristic designed to avoid evaluating tile shapes that are likely to be computationally costly at the beginning of the search. We then iteratively evaluate all the neighbor tile shapes of the current shape and continue to the best-improving neighbor as long as one exists. We consider two tile shapes as neighbors if we can obtain one shape from the other by multiplying or dividing the size of some of its dimensions by two. It is possible to use other small multipliers and dividers, but these may involve unnecessary complex computations. We consider one shape as better than another shape based on the costs received from the cost evaluator. Using the local search algorithm highly speeds up the search process and we found empirically that it often results in a global optimum. This was the case in our AlexNet, SqueezeNet-CIFAR and CryptoNets benchmarks.

Simulator. The simulator receives as inputs the model architecture and a configuration option. At this stage, we can evaluate the configuration by running it on encrypted input under HE. To reduce computational costs, the simulator uses pre-calculated benchmark values such as the CPU time of every HE operation and the memory consumption of a tile (i.e., the memory consumption of a single ciphertext). Then, it evaluates the model on mockup tile tensor objects using these benchmarks. These mockup tile tensors contain only meta data and gather performance statistics. Using this approach, the simulator can simulate an inference operation several order-of-magnitudes faster than when running the complete model on encrypted data. Appendix D reports the simulator accuracy on AlexNet. We stress that performing an analytical evaluation instead of an empirical one over a large NN requires considering many parameters, which is eventually equivalent to running our simulator using the mockup tile tensors. To allow scientific comparisons of different packings, the simulator also outputs the total number of rotations and multiplications per evaluated circuit and per layer.

Cost evaluator. The cost evaluation unit evaluates the simulator output data considering the constraints and optimization targets provided by the user. After testing all possible configurations, the highest scoring configuration(s) is sent back as output to the user.

Evaluating the optimizer performance. To demonstrate the advantage of using both the local search algorithm and the simulator, we performed experiments using AlexNet (see Section 7.2 and Appendix A). Here, we fixed the number of slots to 16,384, the

minimal feasible size for a NN that deep, and set the batch size to 1. The number of configuration options was 1360, with 680 different tile shapes for each convolution packing method. An exhaustive search that uses simulations took 5.1 minutes. In contrast, the local search algorithm took only 6.4 seconds and returned the same result. It did so after evaluating only 40 tile shapes.

Running the local search method on actual encrypted data took 9.95 hours. Using the simulator time estimations, we predict that exhaustive search on encrypted data would take ~ 167 days (assuming unlimited memory). This demonstrates the importance of the mockup-based simulator.

7 EXPERIMENTAL RESULTS

Our experiments involve the model weights of a small NN (CryptoNets [21]) and a large NN (AlexNet [31]) that we trained on the MNIST [33], and COVIDx CT-2A [22] data sets, respectively. We report the results of performing model inference using these weights in encrypted and unencrypted forms. We use AlexNet to demonstrate the power of our framework and CryptoNets to demonstrate the effect of different packing on the computation performance and memory. Technical details of the environment we used for the experiments are described in Appendix A.

7.1 CryptoNets

The CryptoNets [21] architecture and the HE parameters that we use in our experiments are described in Appendix A.1. Generally speaking, this network has a convolutional layer followed by two fully connected layers.

Table 2 reports the latency and memory usage for performing a model inference with different tile shapes when $t_3 = b = 1$. For brevity, we only consider t_1 to be at the extreme points (e.g., $t_1 = 1, 8, 192$) or t_1 value that led to best performing solution, and some additional samples. The best latency and memory usage are achieved for $t_1 = 32$, which allows packing the tensors I, F, W_1 using the minimal number of tiles.

Table 3 reports the latency, amortized latency, and memory usage for performing a model inference with different $t_3 = b$ values. For every such value, we only report the t_1, t_2 values that led to the optimal solutions. Unlike the case where $b = 1$, here every choice of t_3 leads to a different trade-off between the performance measures. For example, when increasing t_3 , the latency and memory consumption increase, but the per-sample amortized latency decreases. The encryption and decryption time also increase with t_3 , except for the case $t_3 = 8, 192$, where we use the naïve SIMD convolution operator.

7.2 AlexNet Benchmark

For this benchmark, we used a variant of AlexNet network [31] that includes 5 convolution layers, 3 fully connected layers, 7 ReLU activations, 3 BatchNormalization layers, and 3 MaxPooling layers. Following [5, 6], we created a CKKS-compliant variant of AlexNet by replacing the ReLU and MaxPooling components with a quadratic polynomial activation and AveragePooling correspondingly along with some additional changes. We trained and tested it on the COVIDx CT-2A dataset, an open access benchmark of CT images

Table 2: Running a model inference with different tile shapes $[t_1, t_2, t_3]$ when $t_3 = b = 1$. The reported values are: the inference latency, the encryption and decryption time, and the memory usage peak.

t_1	t_2	t_3	Latency (sec)	Enc+Dec (sec)	Memory (GB)
1	8,192	1	0.86	0.04	1.58
8	1,024	1	0.56	0.04	0.76
32	256	1	0.56	0.04	0.73
64	128	1	0.57	0.04	0.77
128	64	1	0.61	0.04	0.94
256	32	1	0.68	0.05	1.37
1,024	8	1	1.93	0.14	3.17
8,192	1	1	11.10	0.80	14.81

Table 3: Running a model inference with different tile shapes $[t_1, t_2, t_3]$, reporting only the optimal t_1 and t_2 choices for a range of different $t_3 = b$ values. The reported values are: the inference latency, the amortized latency (latency/ b), the encryption and decryption time, and the memory usage peak.

t_1	t_2	t_3	Latency (sec)	Amortized Latency (sec)	Enc+Dec (sec)	Memory (GB)
32	256	1	0.56	0.56	0.04	0.73
16	128	4	0.56	0.14	0.05	1.20
8	64	16	0.6	0.037	0.10	2.49
4	32	64	0.95	0.015	0.24	6.62
1	32	256	1.94	0.008	0.70	16.38
1	8	1,024	5.6	0.0055	2.68	61.45
1	2	4,096	21.57	0.0053	12.55	242.46
1	1	8,192	41.32	0.005	1.29	354.47

Table 4: AlexNet executed in our framework with different configurations. See configuration description in Section 7.2

Config.	Latency (sec)	Amortized Latency (sec)	Enc+Dec (sec)	Memory (GB)
PT-Latency	298.8	298.8	5.289	163
PT-TP	785	98.1	5.102	604
CT-Latency	350.4	350.4	5.018	252
CT-TP	804.8	201.2	5.671	739

designed by [22]. We resized the images to $224 \times 224 \times 3$ to fit the input size expected by AlexNet. See more details in Appendix A.1.

For the convolutional layers, we used the packing methods described in Section 5.3. The biases were packed in 5-dimensional tile tensors with compatible shapes, allowing us to add them to the convolution outputs. The fully connected layers were handled using the matrix-matrix multiplication technique of Section 4.6. The input to these layers arrives from the convolutional layers as a 5-dimensional tile tensor, $[\frac{*}{t_1}, \frac{1}{t_2}, \frac{1}{t_3}, \frac{9216}{t_4}, \frac{b}{t_5}]$. Therefore, the first fully connected layer is packed in 5 dimensions as well: $[\frac{4,096}{t_1}, \frac{1}{t_2}, \frac{1}{t_3}, \frac{9216}{t_4}, \frac{*}{t_5}]$. Its output, $[\frac{4,096}{t_1}, \frac{1}{t_2}, \frac{1}{t_3}, \frac{1}{t_4}, \frac{b}{t_5}]$, is replicated along dimensions 2

through 4, then flattened using the flatten operator to $[\frac{4 \cdot 096}{t_1}, \frac{*}{t_2 t_3 t_4}, \frac{b}{t_5}]$, from which we can continue normally.

We measured the accuracy of running vanilla AlexNet [31] and the HE-friendly AlexNet [6] using PyTorch¹ over a plaintext test-set. The results were 0.861 and 0.806, respectively. We did not observe additional accuracy degradation when running the HE-friendly AlexNet using our framework over encrypted data. We emphasize that the above accuracy-drop results from using HE-friendly NN and not from using our framework. We expect that future AI improvements will close this gap by offering improved HE-friendly NNs.

Table 4 reports the time and memory consumption for the latter experiment using 4 configurations on a set of 10*b* representative samples. The configurations involve unencrypted model weights (*PT*) and encrypted model weights (*CT*) optimized for low latency (*Latency*) or high throughput (*TP*). For these configurations, we also compared the inference results with the inference results of running HE-Friendly AlexNet on PyTorch over the plaintext test-set by calculating the Root Mean Square Error (RMSE). These were always less than 4e−3.

8 COMPARISON WITH STATE-OF-THE-ART

8.1 Matrix Multiplication

Table 5 compares our tiles-tensor-based matrix-multiplication technique for multiplying two $d \times d$ matrices with the state-of-the-art. We keep the matrices in tile tensors of shapes $[\frac{d}{\sqrt{d}}, \frac{d}{d}, \frac{*}{\sqrt{d}}]$ and $[\frac{*}{\sqrt{d}}, \frac{d}{d}, \frac{d}{\sqrt{d}}]$. Similar to [25], we assume the naïve technique uses the naïve $O(d^3)$ algorithm over ciphertexts that hold one value each. Some previous works bound the dimension d by some function of s . In [25], the authors assume $d < s$, and in [28], the authors assume that $d < \sqrt{s}$. The latter bound was also used in [12] for a version addressing a malicious adversary, and in [11] for a version addressing multi-key encryption. In both cases, s can be made arbitrarily large in two ways. We explain and compare these works to our technique, which does not place any bound on d .

Increasing the scheme’s parameters. By increasing the parameters of the scheme, s can be made arbitrarily large. However, this increases the relative overhead of every HE operation, i.e., increasing s by a factor α increases the time of every HE operation by a factor $\alpha' > \alpha$. Moreover, in practice, there is a maximal setting that every HE library supports. Our technique is superior because it enables us to use the smallest s allowed by the security requirement for the multiplication.

Arbitrarily simulating many slots. An array c of $d^2 > s$ can be simulated naïvely using $t = \lceil \frac{d^2}{s} \rceil$ ciphertexts. In this simulation, each operation performed on c (especially rotations) is translated to $O(t)$ operations on ciphertexts. This makes the complexity deteriorate to $O(\frac{d^3}{s})$. With our technique, we set the shape of the matrices to $[\frac{d}{s^{1/4}}, \frac{d}{s^{2/4}}, \frac{*}{s^{1/4}}]$ and $[\frac{*}{s^{1/4}}, \frac{d}{s^{2/4}}, \frac{d}{s^{1/4}}]$. This yields a complexity of $O(\frac{d^3}{s})$, which is similar to that of [28]. But again, our technique has a lower multiplication depth because it does not involve masking.

When considering complete NN, the authors of [28] argue that it is possible to convert fully-connected and convolutional layers to

¹PyTorch library 1.5.1 <https://pytorch.org>

Table 5: A comparison of techniques to multiply two $d \times d$ matrices using ciphertexts with s slots. The columns represent the multiplication complexity, and the multiplicative-depth of each technique. We assume the shape of the matrices in our technique are $[\frac{d}{\sqrt{d}}, \frac{d}{d}, \frac{*}{\sqrt{d}}]$ and $[\frac{*}{\sqrt{d}}, \frac{d}{d}, \frac{d}{\sqrt{d}}]$.

Method	Complexity	Mul-Depth
Naïve	$O(d^3)$	1
[14, 25]*	$O(d^2 \lceil \frac{d}{s} \rceil)$	1
[28]*	$O(d \lceil \frac{d^2}{s} \rceil)$	3
Ours	$O(d \lceil \frac{d^2}{s} \rceil)$	1

* [14, 25] assumed $d < s$ and [28] assumed $d^2 < s$. In both cases, supporting arbitrarily large d can be done by increasing s or by simulating a large slot number. The first option increases the time to compute every HE operation. Moreover, current implementations of HE-library support only limited values of s . The latter option replaces each operation by $O(t)$ operations on ciphertext, where $t = \lceil \frac{d}{s} \rceil$ for [14, 25] and $t = \lceil \frac{d^2}{s} \rceil$ for [28].

matrix-matrix multiplications. In that sense, our method, as shown above, are better.

8.2 Convolution

The state-of-the-art in implementing convolutional layers includes [29, 30, 44]. We compare these to our implementation by distinguishing between two cases: simple single-input-single-output (SISO) and multiple-input-multiple-output (MIMO).

8.2.1 SISO. The simple SISO case involves one image, one input channel, and one filter, $b = c = f = 1$. Here, all previous approaches pack the image into a single ciphertext and compute the convolution using $w_F h_F - 1$ rotations (excluding the 0 rotation) and $w_F h_F$ multiplications. However, the assumption that the image fits in a single ciphertext limits the maximal allowed image size. As mentioned above, all HE libraries limit the size of their ciphertexts. For example, SEAL [1] limits the ciphertext slots to 16,384, which is not large enough to contain 224×224 images such as those required for AlexNet. While it is possible to simulate a large ciphertext (see Section 8.1), this increases the multiplication depth by at least 1 per convolutional layer. For deep networks such as AlexNet, which have 5 convolutional layers, the extra cost of 5 multiplication levels makes the computation impossible when using libraries such as SEAL, where the maximal depth is bounded and bootstrapping is not supported. Our approach captures the method above as a special case where $w_I \leq t_1$ and $h_I \leq t_2$, i.e., the image fits inside a single tile. Our generalization offers two advantages.

First, it allows the network to efficiently handle images larger than the maximal ciphertext size; this was not described in previous works. As mentioned above, naïvely simulating larger ciphertexts using n smaller ones requires $O(n)$ rotations and increases the multiplication depth. In contrast, in our method, if we divide the image into t square tiles, it multiplies the number of rotations by $O(\sqrt{n})$ instead of $O(n)$ and no extra multiplication levels are consumed. Lemma 1 shows that the number of rotations is $O(w_F \lceil \frac{w_I}{t_1} \rceil + h_F \lceil \frac{h_I}{t_2} \rceil + w_F h_F)$ rotations.

Second, our method makes it beneficial to split even medium size images into smaller tiles. This improves performance on three

accounts: 1) smaller ciphertexts are more efficient per-slot, i.e., if an operation on a ciphertext of s slots costs t time, then the same operation on a ciphertext of $2s$ slots costs more than $2t$ time; 2) as mentioned above, the number of rotations increases sub-linearly; 3) many small operations are easier to parallelize than a few large operations.

Table 6: Comparing time and number of operations for a SISO ($b = c = f = 1$) convolution benchmark of image of size 128×128 , filter size 3×3 , and different tile sizes. Each row has a different setting for t_1, t_2 , and $t_3 = t_4 = t_5 = 1$ (see Section 5). The first row captures as a special case existing SISO methods [29, 30, 44]. Subsequent rows demonstrate the performance advantage of our generalized method.

t_1	t_2	Tiles	Time (sec)	Rotations	Multiplications
128	128	1	0.119	8	9
64	128	2	0.064	10	18
64	64	4	0.030	12	36
32	64	8	0.019	16	72

Table 6 demonstrates this advantage. It shows a SISO convolution computed with different ciphertext sizes, starting with 16,384 slots shaped as a 128×128 tile. The computations go down to the smallest ciphertext of 2,048 slots, which still supports secure computation. The table shows the speedup (more than 6x), the sub-linear growth in the number of rotations, and the linear growth in the number of multiplications. If more computation depth is required, the ciphertexts cannot be made smaller due to security constraints. In this case, we can reduce t_1 and t_2 , while increasing t_5 , the batch dimension. Table 6 will stay the same, except the rows will support batch sizes of 1, 2, 4, 8 respectively, with amortized time per sample of 0.119, 0.068, 0.061, 0.016 seconds.

8.2.2 MIMO. In MIMO, the number of channels c and filters f is larger than 1.

As a special case, the tile tensor and its accompanying notation captures the packing options described in [29, 30, 44], which is $[\frac{w_I}{t_1}, \frac{h_I}{t_2}, \frac{c}{t_3}]$. Previous approaches restricted $w_I \leq t_1, h_I \leq t_2$, but c and t_3 can be set arbitrarily. If $c \leq t_3$ then all channels are packed into a single ciphertext. If $t_3 = 1$, then each channel is in a separate ciphertext.

In HEAR [30] an additional packing optimization is described. They note that after a mean pooling layer with strides, the output comes out with strides δ , i.e., only the pixels at x and y coordinates such that $x \bmod \delta = 0$ and $y \bmod \delta = 0$ are populated with actual pixels. They use the vacant pixels to pack more channels. Using tile tensor shapes this is equivalent to $[\frac{w_I}{w_I}, \frac{\delta}{\delta}, \frac{h_I}{h_I}, \frac{\delta}{\delta}, \frac{c}{t_3}]$, where w_I, h_I are the dimensions of the output image, and the vacant slots are represented by two new dimensions, such that $w_I \delta = t_1$ and $h_I \delta = t_2$. We fill them with channel data, having a total capacity of $\delta^2 c$ channels. Packing those channels together and then summing over them requires processing time. We note that the interleaved packing method described in Section 5 reduces the need for such a packing optimization. As with interleaved packing, if δ divides the number of tiles along a dimension, the stride can be implemented

by simply discarding tiles; this saves computation time and does not create vacant pixels.

Previous approaches described numerous techniques for exploiting hoisted rotations, and for optimizing the number of rotations in MIMO convolutions. Since all existing packing methods are a special case of tile tensor capabilities, these methods are applicable within our framework. Integrating them within our optimizer is reserved for future work.

8.3 Neural Network Inference

Table 7 compares our framework, experiments, and results with prior art. The first column (NI) indicates whether the framework supports non-interactive secure computations. This is a major advantage that HeLayers provides over GAZELLE [29], nGraph-HE [8], SeaLion [42], etc. In general, using the client to assist the server reduces the latency, which in turn enables the evaluation of deep networks such as ResNet and VGG. However, it misses the paradigm of delegating the entire computations to the cloud.

The second column (PL) indicates whether the reported experiments showed practical inference latency of less than two hours. We use this Boolean comparison metric instead of comparing exact latencies because different frameworks reported their measurements on different network architectures, over different datasets with images of different sizes when using different platform configurations such as number of CPUs and memory size. Furthermore, some of the schemes e.g., CHET [17] and SeaLION [42] are not freely available online nor do they provide open source code that we can evaluate, where implementing their code from scratch may result in a biased implementation. Using the PL metric allows us to distinguish HeLayers from some works, while we use other comparison metrics to show HeLayers’ superiority over the other solutions.

The following two columns indicate whether the papers reported on a mechanism or a method to evaluate and select the packing choice that mostly fits the given criteria. Specifically, it indicates whether they support tradeoffs between latency versus memory (LM) and amortized-latency versus latency (ALL). While most schemes considered only one type of packing, as far as we know only CHET suggested using different packing and selecting among them according to some criterion such as latency, energy, or memory. However, they did not present any amortized latency tradeoffs in their paper and they did not discuss how this selection mechanism works.

The other comparison metrics we considered include whether the model is encrypted before the inference evaluation, whether the security strength was at least 128 bits, and whether the image size was at least $224 \times 224 \times 3$. All of these parameters highly affect the performance of the experimental evaluation. It can be observed from the table that HeLayers is the only framework that provides a non-interactive solution over large images and an encrypted model that considered different trade-offs and reported practical results that measure less than several minutes.

When considering the different packing proposals. TenSEAL [7] uses diagonalization techniques for matrix-multiplication and im2col for convolution, assuming a single, image as input. nGraph-HE2 [8] uses SIMD packing, which is a special case of our framework when optimized for the largest possible batch size. The closest

Table 7: Scheme comparison. (NI) - Non interactive. (PL) - reported on practical latency of less than 2 hours. (LM) - Reported support in latency vs memory tradeoff. (ALL) - Reported support in amortized-latency vs. latency tradeoff. (EM) - Reported data for encrypted model. (SB) - Security bits. Larger image size involves more complex computations.

Scheme Name	NI	PL	LM	ALL	EM	SB	Evaluated Networks	Image size
GAZELLE [29]	○	●	○	○	○	128	CryptoNets	$32 \times 32 \times 3$
Jiang et al. [28]	●	●	○	○	○	128	CryptoNets	$28 \times 28 \times 1$
nGraph-HE2 [8]	○	●	○	○	○	128	CryptoNets, MobileNetV2	$224 \times 224 \times 3$
Lloret-Talavera et al. [36]	○	○*	○	○	○	128	MobileNetV2, ResNet-50	$224 \times 224 \times 3$
SeaLION [42]	○	●	○	○	○	128	DNN-30, DNN-100, CNN-16	$28 \times 28 \times 1$
TenSEAL [7]	●	●	○	○	○	128	CryptoNets	$28 \times 28 \times 1$
DOReN [38]	●	○	○	○	○	128	VGG7, ResNet20, Lola, GHE, SHE	$32 \times 32 \times 3$
REDsec [19]	●	○	○	○	○	80	BAlexNet**	$224 \times 224 \times 3$
Lee et al. [34]	●	○	○	○	○	111.6	ResNet-20	$32 \times 32 \times 3$
CHET [17]	●	●	●	○ [‡]	○	128	LeNet-5, CIFAR-SqueezeNet, Industrial	$32 \times 32 \times 3$
HEMET [37]	●	●	○	○	○	128	AlexNet, CIFAR-SqueezeNet, InceptionNet	$32 \times 32 \times 3$
HeLayers (ours)	●	●	●	●	●	128	AlexNet, CIFAR-SqueezeNet, CryptoNets	$224 \times 224 \times 3$

* The reported latency is 63 hours for a batch of 2048 images (1.84 minutes of amortized latency).

** BinaryAlexNet with binary weights and accuracy drop to 61.5%

[‡] CHET supports batching but the paper does not provide a description, analysis, or measurements.

Table 8: Comparison of the SqueezeNet-CIFAR benchmark with our tile tensor framework and CHET [17].

Framework	Latency (sec)	Amortized Latency (sec)
CHET [17] ($b = 1$)	164.7	164.7
Ours-PT ($b = 1$)	131.8	131.8
Ours-CT ($b = 1$)	143.5	143.5
Ours-PT ($b = 64$)	913.1	14.26
Ours-CT ($b = 64$)	1214.2	18.971

work to ours is CHET [17], which uses a similar approach of an abstract data structure, CipherTensor, combined with automatic optimizations. We believe CipherTensors are less flexible than tile tensors. They include a small fixed set of implemented layouts, each with its kernel of algorithms, whereas tile tensors offer a wider variety of options with a single set of generalized algorithms. Further, it was not demonstrated that CipherTensors offer an easy method to trade latency for throughput and control memory consumption, as is possible in tile tensors by controlling the batch dimension. Finally, CipherTensors require replication of the input data using rotations, whereas some of these replications can be avoided using tile tensors. It might be the case that by this time CHET has been enhanced to include other optimizations, but we only compare our solution to data that is publicly available, i.e., to [17].

CHET is the closest framework to ours, and even though it does not provide a freely available solution or an open source one, we attempted below to provide as fair a comparison as possible to the latency reported in [17]. The authors of CHET demonstrated its efficiency for large networks by using SqueezeNet-CIFAR, which is a reduced [15] SqueezeNet [26] model over CIFAR-10. It has at least 50× fewer parameters than AlexNet but it is a bit deeper (23 layers instead of 17). see Appendix A.1. We evaluated SqueezeNet-CIFAR on HeLayers and report the results in Table 8. For a fair comparison

with CHET, we reduced the number of cores to 16 while disabling hyper-threading as in CHET. In addition, the SqueezeNet-CIFAR repository [15] contains two models, we used the larger model among the two (the smaller model has 1.2× less latency than the larger model). When optimizing for latency, our framework is 1.24× faster than CHET. In addition, we also provide the results when optimizing for throughput and for running with encrypted models, which show even better amortized results (an 11.5× speedup over CHET).

9 CONCLUSIONS

We presented a framework that acts as middleware between HE schemes and the high-level tensor manipulation required in AI.

Central to this framework is the concept of the tile tensor, which can pack tensors in a multitude of ways. The operators it supports allow users to feel like they are handling ordinary tensors directly. Moreover, the operators are implemented with generic algorithms that can work with any packing arrangement chosen internally.

The optimizer complements this versatile data structure by finding the best configuration for it given the user requirements and preferences. We demonstrated how this approach can be used to improve latency for small networks, adapt to various batch sizes, and scale up to much larger networks such as AlexNet.

Our tile tensor shape notation proved very useful for both research and development. Having the notation used in debug prints and error messages, configured manually in unit tests, and printed out in the optimizer log files, helped reduce development cycles considerably. Also, in this paper we used it to concisely and accurately describe complicated computations (e.g., Figure 6). We hope the community will adopt it as a standard language for describing packing schemes.

Our framework is the first to report successful and practical inference over a large (in terms of HE) NN such as AlexNet over large images. Today, we are in the process of extending our library to

support bootstrap capabilities. Combining this and our framework should enable us to run even larger networks such as VGG-16 and GoogleNet, which until now were only reported for client-aided designs or for non-practical demonstrations.

ACKNOWLEDGMENTS

This research was conducted while the authors worked at IBM Research - Israel. Other than that, this research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

REFERENCES

- [1] 2020. Microsoft SEAL (release 3.5). <https://github.com/Microsoft/SEAL> Microsoft Research, Redmond, WA.
- [2] 2022. HEBench. <https://hebench.github.io/>
- [3] Ehud Aharoni, Allon Adir, Moran Baruch, Nir Drucker, Gilad Ezov, Ariel Farkash, Lev Greenberg, Ramy Masalha, Guy Moshkovich, Dov Murik, Hayim Shaul, and Omri Soceanu. 2020. HeLayers: A Tile Tensors Framework for Large Neural Networks on Encrypted Data. *CoRR* abs/2011.01805 (2020). [arXiv:2011.01805](https://arxiv.org/abs/2011.01805) <https://arxiv.org/abs/2011.01805>
- [4] Adi Akavia and Margarita Vald. 2021. On the Privacy of Protocols based on CPA-Secure Homomorphic Encryption. *IACR Cryptol. ePrint Arch.* 2021 (2021), 803. <https://eprint.iacr.org/2021/803>
- [5] Ahmad Al Badawi, Jin Chao, Jie Lin, Chan Fook Mun, Sim Jun Jie, Benjamin Hong Meng Tan, Xiao Nan, Aung Mi Mi Khin, and Vijay Ramaseshan Chandrasekhar. 2021. Towards the AlexNet Moment for Homomorphic Encryption: HCNN, the First Homomorphic CNN on Encrypted Data with GPUs. *IEEE Transactions on Emerging Topics in Computing* (2021). <https://doi.org/10.1109/tetc.2020.3014636>
- [6] Moran Baruch, Nir Drucker, Lev Greenberg, and Guy Moshkovich. 2022. A methodology for training homomorphic encryption friendly neural networks. In *Applied Cryptography and Network Security Workshops*. Springer International Publishing, Cham.
- [7] Ayoub Benaissa, Bilal Retiat, Bogdan Ceber, and Alaa Eddine Belfedhal. 2021. TenSEAL: A Library for Encrypted Tensor Operations Using Homomorphic Encryption. *arXiv* (2021). <https://arxiv.org/abs/2104.03152>
- [8] Fabian Boemer, Anamaria Costache, Rosario Cammarota, and Casimir Wierzynski. 2019. NGraph-HE2: A High-Throughput Framework for Neural Network Inference on Encrypted Data. In *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography (WAHC'19)*. Association for Computing Machinery, New York, NY, USA, 45–56. <https://doi.org/10.1145/3338469.3358944>
- [9] Centers for Medicare & Medicaid Services. 1996. The Health Insurance Portability and Accountability Act of 1996 (HIPAA). <https://www.hhs.gov/hipaa/>
- [10] Kumar Chellapilla, Sidd Puri, and Patrice Simard. 2006. High Performance Convolutional Neural Networks for Document Processing. In *Tenth International Workshop on Frontiers in Handwriting Recognition*, Guy Lorette (Ed.). Université de Rennes 1, Suvisoft, La Baule (France). <https://hal.inria.fr/inria-00112631>
- [11] Hao Chen, Wei Dai, Miran Kim, and Yongsoo Song. 2019. Efficient Multi-Key Homomorphic Encryption with Packed Ciphertexts with Application to Oblivious Neural Network Inference (CCS '19). Association for Computing Machinery, New York, NY, USA. <https://doi.org/10.1145/3319535.3363207>
- [12] Hao Chen, Miran Kim, Ilya Razenshteyn, Dragos Rotaru, Yongsoo Song, and Sameer Wagh. 2020. Maliciously Secure Matrix Multiplication with Applications to Private Deep Learning. In *Advances in Cryptology – ASIACRYPT 2020*, Shihō Moriai and Huaxiong Wang (Eds.). Springer International Publishing, Cham, 31–59.
- [13] Jung Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic Encryption for Arithmetic of Approximate Numbers. In *Proceedings of Advances in Cryptology - ASIACRYPT 2017*. Springer Cham, 409–437. https://doi.org/10.1007/978-3-319-70694-8_15
- [14] Jung Hee Cheon, Hyeonmin Choe, Donghwan Lee, and Yongha Son. 2019. Faster Linear Transformations in HElib, Revisited. *IEEE Access* 7 (2019), 50595–50604. <https://doi.org/10.1109/ACCESS.2019.2911300>
- [15] David Corvoysier. 2017. Experiment with SqueezeNets, commit:2619f730b4e91313057039feb81788c5648e3951. <https://github.com/kaizouman/tensorsandbox/tree/master/cifar10/models/squeeze>
- [16] Eric Crockett. 2020. A Low-Depth Homomorphic Circuit for Logistic Regression Model Training. *Cryptology ePrint Archive*, Report 2020/1483. <https://eprint.iacr.org/2020/1483>.
- [17] 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* (Phoenix, AZ, USA) (PLDI 2019). New York, NY, USA, 142–156. <https://doi.org/10.1145/3314221.3314628>
- [18] EU General Data Protection Regulation. 2016. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation). *Official Journal of the European Union* 119 (2016). <http://data.europa.eu/eli/reg/2016/679/oj>
- [19] Lars Folkerts, Charles Gouert, and Nektarios Georgios Tsoutsos. 2021. *REDsec: Running Encrypted DNNs in Seconds*. Technical Report Report 2021/1100. 1–35 pages. <https://eprint.iacr.org/2021/1100>
- [20] Gartner. 2021. *Gartner Identifies Top Security and Risk Management Trends for 2021*. Technical Report. <https://www.gartner.com/en/newsroom/press-releases/2021-03-23-gartner-identifies-top-security-and-risk-management-t>
- [21] 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. <http://proceedings.mlr.press/v48/gilad-bachrach16.pdf>
- [22] Hayden Gunraj, Ali Sabri, David Koff, and Alexander Wong. 2021. COVID-Net CT-2: Enhanced Deep Neural Networks for Detection of COVID-19 from Chest CT Images Through Bigger, More Diverse Learning. *arXiv preprint arXiv:2101.07433* (2021). <https://arxiv.org/abs/2101.07433>
- [23] Shai Halevi. 2017. Homomorphic Encryption. In *Tutorials on the Foundations of Cryptography: Dedicated to Oded Goldreich*, Yehuda Lindell (Ed.). Springer International Publishing, Cham, 219–276. https://doi.org/10.1007/978-3-319-57048-8_5
- [24] Shai Halevi and Victor Shoup. 2014. Algorithms in HElib. In *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 8616)*, Juan A. Garay and Rosario Gennaro (Eds.). Springer, 554–571. https://doi.org/10.1007/978-3-662-44371-2_31
- [25] Shai Halevi and Victor Shoup. 2018. Faster homomorphic linear transformations in HElib. In *Annual International Cryptology Conference*. Springer, 93–120. https://doi.org/10.1007/978-3-319-96884-1_4
- [26] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. (2016). <https://arxiv.org/abs/1602.07360>
- [27] Alberto Ibarondo and Melek Önen. 2018. Fhe-compatible batch normalization for privacy preserving deep learning. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*. Springer, 389–404. https://doi.org/10.1007/978-3-030-00305-0_27
- [28] Xiaoqian Jiang, Miran Kim, Kristin Lauter, and Yongsoo Song. 2018. Secure Outsourced Matrix Computation and Application to Neural Networks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) (CCS '18). New York, NY, USA, 1209–1222. <https://doi.org/10.1145/3243734.3243837>
- [29] 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)*. USENIX Association, Baltimore, MD, 1651–1669. <https://www.usenix.org/conference/usenixsecurity18/presentation/juvekar>
- [30] Miran Kim, Xiaoqian Jiang, Kristin E. Lauter, Elkhan Ismayilzoda, and Shayan Shams. 2021. HEAR: Human Action Recognition via Neural Networks on Homomorphically Encrypted Data. *CoRR* abs/2104.09164 (2021). [arXiv:2104.09164](https://arxiv.org/abs/2104.09164) <https://arxiv.org/abs/2104.09164>
- [31] Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. *Neural Information Processing Systems* 25 (01 2012). <https://doi.org/10.1145/3065386>
- [32] Yann Lecun, Leon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324. <https://doi.org/10.1109/5.726791>
- [33] Yann LeCun, Corinna Cortes, and Christopher JC Burges. 1998. The MNIST database of handwritten digits. 10 (1998), 34. <http://yann.lecun.com/exdb/mnist/>
- [34] Joon-Woo Lee, Hyungchul Kang, Yongwoo Lee, Woosuk Choi, Jieun Eom, Maxim Deryabin, Eunsang Lee, Junghyun Lee, Donghoon Yoo, Young-Sik Kim, and Jong-Seon No. 2022. Privacy-Preserving Machine Learning With Fully Homomorphic Encryption for Deep Neural Network. *IEEE Access* 10 (2022), 30039–30054. <https://doi.org/10.1109/ACCESS.2022.3159694>
- [35] Ryan Lehmkuhl, Pratyush Mishra, Akshayaram Srinivasan, and Raluca Ada Popa. 2021. Muse: Secure Inference Resilient to Malicious Clients. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2201–2218. <https://www.usenix.org/conference/usenixsecurity21/presentation/lehmkuhl>
- [36] Guillermo Lloret-Talavera, Marc Jorda, Harald Servat, Fabian Boemer, Chetan Chauhan, Shigeki Tomishima, Nilesh N. Shah, and Antonio J. Pena. 2022. Enabling Homomorphically Encrypted Inference for Large DNN Models. *IEEE Trans. Comput.* 71, 5 (2022), 1145–1155. <https://doi.org/10.1109/TC.2021.3076123>

- [37] Qian Lou and Lei Jiang. 2021. HEMET: A Homomorphic-Encryption-Friendly Privacy-Preserving Mobile Neural Network Architecture. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.), PMLR, 7102–7110. <https://proceedings.mlr.press/v139/lou21a.html>
- [38] Souhail Meftah, Benjamin Hong Meng Tan, Chan Fook Mun, Khin Mi Mi Aung, Bharadwaj Veeravalli, and Vijay Chandrasekhar. 2021. DOREN: Toward Efficient Deep Convolutional Neural Networks with Fully Homomorphic Encryption. *IEEE Transactions on Information Forensics and Security* 16 (2021), 3740–3752. <https://doi.org/10.1109/TIFS.2021.3090959>
- [39] N/A. 2021. Removed for the purpose of the anonymous review.
- [40] N/A. 2021. Removed for the purpose of the anonymous review. <https://github.com/IBM/fhe-toolkit-linux/tree/42af59512bbf9fa58602a1d54d3e2e28b37dfb01/DEPENDENCIES/ML-HElib/src/helayers/hebase>
- [41] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang Chieh Chen. 2018. MobileNetV2: Inverted Residuals and Linear Bottlenecks. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition* (2018), 4510–4520. <https://doi.org/10.1109/CVPR.2018.00474>
- [42] 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). <https://arxiv.org/abs/1904.12840>
- [43] Alexander Viand, Patrick Jatke, and Anwar Hithnawi. 2021. SoK: Fully Homomorphic Encryption Compilers. *arXiv preprint arXiv:2101.07078* (2021), 1–17. [arXiv:2101.07078](https://arxiv.org/abs/2101.07078) <https://arxiv.org/abs/2101.07078>
- [44] Qiao Zhang, Chunsheng Xin, and Hongyi Wu. 2021. GALA: Greedy Computation for Linear Algebra in Privacy-Preserved Neural Networks. *arXiv preprint arXiv:2105.01827* (2021). <https://arxiv.org/abs/2105.01827>

A EXPERIMENTS SETUP

For the experiments we used an Intel(R) Xeon(R) CPU E5-2699 v4 @ 2.20GHz machine with 44 cores (88 threads) and 750GB memory. Unless specified otherwise, we used only 40 threads and avoided hyper-threading by instructing the OpenMP library to pin one software thread per core. We use the CKKS SEAL [1] implementation targeting 128 bits security, and all the reported results are the average of 10 runs.

A.1 Neural Networks

Table 9 summarizes the network architectures that we evaluated in this paper. It includes CryptoNets [21], over images of size 28×28 , padded to 29×29 , An HE-Friendly variants [6] of AlexNet [31] and SqueezeNet-CIFAR [15], which we train using the methods of [6].

A.1.1 CryptoNets. We implemented the network using tile tensors of shape $\begin{bmatrix} n_1 & n_2 & b \\ t_1 & t_2 & t_3 \end{bmatrix}$, where b is the batch size. In practice, we only report the results for the case $t_3 = b$ that minimizes the overall latency by filling all the ciphertext slots (8,192 in this case). For the convolution layer, we use the naïve SIMD method (Section 5.4) when b equals the number of plaintext slots and $t_1 = t_2 = 1$. Otherwise, we use our variant of the $im2col$ operator (Section 2.4). These methods work better than our novel convolution operator when the images are relatively small (e.g., MNIST images) and the network has one convolutional layer.

Figure 6 shows the tile tensor flow in our implementation. Here, the inputs I and F are the image and filter matrices, respectively, and $I', F' = im2col(I, F)$. In addition, B_c is the trained bias of the convolution layer and W_1, W_2, B_1, B_2 are the trained weights and bias info of the Fully Connected (FC) layers.

CryptoNets HE configurations. We set the plaintext poly-degree to 16,384, and set the modulus chain $\{45, 35, 35, 35, 35, 35, 45\}$ when either $t_1 = 1$ or $t_2 = 1$. Otherwise, we set the modulus chain to $\{45, 35, 35, 35, 35, 35, 35, 45\}$, for increasing the multiplication depth by one, needed for the *clean* operator (see Section 4.6).

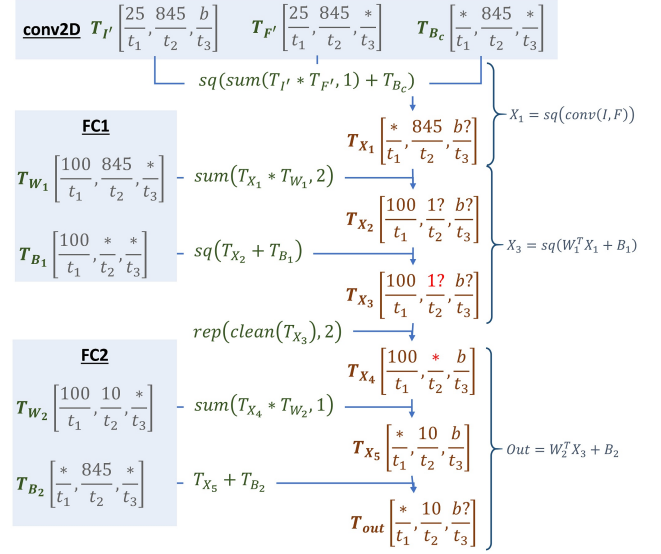


Figure 6: An illustration of our CryptoNets implementation using tile tensors. For simplicity, we only indicate the component-wise square activation layers by the $sq()$ function because they maintain the tile tensor shape. The equations on the right represent the underlying tensor operations. The input tensors are $I, F, B_c, W_1, B_1, W_2, B_2$, where $I', F' = im2col(I, F)$.

A.1.2 AlexNet. Dataset. The COVIDx CT-2A Data-set is an open access benchmark of CT images dataset designed by [22], that contains three classes of chest CT images: *Normal*, *Pneumonia* or *COVID-19* cases. For the experiment we took a subset of 10K images per class for training, 1K images per class for validation, and 201 images in total for test with 67 random samples from each class. We chose a small subset to speed up accuracy measurements. The images were resized to $224 \times 224 \times 3$ to fit the input size expected by AlexNet.

Preparing a model for inference over encrypted data. The batch normalization that we used for training requires division, which is not a CKKS primitive. Therefore, for the model inference we used a technique similar to [27] to “absorb” batch normalization layers into neighboring layers. This was done by modifying the neighbor layer’s parameters in such a way that the resulting transformation of the layer is equivalent to a sequential application of batch normalization and the original layer. The resulting network is computationally equivalent, but does not include batch normalization layers.

HE configurations We set the plaintext poly-degree to 32,768, and use the modulus chain $\{49, 39, 39, \dots, 39, 49\}$ of size 22.

A.1.3 SqueezeNet-CIFAR. For SqueezeNet-CIFAR [15] we used the same techniques that we used for AlexNet. Here, we set the plaintext poly-degree to 32,768, and use the modulus chain $\{50, 30, \dots, 30, 50\}$ of size 28.

Table 9: Network architecture used in our experiments. C, A, P, D, and F denote convolution, activation, pooling, dense, and fire module layers, respectively. A fire module has the form C-A-C2-A, where C2 represents two concated convolutions.

Name	Architecture
CryptoNets [21]	C-D-D
SqueezeNet-CIFAR [15] (HE-friendly [6])	C-A-P-F-F-P-F-F-C-A-P
AlexNet [31] (HE-friendly [6])	C-A-P-C-A-P-C-A-C-A-C-A-P-D-A-D-A-D

B TILE TENSORS DEFINITION

DEFINITION B.1 (EXTERNAL TENSOR). A k -dimensional external tensor E is a k -dimensional tensor that each of its elements is itself a k -dimensional tensor, all having an identical shape. These internal tensors are referred to as tiles, their shape as the tile shape, and the shape of the external tensor as the external shape. A slot in E is identified by $E(a_1, \dots, a_k)(b_1, \dots, b_k)$ where a_i are the external indices of a tile, and b_i are the internal indices inside the tile.

DEFINITION B.2 (TILE TENSOR SHAPE). A k -dimensional tile tensor shape is comprised of an external shape $[e_1, \dots, e_k]$, tile shape $[t_1, \dots, t_k]$, original shape $[n_1, \dots, n_k]$, replication counts $[r_1, \dots, r_k]$, interleaved Boolean indicator $[l_1, \dots, l_k]$, and unknown Boolean indicators $[u_1, \dots, u_k]$. It is required that $\forall_i(r_i = 1 \vee n_i = 1) \wedge (\max(r_i, n_i) \leq e_i t_i)$.

DEFINITION B.3 (EXTERNAL TENSOR LOGICAL INDICES). Given a tile tensor shape S , and an external tensor E , and a specific slot in E specified by external indices (a_1, \dots, a_k) , and internal indices (b_1, \dots, b_k) , then this slot is associated with the logical indices (c_1, \dots, c_k) with respect to S , computed as follows: For $i = 1, \dots, k$, if the interleaved indicator l_i is true, then $c_i = b_i e_i + a_i$ else $c_i = a_i t_i + b_i$.

DEFINITION B.4 (VALIDITY RELATION, PACKED TENSOR). A tile tensor shape S is valid for an external tensor E if their external shapes and tile shapes match, and there exists a tensor $T[n_1, \dots, n_k]$ such that for $T_1 = \text{broadcast}(T, [n_1 r_1, \dots, n_k r_k])$ it holds that $E(a_1, \dots, a_k)(b_1, \dots, b_k) = T_1(c_1, \dots, c_k)$ for all slots with internal, external, and logical indices a_i, b_i, c_i , such that $\forall_i c_i \leq n_i r_i$. For all other slots of E , if $\forall_i((c_i \geq r_i n_i) \rightarrow \neg u_i)$ then these slots are set to zero. T is the packed tensor.

DEFINITION B.5 (TILE TENSOR). Tile tensor is a pair (E, S) where E is an external tensor and S a tile tensor shape that is valid for it.

DEFINITION B.6 (UNPACK OPERATOR). Given a tile tensor $T_A = (E, S)$ the operator $\text{unpack}(E)$ results with the packed tensor of T_A .

DEFINITION B.7 (PACK OPERATOR). Given a tensor A and a tile tensor shape S whose original shape matches the shape of A , then the pack operator $\text{pack}(A, S)$ results with a tile tensor $T_A = (E, S)$ such that A is the packed tensor of T_A .

B.1 Tile Tensor Shape Notation

A tile tensor shape can be specified with a special notation involving a list of symbols. Each element in the list specifies the details of one dimension. $\frac{n_i}{t_i}$ specifies the original and tile shape along this dimension, and $r_i = 1, e_i = \left\lceil \frac{n_i}{t_i} \right\rceil, l_i = u_i = \text{false}$. $\frac{r_i}{t_i}$ further specifies the replication count and $n_i = 1$, and $\frac{*}{t_i}$ specifies $n_i = 1, r_i = t_i$. $\frac{n_i \sim e_i}{t_i}$ specifies $l_i = \text{true}$, and $\frac{n_i \sim e_i}{t_i}$ specifies a value for

e_i other than the default mentioned above. For any of the above mentioned options a "?" symbol above the line indicates $u_i = \text{true}$.

B.2 Operators

DEFINITION B.8 (TILE TENSOR SHAPE COMPATIBILITY). Tile tensor shapes S and S' are compatible for all i , $t_i = t'_i$, $(n_i = n'_i \wedge e_i = e'_i \wedge l_i = l'_i) \vee (n_i = 1 \wedge r_i = t_i) \vee (n'_i = 1 \wedge r'_i = t'_i)$.

DEFINITION B.9 (TILE TENSOR ADDITION). Let $T = (E, S)$ and $T' = (E', S')$ be tile tensors with compatible shapes, then $T + T' = T'' = (E'', S'')$ such that $E'' = E' + E$, $n''_i = \max(n_i, n'_i)$, $r''_i = \min(r_i, r'_i)$, $u''_i = (e_i t_i - n_i r_i \neq e'_i t'_i - n'_i r'_i) \vee u_i \vee u'_i$, $l''_i = l_i \vee l'_i$.

DEFINITION B.10 (TILE TENSOR ELEMENTWISE MULTIPLICATION). Let $T = (E, S)$ and $T' = (E', S')$ be tile tensors with compatible shapes, then $T * T' = T'' = (E'', S'')$ such that $E'' = E' * E$, $n''_i = \max(n_i, n'_i)$, $r''_i = \min(r_i, r'_i)$, $u''_i = ((e_i t_i - n_i r_i = e'_i t'_i - n'_i r'_i) \wedge u_i \wedge u'_i) \vee ((e_i t_i - n_i r_i < e'_i t'_i - n'_i r'_i) \wedge u'_i) \vee ((e_i t_i - n_i r_i > e'_i t'_i - n'_i r'_i) \wedge u_i) \vee l''_i = l_i \vee l'_i$.

DEFINITION B.11 (TILE TENSOR SUMMATION). Let $T = (E, S)$ be a tile tensor such that for a given index i it holds that $u_i = \text{false}$, $r_i = 1$. Then $T' = \text{sum}(T, i)$ is a tile tensor $T' = (E', S')$ computed as follows. Let $E_1 = \text{sum}(E, i)$. E' is computed from E_1 by summing over the dimension i of every tile L of E_1 using the rotate-and-sum algorithms [3]. S' is identical to S except $n'_i = 1$, and if $\forall_j c_j < t_j = 1$ and t_i is a power of 2, then $r'_i = t_i$, else $u'_i = (t_i > 1)$.

Remark 3. The output tile tensor shape of tile tensor summation is due to the behaviour of rotate-and-sum algorithms as explained in [3]. In environments where summing inside a tile can be performed differently, the shape might be different. Specifically, Some HE systems support native multi-dimensional structure to the ciphertexts, allowing rotating a tile along one of its dimensions directly. This allows having replicated output for any dimension.

Remark 4. The constraint $u_i = \text{false}$ in Definition B.11 can be removed with some straightforward modifications. These details are omitted.

DEFINITION B.12 (TILE TENSOR REPLICATION). Let $T = (E, S)$ be a tile tensor and i be an index such that $n_i = 1, r_i = 1$, and $\forall_j u_j = \text{false}$. Then $T' = \text{rep}(T, i)$ is a tile tensor $T' = (E', S')$ computed as follows. E' is computed from E by applying replication along dimension i for on every tile L of E using the rotate-and-sum algorithms of [3]. S' is identical to S except $r'_i = t_i$.

B.3 Tile Tensor Glossary

Below is a short summary of tile tensor terminology.

- Tile A tensor of some fixed shape, stored flattened inside a vector and operated on in SIMD fashion.

- *Tile tensor* A data structure containing an *external tensor* as data and a *tile tensor shape* as meta data.
- *External tensor* A tensor with tiles as elements.
- *Tile shape* The shape of tiles in the external tensor.
- *Tile tensor shape* Meta data specifying the original shape, tile shape, and additional packing details.
- *Packed tensor* The tensor that will be the result of unpacking a tile tensor.
- *Original shape* The shape of the packed tensor.

C OPTIMIZER ALGORITHMS

Algorithm 1 The optimizer operation

Input: m - a JSON description of a machine learning model.
 Req - a key-value dictionary of user requirements.
 O - an ordered list of optimization targets e.g., 'latency' and 'memory'.
 $Constraints$ - a key-value dictionary of constraints.

Output: p - an optimal packing configuration for m .

```

1: procedure OPTIMIZER( $m, Req, O, Cons$ )
2:    $p = null$ 
3:    $pRes = null$ 
4:   for  $conf \in \text{GenerateConfs}()$  do
5:      $packedM = \text{Pack}(m, conf)$ 
6:      $res = \text{Simulate/Eval}(packedM, m)$ 
7:     for  $r \in Req.keys()$  do
8:       if  $res[r] > Req[r]$  then
9:         Continue with next  $conf$ 
10:    for  $r \in Constraints.keys()$  do
11:      if  $res[r] > Constraints[r]$  then
12:        Continue with next  $conf$ 
13:    for  $o \in O$  do
14:      if  $p! = null$  and  $res[o] \geq pRes[o]$  then
15:        Continue with next  $conf$ 
16:     $(p, pRes) = (conf, res)$ 
17:  return  $p$ 

```

Algorithm 2 GenerateConf

Input: m - a JSON description of a machine learning model.
Output: l - A list of possible configurations.

```

1: procedure GENERATECONF( $m$ )
2:    $l = null$ 
3:    $n = \text{GetMinimalMulDepth}(m)$ 
4:    $s = \text{GetTileSize}(n)$ 
5:    $d = \text{GetNumberOfTileTensorDims}(m)$ 
6:   ▷ e.g., depends on the existence of a convolutional layer in  $m$ .
7:    $c = \text{GetSupportedConvModes}(m)$ 
8:    $ts = \text{GetTileTensorShapes}(d, s)$  where
9:     ▷ the product of the shape denominators equals  $s$ 
10:    ▷ the denominators are powers of two
11:  return  $l$ , a list of all element combinations from  $ts$  and  $c$ 

```

Table 10: Simulated time estimations for the configurations $(Conf_i)_{i=1..4}$ formatted as [tile shape - convolution mode (Section 5.3)]: [16, 8, 8, 16, 1]-CWHFB, [8, 8, 8, 32, 1]-CWHFB, [16, 8, 8, 16, 1]-FWHCB, [32, 8, 8, 8, 1]-FWHCB, respectively. The acronyms CWHFB and FWHCB indicate the order of dimensions in the tile tensor. The deviation of the estimated times from the real times are reported in brackets.

Config	Inference (sec)	Model enc. (sec)	Input enc. (sec)
$Conf_1$	4,232 (-11%)	1,509 (-11.5%)	162 (-6.8%)
$Conf_2$	4,758 (-13.9%)	1,493 (-12.1%)	164 (-7.9%)
$Conf_3$	4,927 (-18.1%)	1,680 (-11.5%)	177 (-6.8%)
$Conf_4$	4,798 (-20%)	1,668 (-12.3%)	178 (-7.3%)

D OPTIMIZER ACCURACY

The optimizer's simulator (Section 6) estimates the time and memory usage for a given configuration option on a single CPU thread. For that, it relies on pre-benchmarked measures of the different HE operations. To assess the accuracy of these estimations, we performed the following experiment on HE-friendly AlexNet using encrypted model. We chose the four configuration options that achieved the lowest estimated latency when using local search (Section 6) and compared the inference time and the encryption time of the input and the model between the simulation output and an actual run over encrypted data.

Table 10 summarizes the results. Our empirical tests show that the simulator provides relatively accurate time estimations for all four configurations. The average estimated time deviation is -15.8%, -11.9%, and -7.2% for inference, model encryption, and batch input encryption, respectively. We note that the simulated storage matches the measured storage for all configurations, thus we do not include this data in Table 10.

E A PROOF OF LEMMA 1

Lemma 1. *Let s be the number of slots in a ciphertext. Then, given an input image $I[w_I, h_I]$ and a filter $F[w_F, h_F]$, packing I as $T_I[\frac{w_I}{t_1}, \frac{h_I}{t_2}]$ and the filter as $T_F[w_F, h_F, \frac{*}{t_1}, \frac{*}{t_2}]$, convolution can be computed using: $O(\lceil w_I h_I w_F h_F / s \rceil)$ multiplications, and $O(w_F \lceil \frac{w_I}{t_1} \rceil + h_F \lceil \frac{h_I}{t_2} \rceil + w_F h_F)$ rotations. The input is encoded in $O(w_I h_I / s)$ ciphertexts.*

PROOF. Multiplications. To compute the convolution, we need to multiply each of the $w_I h_I$ elements of the input tensor with each of the $w_F h_F$ elements of the filter, excluding edge cases that do not change the asymptotic behavior. Since each multiplication multiplies s slots, we need only $O(\lceil w_I h_I w_F h_F / s \rceil)$ multiplications.

Rotations. Recall the output is of size $(w_I - w_F + 1)(h_I - h_F + 1)$. We map the k -th slot of different ciphertexts to elements of I with indexes $k \lceil \frac{w_I}{t_1} \rceil \leq x_o < (k + 1) \lceil \frac{w_I}{t_1} \rceil$ and $k \lceil \frac{h_I}{t_2} \rceil \leq y_o < (k + 1) \lceil \frac{h_I}{t_2} \rceil$. It is therefore enough to analyze the cost of computing the convolution for $0 \leq x_o < \lceil \frac{w_o}{t_1} \rceil$ and $0 \leq y_o < \lceil \frac{h_o}{t_2} \rceil$, since computing the other elements of the output has no cost due to the

SIMD feature. It follows that a rotation is needed when $x_o + i \geq \left\lceil \frac{w_I}{t_1} \right\rceil$ or $y_o + j \geq \left\lceil \frac{h_I}{t_2} \right\rceil$. This totals to $O(w_F \left\lceil \frac{w_I}{t_1} \right\rceil + h_F \left\lceil \frac{h_I}{t_2} \right\rceil + w_F h_F)$.

Storage. Since we use $O(s)$ slots of each ciphertext, the input can be encoded in $O(w_I h_I / s)$ ciphertexts. \square

F CONVOLUTION OPERATION COUNTS

It is easy to deduce the number of tiles from a tile tensor shape. The image tile tensor $T_I \left[\frac{w_I}{t_1}, \frac{h_I}{t_2}, \frac{c}{t_3}, \frac{b}{t_4}, \frac{*}{t_5} \right]$ has

$$\left\lceil \frac{w_I}{t_1} \right\rceil \left\lceil \frac{h_I}{t_2} \right\rceil \left\lceil \frac{c}{t_3} \right\rceil \left\lceil \frac{b}{t_4} \right\rceil$$

tiles, and the filters tile tensor $T_F^l [w_F, h_F, \frac{*}{t_1}, \frac{*}{t_2}, \frac{c}{t_3}, \frac{*}{t_4}, \frac{f}{t_5}]$ has

$$w_F h_F \left\lceil \frac{c}{t_3} \right\rceil \left\lceil \frac{f}{t_5} \right\rceil$$

tiles. We multiply every image tile with every filter tile that covers the same range of channels. This results with

$$\left\lceil \frac{w_I}{t_1} \right\rceil \left\lceil \frac{h_I}{t_2} \right\rceil \left\lceil \frac{c}{t_3} \right\rceil \left\lceil \frac{b}{t_4} \right\rceil w_F h_F \left\lceil \frac{f}{t_5} \right\rceil$$

multiplications.

In contrast, it is more difficult to compute the number of rotations. We can compute it exactly for the general case, using the following formulas. The number of image tiles that we rotate due to the filter exceeding to the right, Equation 7; due to exceeding from below, Equation 8; and when both are exceeded, Equation 9. Additional rotations are required for summing over the channels dimension using rotate and sum, Equation 10, and every second convolution layer the same amount again is needed for duplicating along this dimension.

$$(w_F - 1) \left\lceil \frac{h_I}{t_2} \right\rceil \left\lceil \frac{c}{t_3} \right\rceil \left\lceil \frac{b}{t_4} \right\rceil \tag{7}$$

$$(h_F - 1) \left\lceil \frac{w_I}{t_1} \right\rceil \left\lceil \frac{c}{t_3} \right\rceil \left\lceil \frac{b}{t_4} \right\rceil \tag{8}$$

$$(w_F - 1)(h_F - 1) \left\lceil \frac{c}{t_3} \right\rceil \left\lceil \frac{b}{t_4} \right\rceil \tag{9}$$

$$\left\lceil \frac{w_I}{t_1} \right\rceil \left\lceil \frac{h_I}{t_2} \right\rceil \log t_3 \left\lceil \frac{b}{t_4} \right\rceil \left\lceil \frac{f}{t_5} \right\rceil \tag{10}$$

The above formulas are accurate for the general case, however, they over-count for certain special cases. These include the case where the filter is very large, and hence its valid positions do not cause it to exceed with its full length to neighboring tiles. Another case where the tile sizes t_1 or t_2 are 1, and rotating them is redundant.