Memory Efficient Privacy-Preserving Machine Learning Based on Homomorphic Encryption

Robert Podschwadt[1(✉)], Parsa Ghazvinian[1], Mohammad GhasemiGol[2], and Daniel Takabi[2]

[1] Georgia State University, Atlanta, GA 30303, USA
{rpodschwadt1,pghazvinian1}@student.gsu.edu
[2] Old Dominion University, Norfolk, VA 23529, USA
{mghasemi,takabi}@odu.edu

Abstract. Fully Homomorphic Encryption (FHE) enables computation on encrypted data and can be used to provide privacy-preserving computation for machine learning models. However, FHE is computationally expensive and requires significant memory. Single instruction multiple data (SIMD) can offset this cost. Batch-packing, an SIMD technique that packs data along the batch dimension, requires significant memory. In convolutional neural networks, we can exploit their regular and repeating structure to reduce the memory cost by caching recurring values. In this paper, we investigate strategies for dynamically loading data from persistent storage and how to cache it effectively. We propose a method that reorders operations inside the convectional layer to increase caching effectiveness and reduce memory requirements. We achieve up to 50x reduction in memory requirements with only a 13% increase in runtime compared to keeping the data in memory during the entire computation. Our method is up to 38% faster at no significant memory difference compared to not using caching. We also show that our approach is up to 4.5x faster than the operating system's swapping technique. These improvements allow us to run the models on less powerful and cheaper hardware.

Keywords: neural networks · homomorphic encryption · privacy · privacy-preserving machine learning

1 Introduction

Machine learning (ML) and neural networks specifically are widely deployed in many different scenarios, from voice assistants like Siri [27], Alexa [6], and Google Assistant [21] over writing assistants like Grammarly [22], and chatbots like Bard [20] and ChatGPT [39], to medical diagnostic systems [16,30]. Many of these systems deal with privacy-sensitive data, some of which enjoy special legal protections, e.g., medical data. These systems send the data to a server,

C. Pöpper and L. Batina (Eds.): ACNS 2024, LNCS 14584, pp. 313–339, 2024.
https://doi.org/10.1007/978-3-031-54773-7_13

which runs it through its model and returns the result to the client. Since the server needs access to the unencrypted client data to perform the computation, the client's privacy is at risk. The server might use the data to train further ML models, which could expose the data to privacy attacks, or the server itself could be breached and the data stolen. Researchers have recently proposed solutions to protect user data privacy in ML applications using different methods. Differential Privacy [18] solutions preserve the privacy of the training data in the trained model [1,40]. To protect the data during inference, solutions commonly use Secure Multiparty Computation (SMC) [12,14,36,37], Fully Homomorphic Encryption (FHE) [31,33,34] or a mixture between the two [7,38]. SMC allows multiple parties to jointly evaluate a function without revealing their private inputs; however, it requires all parties to stay online during the computation. FHE, on the other hand, can be used entirely offline. FHE is a type of encryption that allows computation on encrypted data without exposing any inputs, intermediate, or final results. Neural networks are a popular choice for privacy-preserving ML models since most operations, like fully connected layers or convolutions, can be performed easily using FHE. Additionally, neural networks perform very well on a wide range of tasks. However, FHE introduces significant time and memory overhead. Some FHE schemes support single instruction multiple data (SIMD) processing, which can offset some time and memory overhead. FHE ciphertexts can be thought of as fixed-sized encrypted vectors containing thousands of elements, called slots. Two approaches for filling the slots have been used for ML. 1.) Pack all the features of an instance into as few ciphertexts as possible and perform convolutions and dot products with the help of rotations [2,9,34], called inter-axis packing. This has the advantage that the number of ciphertexts and total operations is relatively small, making it fast for a small number of instances. However, this approach often requires large rotation keys, and the rotations require additional time. 2.) Pack multiple instances' features into a single ciphertext [17,25,42], called batch-packing. This produces as many ciphertexts as the data has features. Batch-packing allows us to simultaneously compute results for many instances, leading to low amortized per-instance cost and high throughput. However, it suffers from high latency and memory requirements. Batch-packing is beneficial when many instances need to be processed, and low latency is not essential. For example, in a medical image diagnostic system, where images are collected throughout the day, and an ML system analyzes them overnight. This work focuses on convolutional neural networks (CNN), specifically. We address the memory requirements for convolutional layers by trading disc space for main memory. Disc space is typically orders of magnitude cheaper. However, it is also slower. We dynamically load ciphertexts and plaintexts and clear them from memory when no longer needed. We present and compare different strategies and their impact on memory and runtime. Prior work focuses primarily on latency reduction; reduction in memory is often a side effect of inter-axis packing. To the best of our knowledge, this is the first study that performs an in-depth analysis of caching strategies and memory reduction for batch-packed inference. Brutzkus et al. [9] or Lee et al. [34] propose input packing techniques, which reduce the number of ciphertexts and thereby

memory requirements. However, these approaches require additional operations like masking and rotation, which lower the overall throughput. Boemer et al. [7] present a complex encoding, allowing them to fit more values into a ciphertext. This can reduce the number of ciphertexts and plaintext when using inter-axis packing. However, for batch-packing, it only affects the batch size. Approaches that use client interaction, such as Boemer et al. [7], Podschwadt et al. [41], and Cai et al. [10] can often use smaller crypto parameters, since the client interaction resets the noise level, allowing for further computation. However, these approaches require the client to be online during the computation. We make the following main contributions:

- We propose a schedule representation for convolutions that allows us to reorder its fundamental operations to achieve increased caching performance.
- We propose a memory estimation algorithm for schedules.
- We propose an algorithm for executing a schedule using multiple threads.
- We propose multiple strategies for creating schedules, which we analyze and experimentally evaluate with regard to their time and memory requirements.

The paper is organized as follows: in Sect. 2, we discuss the theoretical background and notation. In Sect. 3, we discuss related work before we describe our proposed approach in detail in Sect. 4. Section 5 describes ways to reorder the computation to reduce memory requirements, which we experimentally evaluate in Sect. 6. We conclude the paper in Sect. 7.

2 Background

Here, we consider 2-D convolutional layers since they are commonly used in image classification, a prevalent ML task. However, our proposed approach is not limited to 2-D and can easily be transferred to convolutions in other dimensions. We consider convolutions with inputs X, weights W, and outputs Y, where X, W, and Y are tensors all four-dimensional tensors. The first dimension of X and Y is the batch dimension, and $|\cdot|$ denotes the number of elements in a tensor. Lowercase bold letters, e.g., $\mathbf{x}$, indicate elements of a tensor.

2.1 Fully Homomorphic Encryption

FHE schemes are public key crypto schemes that can evaluate addition and multiplication on encrypted data without decrypting it at any point. The result of the computation is also encrypted. After decryption, the result is as if the computation was performed on plain data. In this paper, we use the Residue Number System (RNS) version of the Cheon-Kim-Kim-Song (CKKS) scheme [13]. Unlike most other schemes, CKKS supports real numbers. However, it performs encrypted computation only approximately, leading to approximation errors. The error appears first in the least significant bits of the result, keeping the error small. We can think of CKKS plain- and ciphertexts as one-dimensional vectors containing multiple values offering vectorized, element-wise SIMD computation

[44]. The maximum number of values, typically called slots, is determined by the security parameters, which is a power of two. The number of filled slots in ciphertext does not impact the performance of the operation, allowing us to perform addition or multiplication of thousands of values at once.

2.2 Batch Packing

We consider n_s to be the number of slots in a ciphertext. For simplicity, we assume that the batch size is equal to n_s. Otherwise, we would need to split the data into multiple batches or pad it. We partially flatten all dimensions in X except the batch dimension to encrypt the inputs. We take each column from the resulting two-dimensional matrix and encrypt that into a ciphertext, leaving us with a vector of ciphertexts. We need to encode the weights as well. Each weight value in W is encoded into its own plaintext. Before encoding, we turn each value into a vector by repeating it n_s times. This produces $|X|/n_s$ ciphertexts and $|W|$ plaintexts. If the model needs to be encrypted as well, we can encrypt the encoded plaintext weights. Similarly, the encoding of W contains $|W|/n_s$ ciphertexts. We can think of the encoding as setting the batch axis to one. The issue that arises is that FHE ciphertexts and plaintexts require a substantial amount of memory. A single ciphertext can be between a few hundred kilobytes to multiple megabytes, depending on the crypto parameters; a plaintext is half the size of a ciphertext. We refer to the encoded and/or encrypted inputs, weights, and outputs as X', W', and Y', respectively, and values taken from them as $\mathbf{x}'$, $\mathbf{w}'$ and $\mathbf{y}'$.

2.3 Convolutional Layers

Here, we consider two-dimensional convolutions commonly used in neural networks; however, other dimensionalities work fundamentally the same. Goodfellow et al. [19] define the operation as follows: given the inputs X,W, and the output Y, which are all tensors, we can define the two-dimensional convolution as:

$$Y_{b,m,n,c_{\text{out}}} = \sum_{j} \sum_{k} \sum_{c_{\text{in}}} X_{b,m-j,n-k,c_{\text{in}}} W_{j,k,c_{\text{in}},c_{\text{out}}} \tag{1}$$

We use the subscript to indicate a single element in the tensor, where b is the batch index, c_{in} the input channel index, and c_{out}, the output channel index. Eq 1 needs to be computed for all values in Y.

2.4 Lock-Free Multi-threaded Convolution

The most straightforward way to compute a convolution with multiple threads is to have each $\mathbf{y}$ computed by a thread; Eq. 1 is computed by a separated thread for each unique $(b, m, n, c_{\text{out}})$. For $s = |Y|$, we can use at most $n_t = s$ parallel threads without requiring some synchronization between the threads since all threads read from the shared resources X and W but do not modify them; every $Y_{b,m,n,c_{\text{out}}}$ is only modified by one thread, ruling out any race conditions that

could lead to lost updates. With fewer than s threads, threads can compute multiple $Y_{b,m,n,c_{\text{out}}}$. With more than s threads, we either need synchronization or can not use these threads. Generally, given n_t threads where each thread is assigned a unique integer $i \in [1, n_t]$, we can use Algorithm 1.

Algorithm 1. Lock-free multi-threaded Convolution

Inputs: input tensor X, weight tensor W, output tensor Y, number of threads n_t
Outpus: output tensor Y containing the result of the convolution

1: **while** $t \leq n_t$ **and** $t \leq |Y|$ **do**
2: **Start Thread** t and execute:
3: $q := t$
4: **while** $q \leq |Y|$ **do**
5: convert q to multi-dimensional index b, m, n, c_{out}
6: $Y_{b,m,n,c_{\text{o}}} := \sum_j \sum_k \sum_{c_{\text{i}}} X_{b,m-j,n-k,c_{\text{i}}} W_{j,k,c_{\text{i}},c_{\text{o}}}$
7: $q := q + n_t$
8: **end while**
9: **End Thread**
10: **end while**

We assume that inputs are stored on a disk (the term disk refers to any persistent storage, i.e., a hard disk drive or solid-state drive) and must be loaded into memory. With the Algorithm 1, we have two options to keep it lock-free. 1.) Load X and Y before we start the computation, or 2.) Each thread loads the $\mathbf{x}$ and $\mathbf{y}$ as needed. 1.) has the upside in that we only need to load each value once and can reuse them at no additional cost. However, the downside is that we need to keep them in memory for the entirety of the computation. 2.) On the other hand, needs to keep much fewer objects in memory. Each thread has only three objects in memory: one $\mathbf{x}$, one weight $\mathbf{w}$, and the output $\mathbf{y}$. However, each thread must perform two loads for each iteration of the nested sums in line 6. Furthermore, multiple threads may load the same $\mathbf{x}$ and $\mathbf{w}$, causing redundant loads. A further issue with this algorithm arises when $|Y|$ is not divisible by n_t. In this case, $|Y| - |Y| \mod n_t$ threads finish one iteration, line 4, early and are idle for the rest of the computation, leading to unused computational resources. However, this impact is small if $|Y|$ is large compared to n_t. Performing the second option on plain data will lead to slower results since arithmetic operations are much faster than data loading. Additionally, single $\mathbf{x}$ and $\mathbf{w}$ are so small that we can not save significant memory by loading them on demand.

Running Algorithm 1 on encrypted data is straightforward using batch-packing described earlier. To do this, we replace X with X', W with W', and Y with Y'. This replacement sets the batch dimension to one, allowing us to remove it from consideration. For the algorithm, it does not matter if W' is encoded FHE plaintexts or ciphertexts if the model is encrypted. In the case of the plaintext model, we assume that unencoded weights W are loaded into memory before the computation starts and are encoded when needed; therefore, we don't need to load them strictly speaking. However, we call this operation loading for simplicity in this context.

3 Related Work

Akavia et al. [3] focus on reducing the storage footprint of FHE ciphertext rather than the in-memory size during computation. They design a protocol that allows multiple data producers to upload and store data in the cloud with no overhead compared to storing AES (Advanced Encryption Standard) encrypted data. Storing AES encrypted on an untrusted server and using secret sharing, a computing server can use the data for HE computation with the help of an auxiliary server. In contrast, our proposed solution reduces the memory footprint at computation rather than the encrypted storage size.

Jiang et al. [28], Brutzukus et al. [9], Lee et al. [34], Dathathri et al. [15], and Lee et al. [33] are conceptually similar works, who all reduce the number of ciphertexts required by using inter-axis packing. While all these approaches reduce the inference latency, they require expensive rotations, lowering the throughput compared to batch-packed solutions. Additionally, they often rely on designing the packing strategy for the specific network architecture.

Other studies rely on interactive solutions for privacy preservation. Hao et al. [23] and Huang et al. [26] both propose efficient matrix multiplications in a two-party setting. Both studies propose rotation-free matrix multiplication over polynomial encoded ciphertexts. However, both require interactive phases where one party must extract specific polynomial coefficients and mask the result. Zheng et al. [46] propose a method for fast private inference using transformers and SMC. The authors use a similar protocol to the one proposed by Juvekar et al. [29], where the server performs much of the expensive matrix multiplication computation in an offline phase. Zheng et al. [46] reduce the number of ciphertext rotations required by packing the same feature of different tokens into the same ciphertext, similar to the batch-packing we use in our approach. However, we compute the intermediate terms in a less memory-consuming way.

Prior work on batch-packed PPML using FHE [8,11,17,24] does not explicitly state how they perform matrix multiplication or convolutions. They focus on other improvements like better polynomial approximation [11,24], or parameter fusion and special value bypass [8]. We believe most of these solutions could decrease memory requirements using our proposed algorithm. Another work that addresses memory limitations is Badawi et al. [4], which implements a CNN over FHE data using GPU acceleration for the basic ciphertext operation. To fit the input to the convolution into GPU memory, they split it into multiple blocks of the same size as the filter. The filter and as many of these blocks as possible are loaded into GPU memory, where the convolution is performed. Compared to our proposed approach, this process only reduces the memory requirement on the GPU. The input and weights still need to be present in the main memory. Shivdikar et al. [43] also present techniques aimed at GPUs. They aim to reduce the repeated memory reads inside the GPU when performing polynomial multiplication for HE primitives. While this speeds up the low-level operations underpinning most HE schemes, unlike our work, it does not address the issue of requiring a large number of plaintexts or ciphertexts in memory.

4 Our Proposed Approach

To address the issues of memory consumption and unused resources, we model the convolutional layer as a schedule, which determines the order of operations. We present and compare multiple schedule construction strategies based on the computation and available resources. We further present an algorithm to execute a schedule. From now on, we assume that all tensors are flattened.

4.1 Modeling the Problem as a Schedule

We can write each element $\mathbf{y}$ as a sum of products of $\mathbf{x}$ and $\mathbf{w}$. We denote a product of two elements of $\mathbf{x}$ and $\mathbf{w}$ as the triple $t = (\mathbf{x}, \mathbf{w}, \mathbf{y})$, where $\mathbf{y}$ is the result that holds the sum that the product $\mathbf{xw}$ is a part of. To refer to an element in a triple t, we use the following notation $t^i; i \in \{x, w, y\}$.

Algorithm 2. Generating a schedule from a 2D Convolution

Inputs: Output shape $h_{\text{out}}, w_{\text{out}}, c_{\text{out}}$, Input shape $h_{\text{in}}, w_{\text{in}} c_{\text{in}}$, Filter size w_h, w_w
Output: The schedule S

1: $S := [\,]$
2: **for each** $i \in [1, \ldots, h_{\text{out}} w_{\text{out}} c_{\text{out}}]$ **do**
3: convert i to multi-dimensional index (m, n, t^y)
4: **for each** $j \in [1, \ldots, w_h w_w c_{\text{in}}]$ **do**
5: convert j to multi-dimensional index (p, q, r)
6: $t^x := m - p + ((n - q) w_{\text{in}} + (r h_{\text{in}} w_{\text{in}})$
7: $t^w := p + (q w_h) + (r w_w w_h) + (r w_w w_h c_{\text{in}}))$
8: append (t^x, t^w, t^y) to S
9: **end for**
10: **end for**

Definition 1 (Schedule). *Let f be a convolutional layer; we say $t_i \in f$ iff the sum to compute t_i^y contains the product $t_i^x t_i^w$. A schedule is an ordered list of triples t_i that contains all $t_i \in f$ exactly once.*

In other words, we represent f as a sequence of all its element-wise products. To compute the function f, we need to compute all products given in the schedule. Additionally, we must sum all products with the same value for $\mathbf{y}$. We call the number of triples in a schedule the length or steps of a schedule, denoted by $|S|$. Algorithm 2 shows how to generate a schedule for two-dimensional convolutions. Higher dimensional convolutions work analogously by expanding the iteration bounds in lines 2 and 4, the decomposition of i and j in lines 3 and 5, and the formula for t^x and t^w by the extra dimensions. In addition to the computation steps, we also insert load instructions into the schedule. Load instructions specify which elements to load into memory, discard from memory, or write back to disk in case they were updated.

4.2 Executing a Schedule

To execute a schedule, we evaluate all triples in order. To evaluate a triple t we multiply the input X_{t^x} with the weight W_{t^w} and add the result to Y_{t^y}; $Y_{t^t} = Y_{t^y} + X_{t^x} W_{t^w}$. We assume that all $\mathbf{y}$ are 0 at the beginning. We parallelize the execution of the schedule across multiple threads. Each repeatedly evaluates the first unevaluated triple. This requires synchronization at two points. 1.) We must ensure that every triple is evaluated exactly once. 2.) Unlike in Algorithm 1, we cannot guarantee that multiple threads do not write to the same output; therefore, we need locking to prevent race conditions. We use the following algorithm to ensure all values are correctly summed into the output value. We show our proposed algorithm in Algorithm 3. The parts that must be protected from concurrent access are marked as Critical Section.

Algorithm 3. Executing a schedule

Inputs: inputs X, weights W, output Y, number of threads n_t, schedule S
Outputs: Y containing the result of the convolution

```
1:  ensure Y_i = ∅; ∀i ∈ [1, |Y|]
2:  i_s := 1
3:  while i ≤ n_t and i ≤ |S| do
4:      Start Thread i and execute:
5:          while i_s ≤ |S| do                 } Critical
6:              j := i_s                       } Section
7:              i_s := i_s + 1                 }
8:          perform load instructions
9:          t := S_j
10:         r := X_{t^c} · W_{t^p}                          ▷ HE multiplication
11:         if Y_{t^o} = ∅ then                }
12:             Y_{t^o} := r                   } Critical
13:         else                               } Section
14:             v := Y_{t^o}                   }
15:             Y_{t^o} := ∅                   }
16:         r := v + r                                      ▷ HE addition
17:         goto line 11
18:         end if
19:     end while
20:     End Thread
21: end while
```

We indicate where in the algorithm process load instructions in line 8. A load instruction has three attributes: 1.) the step that is executed on, 2.) the type of instruction, load or unload, and 3.) the object to load. Every iteration, each thread checks if there is an unprocessed load instruction with a step equal to or lower than the step the thread is executing. If there is, the thread marks it complete and executes it. Again, we must ensure that only one thread updates the load instructions at any time. Each thread tries to execute any outstanding

load instructions before moving on. Objects loaded through load instructions stay cached until explicitly unloaded through another load instruction or until the computation is complete. If a thread requires values not loaded by any load instructions, it loads them on demand and does not cache them.

4.3 Cost of a Schedule

We can use the schedule to estimate the maximum memory required on encrypted data. Maximum memory is important since we cannot execute the schedule if it requires more than the available memory. Most of the memory required during execution stems from the ciphertexts and plaintexts; therefore, we ignore additional objects like keys, the schedule, and other data in our estimation. To estimate the cost, we look at the load instructions, the number of threads, and the objects loaded on demand. We first examine the simpler case with only one thread and extend it to multiple threads later. Let s_x be the size of a single $\mathbf{x}'$, s_w the size of a single $\mathbf{w}'$, and s_y the size of a single $\mathbf{y}'$. To estimate the memory requirement of a schedule, we need to perform the following steps:

1. Split the schedule into parts at the load instructions so that each part begins with load instructions and contains no other load instructions except those at the beginning. A part must not only contain load instructions.
2. For each part, count how many $\mathbf{x}'$, $\mathbf{w}'$, and $\mathbf{y}'$ are loaded and unloaded.
3. Weight the count of $\mathbf{x}'$, $\mathbf{w}'$, and $\mathbf{y}'$ by s_x, s_w, and s_p, respectively.
4. For every step, weigh the on-demand loaded objects and add them.
5. For each part, add the weighted counts from step 3 and the maximum from step 4. The maximum of all the parts is our estimate for the schedule.

We now extend the estimation to multiple threads. The estimate for multiple threads is less precise than that for a single thread since we can only make assumptions about how multiple threads will interact. We make the following simplifying assumptions: 1.) threads execute schedule steps at the same speed, and 2.) a continuous block of load instructions is executed simultaneously, no matter how many instructions are in that block. The main ideas are that if we have split the schedule into parts that contain fewer steps than we have threads n_t, we merge adjacent parts until all parts contain at least as many steps as we have threads available. Then we identifiy the n_t steps that require the most memory in each part. To do this we start as we did in the single thread case above. Next, we look at the number of steps in each part. If the part has fewer steps than the number of threads n_t, we combine it with the next part to form a new part by adding the cost of the load instructions. We repeat this until the new combined part has more steps than threads. We repeat this for all parts of the schedule. To estimate the cost of the on-demand loaded objects, we assume that n_t schedule steps are executed at the same time. In the final step, we handle the cost of the schedule steps. We compute the on-demand cost for all steps in the schedule parts created in the previous step. The computation happens the same way as described in the single-threaded case above. However, now we not only

add the step with the highest cost; we add the n_t steps with the highest costs. This method provides a reasonable estimate for the memory cost of a schedule with multi-threaded execution.

4.4 Threat Model

In this work, we assume that all parties are honest but curious. They follow all protocols and algorithms without deviation. However, they do try to learn as much information as they can. The server offers private inference to the client. Only the dimensions of the data and the data domain need to be shared between the client and server in plaintext. The actual instances and the inference result are only ever shared in encrypted form. Besides the input and output dimensions of the model, the client gains no additional information about the model. However, model extraction attacks by the client, as described by Tramer et al. [45], can still threaten the server-side model. Additionally, since we rely on the CKKS scheme, the client needs to make sure not to share decrypted inference results with the server since this can be used to compromise the security of the client's secret key [35]. Our proposed approach reorders the operation the server performs of the encrypted data, which is not observable by the client. The client can only observe the time the server needs to perform the computation. Without knowing the server's hardware configuration, this does not provide any useful information to the client. Even if the client knows the exact hardware configuration, it can not learn any other information than it would learn if the server used a different computational model.

5 Reduced Memory Schedules

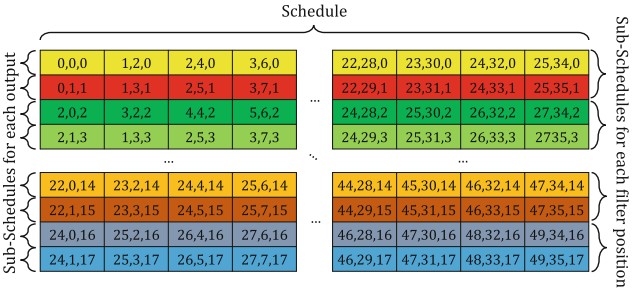

Fig. 1. Breaking an example base schedule down into multiple sub-schedules. This schedule is executed row-wise.

In this section, we propose different ways to construct schedules. These schedules provide trade-offs between runtime and memory. The fastest we can execute a schedule is by loading all data at the beginning of the computation and then

using the lock-free Algorithm 1. However, this requires a large amount of memory. We can reduce the memory footprint by loading everything on demand. However, this increases runtime significantly.

We can transform the computation performed by Algorithm 1 into a schedule. Again, consider n_t to be the number of threads, n_o the number of outputs of the computation, and n_f the number of products that sum up into a single output. In Algorithm 1, every thread executes a subschedule where all $t^y \mod i = 0; t \in S$, where i is the thread id. We can obtain the combined schedule by taking all subschedules and interleaving them elementwise. See Fig. 2 for an example with three threads.

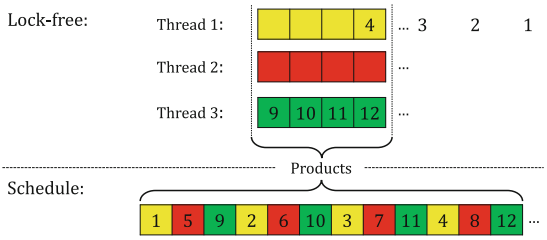

Fig. 2. Example of how to turn a lock-free execution with three threads into a schedule

The lock-free algorithm computes and needs to keep in memory n_t outputs simultaneously. The base schedule, on the other hand, fully computes a single output before moving on to the next one. This allows us to keep fewer outputs in memory. This is the lowest amount of memory we can achieve. However, we need to load objects from disk frequently and are not using any caching. Caching aims to reduce the number of loading operations as much as possible. We can exploit the regular structure of convolutions to find the best values for caching. We can split a schedule into a regular, repeating pattern defined by the size and number of filters and input channels. In two-dimensional convolutions, as used in neural networks, we have a four-dimensional filter volume, W, where the dimensions are in order: i, j the position in the filter, c_{in} the input channel, and c_{out} the output channel. We move W across the entire input, creating c_{out} outputs at every position. Note how far W we move the filter is given by the stride, which we assume to be one here. However, our method remains applicable to other stride values. Each output, at a given position of W, uses the same values from X. We call each unique position of the filter on the inputs the filter position or window.

We need to keep three kinds of objects in memory during the computation. Inputs $\mathbf{x}$, weights $\mathbf{w}$ and, outputs $\mathbf{y}$. We design multiple caching strategies based on the memory available. We will not go over the trivial case that we can fit all values of $\mathbf{x}$ and $\mathbf{w}$ into memory.

5.1 Caching by Object Type

The simplest caching strategy, is to load either all values from X' or W' at the beginning of the computation and load the other values on demand as needed. This strategy creates very simple schedules; however, it underutilizes caching. If we preload all $\mathbf{x}'$, we load too many values much earlier than needed in the computation, and if we preload all $\mathbf{y}'$ we need to load $\mathbf{x}'$ values frequently.

5.2 Full Window Caching

We can improve the caching by object type strategy by utilizing the underlying structure of the convolution operation. To obtain the output values we move the filter across the input values. Each filter channel creates one output value. The filter values at every position are the same. Therefore, if we can load them only once and cache them for the duration of the computation we can save a significant amount of load operations. However, for each position the input values change. Each position of W requires only $|W'|/c_{\text{out}}$ $\mathbf{x}'$. If we can fit these objects and all $\mathbf{y}'$ into memory, we only load W' once. Since W' usually moves over the inputs with some overlap, i.e., the stride is smaller than the width and height of the filter, we can reuse many $\mathbf{x}'$ and only need to unload and reload a small amount. We start at the top left and move W from left to right. Once we reach the end on the right, we move down and start over on the right, repeating until we reach the bottom right. If the filter size or stride is not symmetrical, it is beneficial to change the behavior to first move in the direction that has the most overlap, reducing the number of values that need to be loaded and increasing the number of values that can be reused.

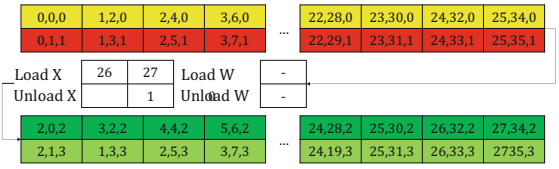

Fig. 3. Load instructions that are necessary when moving from the first window to the second using window caching with a 5×5x2 input and a 3×3x2 $\times$ 2 kernel.

5.3 Partial Window Caching

If we can fit $|W'|/c_{\text{out}}$ $\mathbf{x}'$ but not all of W' into memory, we can modify the full window caching strategy to reduce the number of loads. Let n be the number of $\mathbf{w}'$ that we can fit into memory in addition to all the $\mathbf{x}'$ in the window. We then split the schedule into sub-schedules for every position of W. To reorder the

sub-schedules to increase caching potential we reverse every second sub-schedule; see Fig. 4. This reordering makes it so that i values to the left and right of the sub-schedule boundary are the same for all $i \in [1, |W|]$. This allows us to cache n values before the sub-schedule boundary and reuse them in the next one.

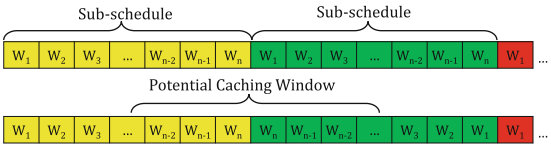

Fig. 4. Weights W_i only in Sub-Schedules that correspond to individual filter positions and how they can be reordered to increase caching potential.

5.4 Column-Wise Caching

If we cannot fit $|W|/c_{\text{out}}$ $\mathbf{x}'$ or W' into memory, we cannot use any of the caching methods described above. However, we can construct a different schedule that allows us to cache $\mathbf{x}'$ values. For this schedule, we need to be able to fit c_{out} $\mathbf{y}'$ into memory. By taking each window sub schedule and reordering it column-major instead of row-major, see Fig. 5, we can reuse the same $\mathbf{x}'$ multiple times before we unload it. This ordering requires us to keep c_{out} $\mathbf{y}'$ in memory. This ordering is most beneficial when the number of input channels is much larger than the output channels or the filter is relatively large. Both scenarios lead to a large number of $\mathbf{x}'$ in a window. Depending on how much memory is available, we can cache multiple columns. Additionally, we can combine this with the idea from partial window caching of reordering the computation to generate adjacent window subschedules that end and start with the same X values.

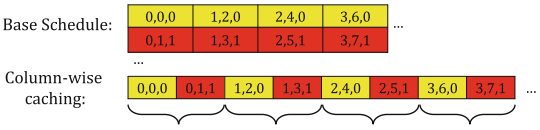

Fig. 5. Transforming the base schedule into a column-wise caching schedule

A downside of the proposed approach is that in order to achieve any benefits, we require the data to be batch-packed and a convolutional layer. Only batch-packing allows us to reorder the computation on a granular level. If this approach provides any benefits with inter-axis packing strategies is beyond the scope of this work. We need a convolutional layer to exploit its repeating weight structure. It is possible that we could use similar optimizations with recurrent layers since they also have repeating weights. However, recurrent layers impose additional challenges when used with HE [42].

6 Experimental Evaluation

We evaluate our proposed solution on the layers of a convolutional neural network (CNN) trained on the CIFAR-10 [32] dataset. We first estimate the memory requirements and then compare them to the measurements we obtain by running the model on encrypted data. Table 1 shows the model's architecture. We have two different models. One for plain data and one adapted to be HE-friendly, meaning it only contains operations that are easy to compute on encrypted data. Both models achieve very similar accuracies on the test data, 70.9% for the original model and 69.7% for the HE-friendly model. The main interest of this paper is not to propose new models or techniques that increase the accuracy of models on encrypted data but to analyze and reduce the memory consumption of these models.

Table 1. Architecture of the evaluation model with the layer parameters showing the filter size (FS), stride (S), number of filters (NF), and the activation or pooling function used on plain text (PT) and on encrypted data (HE).

Layer	Input Shape	Output Shape	Parameters
Conv 2D (1)	$32 \times 32 \times 3$	$30 \times 30 \times 32$	FS: 3×3, S: 1×1, NF: 32, PT: ReLU, HE: x^2
Pooling	$30 \times 30 \times 32$	$15 \times 15 \times 32$	FS: 2×2, S: 2×2, PT: Max, HE: Average
Conv 2D (2)	$15 \times 15 \times 32$	$13 \times 13 \times 64$	FS: 3×3, S: 1×1, NF: 64, PT: ReLU, HE: x^2
Pooling	$13 \times 13 \times 64$	$6 \times 6 \times 64$	FS: 2×2, S: 2×2, PT: Max, HE: Average
Conv 2D (3)	$6 \times 6 \times 64$	$4 \times 4 \times 64$	FS: 3×3, S: 1×1, NF: 64, PT: ReLU, HE: x^2
Flatten	$4 \times 4 \times 64$	1024	-
Dense	1024	64	Units: 64, PT: ReLU, HE: x^2
Dense	64	10	Units: 10

We define three sets of crypto parameters: small, medium, and large. All parameters guarantee at least 128-bit security. We use OpenFHE [5] as the underlying crypto library in our implementation. The small parameters have a ring dimension of 2^{14} and a multiplicative depth of 2. The medium parameters have a ring dimension of 2^{14} and a multiplicative depth of 8. And the large parameters have a ring dimension of 2^{15} and a multiplicative depth of 19. This results in a ciphertext size of 0.75 MB, 2.225 MB, and 10 MB for the small, medium, and large parameters respectively. A plaintext is always half the size of a ciphertext. We have two machines. One with 16 cores, 20 GB of memory, and 32 GB of operating system (OS) swap space, and another with 104 cores and 768 GB of memory. Both machines have two TB solid-state drives. We define different schedules, then estimate the required memory using the technique described in Sect. 4.3, and finally execute the schedules to obtain real measurements.

We define several schedules that we estimate and measure the memory requirements for. The names of the schedules are given italicized. We use the Lock-free algorithm (Algorithm 1) as our baselines once we load all values on demand (*Lock-free on demand*) and once we preload all values before execution *Lock-free Preload*. We compare these baselines to their direct equivalent using our proposed algorithm (Algorithm 3), where we preload all values (*Preload everything*). Next, we investigate the behavior when we either preload all of X', *Preload* X', or all W' values *Preload* W', x' *on demand*. Finally, we look closer at the window, partial window, and column-wise caching. For (partial) window caching, we always load all of $\mathbf{x}'$ in the window and investigate the following strategies for loading $\mathbf{w}'$s:

- load all of W', *Load* X' *window* W'
- load $\mathbf{w}'$s on demand, *Load* X' *window,* w' *on demand*
- load half of W' values, *Load* X' *window,* $W'/2$
- load a quarter of W', *Load* X' *window,* $W'/4$

We only cache one $\mathbf{x}'X$, *Column Major* for column-wise caching. For all schedules, we cache the y's from their first appearance in the schedule to their last.

6.1 Memory Estimate

To demonstrate that our proposed solution is scalable from large servers to consumer hardware, we run the selected schedules on two different machines. A desktop PC with a 16-core AMD Ryzen CPU, 20 GB of RAM, 32 GB of swap space, and a large server with two Intel 54-core CPUs and 756 GB of RAM. Both machines have a 2 TB solid-state drive and run Ubuntu Linux 20.04 LTS. In the tables and figures throughout this paper, we refer to the server and PC by their number of threads: 104 and 16, respectively.

We use the algorithm described in Sect. 4.3 to estimate the cost of all convolutional layers for small, medium, and large parameters and 16 and 104 threads. We need to estimate the memory requirements based on the number of threads that are used during execution since that can influence the number of objects in memory. The estimate column in Table 2, 3, and 4 shows the estimates for each layer and schedule for large parameters (for the small parameters see the appendix). We can see that, especially for the large parameters, the estimate frequently goes beyond the 20 GB of the PC. The estimate also often exceeds the 52 GB of memory and swap space combined. The estimate never exceeds the 756 GB of the server. For the estimate and following experiments, we assume the input X' is encrypted while the model W' is in plain.

Unsurprisingly, the schedules that preload all objects, *Preload everything* and *Lock-free Preload*, have the highest memory estimate. On the other hand, schedules that load most objects on demand and cache very little, *Lock-Free on demand* and *Column Major* have the lowest memory estimate. For the Conv 2d (1) layer, the estimates range from 380 MB to about 35 GB. Schedules that do

not load all of X' are significantly below that value, estimated at most 6193 MB. For the second layer, Conv 2D (2), both the number of $\mathbf{x}'$ and $\mathbf{w}'$ is significantly larger. This, however, does not significantly change the estimate for the *Lock-Free on demand* schedule. This observation also holds for the next layer, Conv 2D (3). The estimation aligns with the insights of a theoretical analysis of the execution. As discussed earlier, during runtime, this schedule has at most n_t of each $\mathbf{x}'$, $\mathbf{w}'$, and $\mathbf{y}'$ in memory, where n_t is the number of threads. Therefore, the memory consumption of the schedule is only influenced by the number of threads and independent of the layer. For the Conv 2D (2) layer, we also encounter values outside the PC's available memory, ranging from 400 MB to 164 GB. We see a similar picture for the last convolutional layer, Conv 2d (3). Large estimates of up to 208 GB, especially for layers that load and cache W' values.

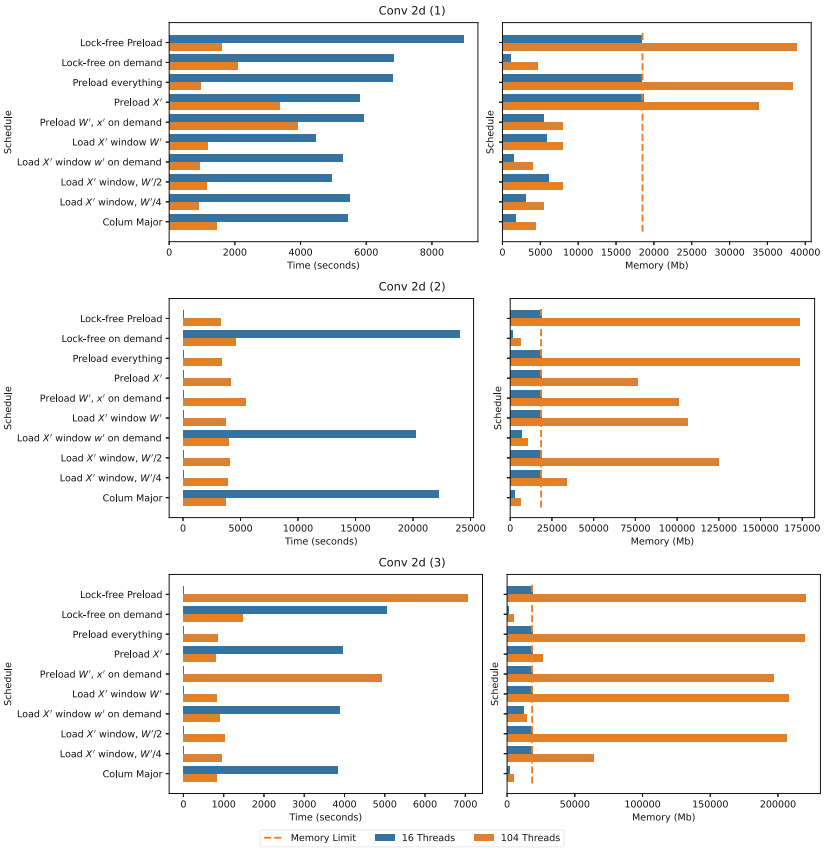

Fig. 6. Time and Memory requirements schedules, run with large parameters, on the 104 threads servers and the 16 threads PC. The memory graphs also include the PC's memory limit of 18000 MB.

6.2 Measurements

After obtaining the estimates, we execute the schedules on both the server and the PC. We measure the time it takes to execute the schedules, and the memory the process requires. For the memory measurement, it is important to note that it does include swap memory and only measures actual main memory usage. The PC has 20 GB of memory, about 1.5 GB of which the OS uses, leaving about 18.5 GB for the execution of the schedule. Therefore, measurements in the range of 18.5 GB on the PC will likely have used the OS's swapping mechanism, especially if the estimated value is much larger. As mentioned in the previous section, for some schedules, the memory available is insufficient, even with swapping. In these cases, the execution is terminated by the OS, yielding no result. We deliberately leave the OS swapping mechanism on to test if our implementation is faster than simply relying on the in-built OS methods. We further assign each schedule a score combining time and memory requirements. To calculate the score, we compute the geometric mean of the time t and m as $\sqrt{tm}$. The lower the score, the better. However, the schedule with the lowest score is automatically the best schedule on a given machine. The best schedule is typically the schedule that executes the fastest on the machine. It is possible for a slower schedule to achieve a lower score due to it requiring less memory. This, however, indicates that we could perform the computation on a machine with less memory.

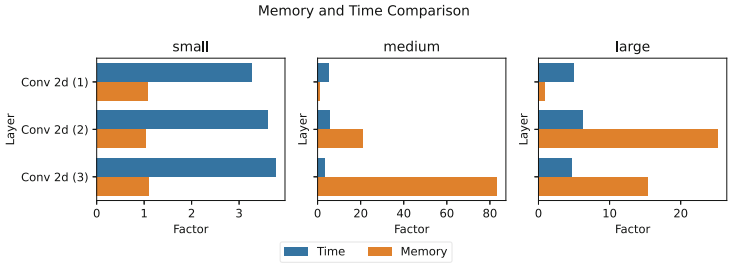

Fig. 7. Comparison of the fastest schedule for each layer with 16 and 104 threads. For each layer, the Figure shows the increase factor in runtime from 104 to 16 threads and the increase factor in memory from 16 to 104 threads for the fastest schedule

Tables 2, 3, and 4 list the time and memory requirements and the score, using the large crypto parameters (for medium and small, see the Appendix). The first important observation is the accuracy of the estimation algorithm. We expect the memory measurements to be larger than the estimate since there is runtime overhead, like the schedule itself, key material, and other data structures that the estimation does not take into account. However, in some cases, the estimate is off by a factor of 4–5. This is especially true for smaller values. An explanation for the discrepancy in estimate and measurements most likely lies in how we process cache instructions that drop data from memory. To ensure that we do not delete data that other threads still need, we only execute the delete instructions once all

threads have passed the point for which the instructions are scheduled. During execution, we have little control over how fast threads advance. It is certainly possible for some threads to fall far behind, waiting for locks or input/output operations, thereby preventing the deletion of objects from memory. We have no way of predicting how the threads will interact at runtime and, therefore, need to make simplifying assumptions that can cause the differences in estimated and measured values. Overall, the estimate can still provide us with a useful tool to understand the schedule's memory requirements without running it.

The most important metric is time. The schedule that executes the fastest is typically the schedule that uses the available resources the most efficiently. Figure 6 shows the time and memory requirement for large parameters and all schedules. Note that the OS terminated schedules that do not display a time for 16 Threads (the PC) for running out of memory. For Conv 2d (2) and Conv 2d (3) we can see multiple schedules that reach the critical limit of 18000 GB memory, after which the OS's swapping system kicks in. On the medium parameters and Conv 2d (2) (complete Figures and Table in the Appendix), we observe that *Load*

Table 2. Time in s, **Memory** in MB requirements, for all Schedules on Conv 2d (1) with large parameters on the PC with 16 and the server with 104 Threads. Additionally, shown are memory **Estimate** in MB and **Score**.

Schedule	Threads	Time	Memory	Estimate	Score
Lock-free Preload	16	8952	18432	35215	12845
	104	1600	38839	36095	7883
Lock-free on demand	16	6830	1179	400	2838
	104	2091	4662	2601	3123
Preload everything	16	6805	18408	35075	11192
	104	973	38365	35885	6109
Preload X'	16	5812	18680	30833	10420
	104	3379	33826	32084	10691
Preload W', x' on demand	16	5911	5459	4502	5681
	104	3925	7965	6193	5591
Load X' window W'	16	4458	5857	4622	5110
	104	1183	8036	5432	3083
Load X' window w' on demand	16	5278	1486	380	2801
	104	933	4074	1631	1950
Load X' window, $W'/2$	16	4955	6104	2461	5499
	104	1144	7972	3271	3021
Load X' window, $W'/4$	16	5498	3061	1451	4102
	104	899	5504	2701	2225
Colum Major	16	5442	1736	410	3074
	104	1452	4363	2531	2517

Table 3. textbfTime in s, **Memory** in MB requirements, for all Schedules on Conv 2d (2) with large parameters on the PC with 16 and the server with 104 Threads. Additionally, shown are memory **Estimate** in MB and **Score**.* indicate out of memory.

Schedule	Threads	Time	Memory (MB)	Estimate	Score
Lock-free Preload	16	*	18875	164390	
	104	3257	173712	165270	23787
Lock-free on demand	16	24060	1531	400	6069
	104	4599	6016	2601	5260
Preload everything	16	*	18819	164250	
	104	3364	173326	164250	24146
Preload X'	16	*	18949	72131	
	104	4138	76506	72571	17792
Preload W', x' on demand	16	*	18816	92379	
	104	5416	101272	93260	23419
Load X' window W'	16	*	18754	95110	
	104	3662	106345	95110	19735
Load X' window w' on demand	16	20246	6888	2991	11809
	104	3914	10203	3431	6319
Load X' window, $W'/2$	16	*	18839	49011	
	104	4003	125113	49011	22379
Load X' window, $W'/4$	16	*	18871	26031	
	104	3843	33731	26471	11385
Colum Major	16	22251	2809	730	7906
	104	3724	6516	2291	4926

X' window W' schedule reaches the swapping limit and takes 10675 s. The *Load X' window w' on demand* schedule does not reach that limit needing ∼2 GB. However, despite needing to encode data more often, it is faster at 3885 s. This strongly suggests that our algorithm is more efficient than relying on the OS's swapping mechanism.

Table 5 and Fig. 7 compare the fastest schedule for each layer and set of parameters. We are most interested in the increase in runtime and the reduction in memory when running on the 16-thread PC as compared to running on the 104-thread server. For the small parameters, the fastest schedule is either the *Lock-free Preload* or *Preload everything* schedule. Since these schedules have very similar memory requirements, there is no significant reduction in memory. The time, however, increases by a factor of 3.3–3.8. We start to see a much bigger difference when moving to the medium parameters. For the Conv 2d (1) layer, the time increases by a factor of 5.4 while the memory usage stays almost the same between PC and server. For this layer, both systems can still use the *Lock-free Preload* schedule, which explains the negligible reduction in memory. The

Table 4. Time in s, **Memory** in MB requirements, for all Schedules on Conv 2d (3) with large parameters on the PC with 16 and the server with 104 Threads. Additionally, shown are memory **Estimate** in MB and **Score**.* indicate out of memory.

Schedule	Threads	Time	Memory (MB)	Estimate	Score
Lock-free Preload	16	*	18788	207608	
	104	7074	220310	208489	39479
Lock-free on demand	16	5054	1133	400	2393
	104	1469	4823	2601	2662
Preload everything	16	*	18838	207468	
	104	855	219812	207468	13709
Preload X'	16	3955	18910	23150	8648
	104	815	26310	23590	4631
Preload W', x' on demand	16	*	18859	184578	
	104	4935	196842	185459	31168
Load X' window W'	16	*	18775	190191	
	104	831	207599	190191	13134
Load X' window w' on demand	16	3876	12218	5872	6881
	104	913	14437	6313	3631
Load X' window, $W'/2$	16	*	18860	97992	
	104	1024	206385	97992	14539
Load X' window, $W'/4$	16	*	18871	51962	
	104	955	63491	52402	7788
Colum Major	16	3834	1714	730	2564
	104	825	4609	2291	1950

time increase for the next two layers is 5.9 and 3.5, respectively; however, the memory reduction is significant and a factor of 21 and 83.3. While the server still uses the *Lock-free Preload* schedule the PC is forced to use window caching and column-wise window caching to fit the objects into memory. The picture repeats for the large parameters. Except that now the server uses a more memory-efficient schedule for Conv 2d (3), which leads to only 15.3 times memory reduction and an increase in runtime by 4.7. An interesting observation: on the small parameters, the PC seems to have a higher per-thread performance as the time increase is only around 3.5 for all layers despite the number of threads on the server being 6.5 more. As the parameters get larger, the time increase seems to approach 6.5 as expected.

Additionally, we compare the time and memory of the different schedules run on the large crypto parameters executed on the server. For the Conv 2d (1) layer the fastest schedule is *Load X' window, $W'/4$*. It is 74 s, 8%, faster than the *Preload everything schedule*. The *Preload everything schedule*, in turn, is much

Table 5. Fastest Schedule for each layer and parameter size (Param.) on the server, 104 Threads (T), and PC, 16 Threads. As well as the increase (Inc.) in time and reduction (Red.) of memory.

Param	Layer	T	Time	Inc.	Memory	Red	Schedule
small	Conv 2d (1)	16	135	3.3	3084	1.1	Lock-free Preload
		104	41		3319		Lock-free Preload
	Conv 2d (2)	16	580	3.6	15164	1.0	Preload everything
		104	160		15570		Lock-free Preload
	Conv 2d (3)	16	116	3.8	18699	1.1	Lock-free Preload
		104	30		20727		Lock-free Preload
medium	Conv 2d (1)	16	859	5.4	8447	1.1	Lock-free Preload
		104	159		9141		Lock-free Preload
	Conv 2d (2)	16	3885	5.9	1939	21.0	Load X' window w' on demand
		104	653		40775		Lock-free Preload
	Conv 2d (3)	16	743	3.5	629	83.3	Colum Major
		104	211		52415		Lock-free Preload
large	Conv 2d (1)	16	4457	5.0	5857	0.9	Load X' window W'
		104	899		5504		Load X' window, $W'/4$
	Conv 2d (2)	16	20246	6.2	6887	25.2	Load X' window w' on demand
		104	3257		173712		Lock-free Preload
	Conv 2d (3)	16	3834	4.7	1714	15.3	Colum Major
		104	815		26309		Preload X'

faster, 627 s (64%), than the *Lock-free Preload* schedule. However, both preload schedules require 38 GB of memory, compared to the 5.4 GB of the *Load* X' *window,* $W'/4$ schedule. For the second layer, Conv 2d (2), the *Lock-free Preload* schedule is the fastest at 3257 s. The *Preload every* is marginally slower at 3364 s. Both schedules require 170 GB of memory. Schedules that require significantly less memory *Load* X' *window,* $W'/4$ (33 GB) and *Column Major* (6.3 GB) are

only slightly slower at 3662 s and 3843 s. For the Conv 2d (3) layer the *Lock-free Preload* schedule is the slowest and consumes the most memory at 7074 s and 215 GB. The comparable *Preload everything* schedule requires approximately the same amount of memory, but only 12.5% of the time, 855 s. Interestingly, schedules that cache very little *Preload* X' and *Column Major* are faster than the *Preload everything* at 815 s and 825 s. Both of these schedules also require significantly less memory, at 25.7 GB and 4.5 GB. This is a reduction factor of 47.8 between *Lock-free Preload* and *Column Major*.

Interestingly, schedules with minimal caching of w's are often faster than schedules that substantially cache these values. A potential explanation could be the cache locality inside the CPU. Values that are not cached by our method and are loaded on demand could be accessed faster because they are placed inside the CPU cache. Alternatively, locking that is required for processing the load instructions could introduce additional slowdowns that are not present when values are loaded on demand. Another interesting observation is the poor performance of the *Lock-free Preload* schedule in the Conv 2d (3) layer. It is eight times slower than *Preload everything* schedule. Both schedules load all the required data at the start of the computation and do not need to load any values during. Where they differ is the points at which they write the results to disk. If we assume that all threads advance in lockstep, in the *Lock-free Preload* schedule, all threads want to write to disk at once. In *Preload everything* schedule, the write operations are more spaced out. It could be that the large number of simultaneous writes slows the schedule down significantly.

7 Conclusion

In this paper, we present ways of reordering the computation to tailor the memory requirements to the hardware available while executing as fast as possible. We further present a technique to estimate the required memory of convolutions over batch-packed, encrypted data. We show that our proposed caching mechanism is faster than relying on the OS's swapping mechanism. The method proposed in this paper is especially suited for ML workloads with thousands of instances that can run longer, i.e., overnight or over the weekend, and don't need a fast turnaround. Since our method can reduce the memory requirements for inference, it opens up the potential to save on hardware costs.

Appendix

Table 6. Measurements for **Time** in seconds, Memory (**Mem**) in MB, for all Schedules with small parameters on the PC with 16 and the server with 104 Threads. Additionally, the table shows the memory Estimate in MB and **Score**.

Layer	Schedule	Threads	Time	Memory (MB)	Estimate	Score
Conv 2d (1)	Lock-free Preload	16	136	3084	2645	648
		104	42	3319	2711	372
	Lock-free on demand	16	433	290	30	354
		104	131	652	195	292
	Preload everything	16	142	3024	2634	656
		104	45	3316	2695	387
	Preload X'	16	184	2593	2316	691
		104	52	2872	2409	385
	Preload W', x' on demand	16	333	724	338	491
		104	108	1027	465	333
	Load X' window W'	16	143	771	347	332
		104	46	1186	408	234
	Load X' window w' on demand	16	182	325	29	243
		104	56	717	122	199
	Load X' window, $W'/2$	16	201	818	185	405
		104	57	1197	246	261
	Load X' window, $W'/4$	16	190	450	109	293
		104	74	972	203	268
	Colum Major	16	233	599	31	373
		104	131	891	190	342
Conv 2d (2)	Lock-free Preload	16	583	15327	12346	2990
		104	161	15570	12412	1583
	Lock-free on demand	16	1708	529	30	951
		104	524	892	195	683
	Preload everything	16	581	15165	12335	2968
		104	187	15446	12335	1700
	Preload X'	16	727	5930	5417	2076
		104	205	6187	5450	1127
	Preload W', x' on demand	16	1341	9729	6938	3613
		104	449	10054	7004	2126
	Load X' window W'	16	609	10180	7143	2490
		104	194	10427	7143	1423
	Load X' window w' on demand	16	765	934	225	845
		104	210	1176	258	497
	Load X' window, $W'/2$	16	816	10281	3681	2896
		104	239	11778	3681	1679
	Load X' window, $W'/4$	16	796	3257	1955	1610
		104	223	3770	1988	917
	Colum Major	16	853	1099	55	968
		104	492	1434	172	840
Conv 2d (3)	Lock-free Preload	16	117	18700	15591	1477
		104	31	20728	15657	800
	Lock-free on demand	16	329	254	30	289
		104	101	614	195	249
	Preload everything	16	132	19362	15581	1601
		104	45	20710	15581	970
	Preload X'	16	141	1980	1739	528
		104	39	2221	1772	292
	Preload W', x' on demand	16	278	18688	13862	2281
		104	95	18975	13928	1346
	Load X' window W'	16	134	19276	14283	1605
		104	48	19769	14283	972
	Load X' window w' on demand	16	146	1085	441	399
		104	43	1330	474	239
	Load X' window, $W'/2$	16	167	19254	7359	1792
		104	56	20839	7359	1081
	Load X' window, $W'/4$	16	152	5781	3902	938
		104	45	6374	3935	539
	Colum Major	16	160	416	55	258
		104	94	787	172	272

Table 7. Measurements for **Time** in seconds, Memory (**Mem**) in MB, for all Schedules with medium parameters on the PC with 16 and the server with 104 Threads. Additionally, the table shows the memory Estimate in MB and **Score**. * values are unavailable because the execution ran out of memory.

Layer	Schedule	Threads	Time	Memory (MB)	Estimate	Score
Conv 2d (1)	Lock-free Preload	16	859	8448	7928	2695
		104	160	9142	8126	1208
	Lock-free on demand	16	1120	409	90	676
		104	233	1428	586	576
	Preload everything	16	873	8422	7897	2712
		104	171	9286	8079	1258
	Preload X'	16	1014	7304	6942	2721
		104	215	8182	7223	1327
	Preload W', x' on demand	16	973	1472	1014	1197
		104	188	2375	1394	669
	Load X' window W'	16	891	1600	1041	1194
		104	174	2634	1223	677
	Load X' window w' on demand	16	1023	519	86	729
		104	188	1600	367	548
	Load X' window, $W'/2$	16	988	1651	554	1277
		104	194	2439	737	688
	Load X' window, $W'/4$	16	993	797	327	889
		104	185	1732	608	567
	Colum Major	16	1177	751	92	940
		104	242	1726	570	646
Conv 2d (1)	Lock-free Preload	16	17154	18531	37011	17829
		104	654	40776	37209	5162
	Lock-free on demand	16	4636	631	90	1710
		104	944	1668	586	1255
	Preload everything	16	16589	18764	36979	17643
		104	769	40970	36979	5612
	Preload X'	16	3946	16852	16239	8155
		104	881	17736	16339	3952
	Preload W', x' on demand	16	12054	19356	20798	15275
		104	955	24631	20996	4850
	Load X' window W'	16	10675	19036	21413	14255
		104	803	25897	21413	4561
	Load X' window w' on demand	16	3885	1939	673	2745
		104	958	2864	773	1656
	Load X' window, $W'/2$	16	11570	18950	11034	14807
		104	1069	29330	11034	5598
	Load X' window, $W'/4$	16	3989	8043	5861	5664
		104	1438	10983	5960	3974
	Colum Major	16	3933	1395	164	2342
		104	1759	2758	516	2203
Conv 2d (1)	Lock-free Preload	16	*	18640	46741	
		104	212	52416	46939	3331
	Lock-free on demand	16	817	387	90	562
		104	329	1396	586	677
	Preload everything	16	*	18971	46709	
		104	325	52493	46709	4131
	Preload X'	16	748	5552	5212	2038
		104	283	6296	5311	1336
	Preload W', x' on demand	16	5237	18853	41556	9936
		104	547	47242	41754	5083
	Load X' window W'	16	*	18821	42819	
		104	347	49830	42819	4159
	Load X' window w' on demand	16	753	2950	1322	1491
		104	263	3475	1421	956
	Load X' window, $W'/2$	16	*	19015	22062	
		104	441	52804	22062	4826
	Load X' window, $W'/4$	16	752	15296	11699	3392
		104	354	17400	11798	2482
	Colum Major	16	744	630	164	684
		104	323	1715	516	745

References

1. Abadi, M., et al.: Deep learning with differential privacy. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 308–318 (2016)
2. Aharoni, E., et al.: HeLayers: A Tile Tensors Framework for Large Neural Networks on Encrypted Data. Proceedings on Privacy Enhancing Technologies 2023(1), 325–342 (Jan 2023). https://doi.org/10.56553/popets-2023-0020, http://arxiv.org/abs/2011.01805, arXiv:2011.01805 [cs]
3. Akavia, A., Oren, N., Sapir, B., Vald, M.: Compact storage for homomorphic encryption. Cryptology ePrint Archive (2022)
4. Al Badawi, A., et al.: Towards the AlexNet Moment for Homomorphic Encryption: HCNN, the First Homomorphic CNN on Encrypted Data with GPUs. IEEE Trans. Emerg. Topics Comput. (2020). https://doi.org/10.1109/TETC.2020.3014636, conference Name: IEEE Transactions on Emerging Topics in Computing
5. Al Badawi, A., et al.: OpenFHE: open-source fully homomorphic encryption library. in: proceedings of the 10th workshop on encrypted computing & applied homomorphic cryptography, pp. 53–63. WAHC'22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3560827.3563379 event-place: Los Angeles, CA, USA
6. Amazon.com, I.: Amazon alexa voice ai, alexa developer offical site. https://developer.amazon.com/en-US/alexa Accessed 17 Oct 2023
7. Boemer, F., Costache, A., Cammarota, R., Wierzynski, C.: 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, pp. 45–56 (2019)
8. Boemer, F., Lao, Y., Cammarota, R., Wierzynski, C.: ngraph-he: a graph compiler for deep learning on homomorphically encrypted data. In: Proceedings of the 16th ACM International Conference on Computing Frontiers, pp. 3–13 (2019)
9. Brutzkus, A., Gilad-Bachrach, R., Elisha, O.: Low latency privacy preserving inference. In: International Conference on Machine Learning, pp. 812–821. PMLR (2019)
10. Cai, Y., Zhang, Q., Ning, R., Xin, C., Wu, H.: Hunter: he-friendly structured pruning for efficient privacy-preserving deep learning. In: Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security, pp. 931–945 (2022)
11. Chabanne, H., de Wargny, A., Milgram, J., Morel, C., Prouff, E.: Privacy-preserving classification on deep neural network. IACR Cryptol. ePrint Arch. **2017**, 35 (2017)
12. Chaudhari, H., Rachuri, R., Suresh, A.: Trident: efficient 4pc framework for privacy preserving machine learning. In: Proceedings 2020 Network and Distributed System Security Symposium. NDSS 2020, Internet Society (2020). https://doi.org/10.14722/ndss.2020.23005, http://dx.doi.org/10.14722/ndss.2020.23005
13. Cheon, J.H., Han, K., Kim, A., Kim, M., Song, Y.: A full RNS variant of approximate homomorphic encryption. In: Cid, C., Jacobson, M.J. (eds.) Selected Areas in Cryptography – SAC 2018: 25th International Conference, Calgary, AB, Canada, August 15–17, 2018, Revised Selected Papers, pp. 347–368. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-10970-7_16
14. Choi, W.S., Reagen, B., Wei, G.Y., Brooks, D.: Impala: Low-Latency, Communication-Efficient Private Deep Learning Inference. arXiv preprint arXiv:2205.06437 (2022)

15. Dathathri, R., et al.: CHET: an optimizing compiler for fully-homomorphic neural-network inferencing. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 142–156. PLDI 2019, Association for Computing Machinery, New York, NY, USA (Jun 2019). https://doi.org/10.1145/3314221.3314628, https://doi.org/10.1145/3314221.3314628
16. Dilsizian, S.E., Siegel, E.L.: Artificial intelligence in medicine and cardiac imaging: harnessing big data and advanced computing to provide personalized medical diagnosis and treatment. Curr. Cardiol. Rep. **16**, 1–8 (2014)
17. Dowlin, N., Gilad-Bachrach, R., Laine, K., Lauter, K., Naehrig, M., Wernsing, J.: Cryptonets: applying neural networks to encrypted data with high throughput and accuracy. In: International Conference on Machine Learning, pp. 201–210 (2016)
18. Dwork, C., Roth, A.: The algorithmic foundations of differential privacy. Found. Trends Theor. Comput. Sci. **9**(3–4), 211–407 (2014)
19. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press (2016). http://www.deeplearningbook.org
20. Google, I.: Bard - chat based ai tool from google, powered by palm2. https://bard.google.com/ Accessed 17 Oct 2023
21. Google, I.: Google assitant, your own personal google. https://assistant.google.com/Accessed 17 Oct 2023
22. Grammarly, I.: Grammarly: free writing ai assistance. https://www.grammarly.com/Accessed 17 Oct 2023
23. Hao, M., Li, H., Chen, H., Xing, P., Xu, G., Zhang, T.: Iron: Private Inference on Transformers. In: Advances in Neural Information Processing Systems (2022)
24. Hesamifard, E., Takabi, H., Ghasemi, M.: Cryptodl: Deep neural networks over encrypted data. arXiv preprint arXiv:1711.05189 (2017)
25. Hesamifard, E., Takabi, H., Ghasemi, M.: Deep Neural networks classification over encrypted data. In: Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy, pp. 97–108. ACM, Richardson Texas USA (Mar 2019). https://doi.org/10.1145/3292006.3300044
26. Huang, Z., Lu, W.j., Hong, C., Ding, J.: Cheetah: Lean and Fast Secure $$two-party$$ Deep Neural Network Inference. In: 31st USENIX Security Symposium (USENIX Security 22), pp. 809–826 (2022)
27. Inc., A.: Siri - apple. https://www.apple.com/siri/ Accessed 17 Oct 2023
28. Jiang, X., Kim, M., Lauter, K., Song, Y.: Secure outsourced matrix computation and application to neural networks. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 1209–1222. ACM, Toronto Canada (Oct 2018). https://doi.org/10.1145/3243734.3243837
29. Juvekar, C., Vaikuntanathan, V., Chandrakasan, A.: GAZELLE: a low latency framework for secure neural network inference. In: 27th USENIX Security Symposium (USENIX Security 18), pp. 1651–1669 (2018)
30. Kashyap, A., Plis, S., Ritter, P., Keilholz, S.: A deep learning approach to estimating initial conditions of brain network models in reference to measured fmri data. Front. Neurosci. **17** (2023)
31. Kim, D., Park, J., Kim, J., Kim, S., Ahn, J.H.: HyPHEN: A Hybrid Packing Method and Optimizations for Homomorphic Encryption-Based Neural Networks. arXiv preprint arXiv:2302.02407 (2023)
32. Krizhevsky, A., Hinton, G.: Learning multiple layers of features from tiny images,: publisher: Toronto. ON, Canada (2009)
33. Lee, E., et al.: Low-complexity deep convolutional neural networks on fully homomorphic encryption using multiplexed parallel convolutions. In: International Conference on Machine Learning, pp. 12403–12422. PMLR (2022)

34. Lee, J.W., et al.: Privacy-Preserving Machine Learning With Fully Homomorphic Encryption for Deep Neural Network. IEEE Access **10**, 30039–30054 (2022). https://doi.org/10.1109/ACCESS.2022.3159694, conference Name: IEEE Access
35. Li, B., Micciancio, D.: On the security of homomorphic encryption on approximate numbers. In: Canteaut, A., Standaert, F.-X. (eds.) Advances in Cryptology – EUROCRYPT 2021: 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17–21, 2021, Proceedings, Part I, pp. 648–677. Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-77870-5_23
36. Li, S., et al.: FALCON: a fourier transform based approach for fast and secure convolutional neural network predictions. In: 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), pp. 8702–8711. IEEE, Seattle, WA, USA (Jun 2020). https://doi.org/10.1109/CVPR42600.2020.00873, https://ieeexplore.ieee.org/document/9156980/
37. Liu, J., Juuti, M., Lu, Y., Asokan, N.: Oblivious neural network predictions via minionn transformations. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 619–631 (2017)
38. Mohassel, P., Zhang, Y.: SecureML: a system for scalable privacy-preserving machine learning. In: 2017 IEEE Symposium on Security and Privacy (SP), pp. 19–38 (May 2017). https://doi.org/10.1109/SP.2017.12, iSSN: 2375-1207
39. OpenAI: Chatgpt. https://openai.com/chatgpt Accessed 17 Oct 2023
40. Papernot, N., Song, S., Mironov, I., Raghunathan, A., Talwar, K., Erlingsson, U.: Scalable private learning with pate. arXiv preprint arXiv:1802.08908 (2018)
41. Podschwadt, R., Takabi, D.: Classification of encrypted word embeddings using recurrent neural networks. In: PrivateNLP@ WSDM, pp. 27–31 (2020)
42. Podschwadt, R., Takabi, D.: Non-interactive privacy preserving recurrent neural network prediction with homomorphic encryption. In: 2021 IEEE 14th International Conference on Cloud Computing (CLOUD), pp. 65–70. IEEE (2021)
43. Shivdikar, K., et al.: Accelerating polynomial multiplication for homomorphic encryption on gpus. In: 2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED), pp. 61–72. IEEE (2022)
44. Smart, N.. P.., Vercauteren, F..: Fully homomorphic SIMD operations. Designs, Codes and Cryptography **71**(1), 57–81 (2014). https://doi.org/10.1007/s10623-012-9720-4
45. Tramèr, F., Zhang, F., Juels, A., Reiter, M.K., Ristenpart, T.: Stealing Machine Learning Models via Prediction $$apis$$. In: 25th USENIX security symposium (USENIX Security 16), pp. 601–618 (2016)
46. Zheng, M., Lou, Q., Jiang, L.: Primer: fast private transformer inference on encrypted data (Mar 2023). arXiv:2303.13679 [cs]