Memory-Efficient Deep Learning Inference in Trusted Execution Environments

Jean-Baptiste Truong, William Gallagher
Tian Guo, Robert J. Walls
{jtruong2, wfgallagher, tian, rjwalls}@wpi.edu
Worcester Polytechnic Institute

***Abstract*—This study identifies and proposes techniques to alleviate two key bottlenecks to executing deep neural networks in trusted execution environments (TEEs): page thrashing during the execution of convolutional layers and the decryption of large weight matrices in fully-connected layers. For the former, we propose a novel partitioning scheme, *y-plane partitioning*, designed to *(i)* provide consistent execution time when the layer output is large compared to the TEE secure memory; and *(ii)* significantly reduce the memory footprint of convolutional layers. For the latter, we leverage quantization and compression. In our evaluation, the proposed optimizations incurred latency overheads ranging from 1.09X to 2X baseline for a wide range of TEE sizes; in contrast, an unmodified implementation incurred latencies of up to 26X when running inside of the TEE.**

***Index Terms*—SGX, Trusted Execution Environments, Deep Learning.**

I. Introduction

Deep learning model owners often rely on others' hardware, such as cloud providers or end-users, for model execution. However, the consequent model exposure impacts user privacy, model security, and the owner's intellectual property interests [1], [3], [13], [14]. To address these issues, recent works have explored the use of *trusted execution environments (TEEs)*, i.e., isolated environments which provide a set of security features that allow running verified code safely on untrusted hardware [9], [11]. While TEEs provide a natural foundation for sensitive computations, their severe memory constraints have important performance implications. In the context of deep learning, where redundant access to large memory areas is frequent, relying solely on existing TEE paging mechanisms results in prohibitively high overheads—upwards of 26X increases in model latency (see Table I).

In this paper, we characterize two bottlenecks that impact TEE performance and consider methods to address them. For the *page thrashing bottleneck* described in Section V, we propose a data partitioning scheme, *y-plane partitioning*, that allows for efficient computation of convolutional layers in TEEs with as little as 28MB of secure memory. Additionally, in Section VI, we identify a previously unexplored performance bottleneck, the *decryption bottleneck*, that arises from parameter decryption and propose a mitigation strategy based on compression and quantization. We used SGX-based TEEs on Microsoft Azure cloud servers [6] to measure the impact of these bottlenecks and evaluate the proposed solutions. For the most extreme case (shown in Table I), the bottlenecks increased model latency to 26X over the unmodified baseline, while the proposed optimizations reduced model latency from 26X to 1.09X.

The optimizations proposed in this study significantly reduce the per-layer memory footprint for a model, which is a limiting factor of prior work such as Vessels [9]. Further, we demonstrate that the proposed y-plane partitioning scheme offers complimentary design tradeoffs, with different strengths and weaknesses, to channel partitioning [11]. Our evaluation suggests that a combination of y-plane and channel partitioning provides the smallest memory footprint for convolutional layers. The choice of scheme depends on the size of the layer's output versus the size of the weights. Finally, reducing memory footprint improves model latency and allows for greater concurrency, allowing more TEEs to coexist on the same system [9]. We leave an exploration of model concurrency for future work.

In summary, we make the following contributions:

- the introduction of a novel *y-plane partitioning* scheme that complements channel partitioning, alleviating the page thrashing bottleneck and reducing the memory footprint of convolution layers;
- a characterization of the previously unexplored decryption bottleneck in fully-connected layers;
- an evaluation of quantization and compression as a means to address the decryption bottleneck and reduce the memory footprint of fully-connected layers.

II. Background

Trusted Execution Environments. While the exact capabilities of TEE implementations vary, some of the more common security features include *(i)* isolation, i.e., the confidentiality and integrity of the code and data located inside the TEE, and *(ii)* remote attestation, i.e., the ability to verify the state of the TEE remotely. These properties are why recent works have explored TEEs as a means to protect the confidentiality of both user [11] and model [9] data when running deep learning inference on untrusted hardware.

An *SGX enclave* is a TEE implementation provided by Intel's software guard extensions (SGX) [8]. SGX enclaves include an area of *secure memory*, called the *processor reserved memory (PRM)*, which is isolated from the rest of the system. This secure memory is only accessible from code within the enclave. The secure memory size is usually small relative to

	Outside TEE	Inside TEE	
		Optimized	Unmodified
	(s)	(s)	(s)
28MB+1vCPU	3.174	3.468 (1.09X)	84.639 (26.73X)
56MB+2vCPU	1.858	2.430 (1.31X)	28.599 (15.39X)
112MB+4vCPU	1.112	1.868 (1.68X)	11.256 (10.12X)
168MB+8vCPU	0.808	1.667 (2.06X)	4.377 (05.42X)

TABLE I: Model Latency of VGG-16. The "Optimized" column records the latency improvements after applying the optimizations proposed in Sections V and VI. Each row represents an SGX enclave configuration; for example, 28MB+1vCPU means the enclave has 28MB of secure memory and 1 virtual CPU core. Numbers are averaged over 30 runs.

the rest of the system, typically far less than what deep learning models require for inference. For example, the enclaves used for this study offered between 28MB and 168MB of memory, whereas VGG-16 [15] requires over 1GB of memory. To support programs with higher memory requirements, SGX provides paging mechanisms to encrypt and swap memory pages between secure and main memory. When code running inside the enclave attempts to access a virtual memory address on a page that is not currently in the enclave, a *page fault* is raised. SGX transparently services this page fault: evicting an older page and transferring, decrypting, and checking the requested page's integrity. We refer to this as *secure paging*.

The memory constraints for SGX-based TEEs differ from the memory constraints for edge devices (e.g., mobile phones). For example, the main memory size on most mobile devices is much larger than the secure memory available to the SGX enclaves used in our experiments. Further, SGX's secure paging introduces unique encryption and decryption bottlenecks, as we discuss later.

Convolutional Neural Networks. *Convolutional neural networks (CNNs)* are a type of deep neural network that contain neurons organized into *layers*, including the eponymous convolutional layers. CNNs are commonly used for vision tasks but are garnishing attention in other domains. The process of using a CNN model for classification is referred to as *inference* or *model execution* and the time taken to perform this inference is called *model latency*.

Layer execution refers to the process of transforming the *inputs* (i.e., the output of the previous layer) and *parameters (e.g., weights)* into the *outputs* for an individual layer. The precise computation performed depends on the layer's type. We collectively refer to the model's static parameters (e.g., weights) and any values calculated at runtime as *model data*.

Broadly, CNNs use three different types of hidden layers: *fully-connected*, *convolutional*, and *pooling layers*. The inputs and outputs of convolutional and pooling layers are 3D arrays which resemble stacks of 2D images called *channels* (e.g. the RGB channels of an image). In fully-connected layers, the inputs and outputs are simple 1D vectors.

III. Related Work

Deep Learning and Trusted Execution Environments. In Vessels [9], Kim et al. optimize the memory usage of neural networks in TEEs by analyzing the dependency graph of the model's layers and then allocating a memory pool in which only the required data is stored at any given time. The rationale is that the sequential nature of neural networks' architecture allows reusing most memory buffers, avoiding unnecessary paging. Furthermore, as all of the computations are done in a pre-allocated memory area, a single machine can host multiple enclaves to compute different models concurrently. As long as the different enclaves do not fill up the secure memory, the contention is minimized. The limiting factor for such a system is the size of the memory pool, which relies primarily on the size of the largest layers.

Partitioning is one mechanism to reduce the per-layer memory requirements. For example, we propose a convolution-layer partitioning scheme, y-plane partitioning, in Section V. Another example is Occlumency's *channel partitioning*. Occlumency [11] is an inference framework implemented on top of SGX that uses channel partitioning to divide the computation and memory requirements of convolutional layers.

As our work does not propose an end-to-end system, we cannot provide a direct comparison with Occlumency and Vessels. However, we compare Occlumency's channel partitioning scheme to y-plane partitioning and explain why a combination of both schemes provides the best performance in Section V-D.

Grover et al. proposed Privado [16], a system designed to remove any input-dependent memory accesses, thereby preventing information leakage from the TEE. Chiron [7] uses SGX enclaves to train machine learning models, protecting the confidentiality of the user's training data, the model's architecture, and the training procedure. Neither work attempts to address the performance challenges described in this paper.

Encryption for Deep Learning. Cryptographic techniques offer an alternative to trusted execution environments for maintaining user privacy [5], [12]. These techniques rely on homomorphic encryption to process encrypted data on a server. Such systems usually have high inference latencies, which they make up for with high throughput. Thus, these systems are more appropriate for processing large batches of input data. Further, existing cryptographic systems like CryptoNets [5] do not protect model weights from disclosure—protecting the model confidentiality in Cryptonets would significantly degrade performance.

IV. Methodology

We conducted our experiments on virtual machines provided by Microsoft's Azure cloud computing infrastructure. The four tested enclave configurations represent all of the configurations offered by Microsoft Azure at the time of writing. We refer to each VM using its enclave size and number of virtual CPUs; for example, *28MB+1vCPU* refers to the VM configuration with 28MB of secure memory and 1 virtual CPU. All configurations ran Ubuntu 18.04 and used an Intel Xeon E-2288G CPU. Unless otherwise specified, the number of execution threads for each system was equal to the number of virtual CPUs—this is why the baseline model latency varies, for example.

Our evaluation methodology emphasizes the *per-layer* performance of convolutional neural networks (CNNs). Focusing on individual layers offers two distinct benefits. First, it allows us to examine each of the components in isolation. Second, it helps us determine the performance implications for a variety of CNN architectures. For instance, we observed that the performance benefits offered by quantization and compression for the large fully-connected layers in VGG-16 directly translated to performance benefits for the large fully-connected layer in AlexNet—though we elide the AlexNet numbers for space. Consequently, we focus primarily on the VGG architecture as VGG models contain a variety of fully-connected and convolutional layers that range in size, shape, and memory requirements.

We use Darknet as the baseline inference framework due to its portability. In particular, Darknet uses C and lacks external dependencies, making it possible to port the framework to SGX with relatively minor changes. In contrast, PyTorch is a more popular framework, but moving model execution entirely into the TEE would require significant engineering efforts.

V. The Page Thrashing Bottleneck

The mismatch between enclave size and convolutional-layer memory requirements manifests as inefficient paging patterns during model inference, i.e., a *page thrashing bottleneck*. In this section, we characterize this phenomenon. We then propose y-plane partitioning as a means to mitigate this bottleneck. We compare y-plane partitioning to a prior scheme and argue that the combination offers the best latency and smallest memory footprint.

A. Characterization

As observed by Lee et al. [11], the challenge for convolutional layers is that the memory access pattern during execution leads to page thrashing, i.e., the constant transfer of pages into and out of the TEE.[1] Every transfer between secure and main memory adds significant overhead.

Darknet, like many other frameworks, uses the im2col transformation to speed up convolutional layer execution. This transformation expands the 3D input array into a large 2D matrix and organizes the weights into a different 2D matrix. These transformations allow the convolution operations to be computed using a large matrix-matrix multiplication. They thus benefit from highly optimized *general matrix to matrix multiplication (GEMM)* functions provided by BLAS libraries (e.g. OpenBLAS [17]). This method's inherent tradeoff is that the im2col transformation duplicates the inputs, resulting in a transformed im2col matrix that is significantly larger than the original input array.

The above scheme has unintended consequences when used naively in the TEE. First, im2col's expansion of the input array—a factor of 9 in VGG-16—causes many memory pages to be evicted from the TEE only to be brought back into the TEE during the matrix-matrix multiplication. Second, as the im2col matrix cannot fit entirely in TEE memory, the pattern of memory accesses to this matrix has important performance implications. For example, in Darknet the output matrix is computed row by row, resulting in an unfavorable memory access pattern that triggers cascading evictions and page faults. In particular, computing one row of the output requires a lookup of the entire transformed im2col matrix, and this lookup process is repeated for all rows of the output.

[1]Denning [2] defines thrashing as "excessive overhead and severe performance degradation or collapse caused by too much paging."

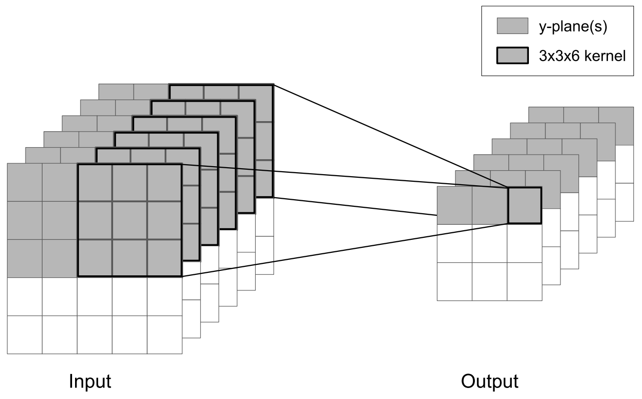

Fig. 1: Illustration of Y-Plane Partitioning. A 5x5x6 input is convolved with a 3x3x6 kernel. This figure highlights the computation of 1 output value. Three input y-planes are required to compute one output y-plane.

In our experiments with Darknet and VGG-16, we observed that convolutional layers cause more page evictions, by multiple orders of magnitude, when run in a 28MB enclave versus the 168MB enclave. Layer 8 triggers 1.8 million page evictions in the 28MB enclave, but only 1,700 in the 168MB enclave.

B. Partitioning

Partitioning addresses the thrashing bottleneck by applying the im2col transformation to a subset of the input array. The subset, i.e., partition, can be processed efficiently using the limited secure memory of the TEE.

We evaluate two partitioning schemes: *y-plane partitioning* and *channel partitioning*. The former is a contribution of this paper, and the latter was previously used as part of Occlumency [11]. While channel partitioning splits the input by channels, *orthogonal* to the depth direction, *y-plane partitioning* uses planes *parallel* to the depth direction; Figure 1 provides a visual representation of y-plane partitioning.

At a high level, both schemes first split the input into partitions and compute the contribution of that partition to the output by *(i)* applying im2col on each partition, *(ii)* computing a matrix-matrix multiplication with the corresponding subset of the weight matrix, and *(iii)* adding the result to the output buffer. Y-plane and channel partitioning offer different design tradeoffs, with complementary strengths and weaknesses. In

		28MB+1vCPU		56MB+2vCPU		112MB+4vCPU		168MB+8vCPU	
Layer	**Input (MB)**	**Latency (s)**	**Evictions (#)**	**Latency (s)**	**Evictions (#)**	**Latency (s)**	**Evictions (#)**	**Latency (s)**	**Evictions (#)**
Outside TEE									
2	123	0.547	-	0.340	-	0.219	-	0.168	-
5	61	0.455	-	0.274	-	0.152	-	0.106	-
8	31	0.416	-	0.237	-	0.127	-	0.085	-
Partitioning in TEE									
2	123	0.399	8,270	0.223	3,503	0.162	3,251	0.153	3,184
5	61	0.296	1,999	0.177	1,716	0.120	1,718	0.101	1,712
8	31	0.292	1,374	0.172	1,364	0.102	1,362	0.084	1,360
Unmodified in TEE									
2	123	17.899	**1,859,815**	10.989	**1,881,017**	6.153	**1,858,259**	0.746	30,286
5	61	17.664	**1,838,563**	10.651	**1,848,772**	0.682	8,799	0.381	1,729
8	31	17.556	**1,827,409**	1.121	4,909	0.538	1,361	0.323	1,700

TABLE II: Latency and Page Evictions for Convolution Layers using Y-plane Partitioning. Only the three convolutional layers with the largest inputs are shown. Input size was measured after the im2col transformation.

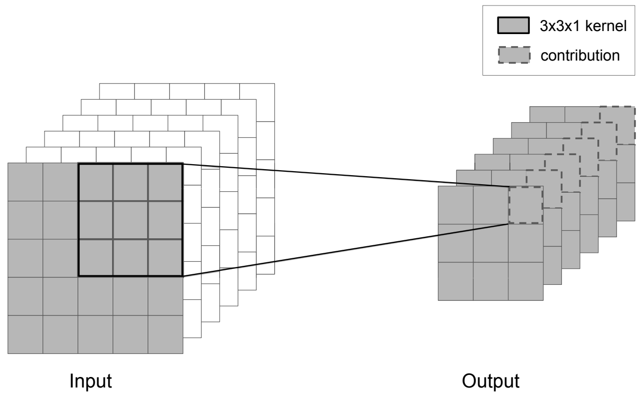

Fig. 2: Illustration of Channel Partitioning. This figure highlights the contribution of 1 input channel to 1 value on each output channel. Each input channel contributes to the entire output.

particular, we find that y-plane partitioning is more memory-efficient when the layer output is large, while channel partitioning is better when the weight matrix is large.

Y-Plane Partitioning. As illustrated in Figure 1, *y-planes* are the concatenation of one row from each channel of a 3D array. For this scheme, a partition is a group of contiguous y-planes; both the layer's inputs and outputs are logically divided into y-plane partitions. Each output y-plane is computed from a small and contiguous subset of the input y-planes. The convolution kernel size and stride determine the relationship between input and output y-planes.

Each round of computation involves three elements: *(i)* an output partition composed of contiguous y-planes, *(ii)* the corresponding subset of input y-planes, and *(iii)* the entire weight array. This repeated access to the entire weight array makes the weights size the limiting factor of y-plane partitioning.

Channel Partitioning. Channel partitioning, illustrated in Figure 2, divides the input into partitions of one or more channels [11], using a partition of the weights to calculate each contribution to the output. Note that the output is not partitioned and needs to be accessed during every round of computation to add the input-weight partition pairs' contribution. Thus, the output size is the limiting factor.

In practice, deep neural networks contain many convolutional layers, and the output and weight sizes of each layer vary. This observation, along with the aforementioned differences between y-plane and channel partitioning, suggests that a combination could yield the best results. Such a scheme would use the best partitioning scheme for the given layer. Further, the cost of switching from y-plane to channel partitioning (and vice versa) is negligible. We explore this idea in Section V-D.

C. Performance of Y-plane Partitioning

Table II illustrates the page thrashing bottleneck in convolutional layers, showing the impact of enclave size on latency and page evictions for the unmodified baseline running outside of the TEE, inside of the TEE, and inside of the TEE with y-plane partitioning. We make several observations that are consistent with prior work [11].

First, with y-plane partitioning, the convolutional layer latency decreased significantly and remained stable for all secure memory sizes. Second, page thrashing in Darknet was triggered when the size of the im2col-transformed input exceeded the enclave size—as measured by the drastic difference in page evictions. For example, layer 2, with its 123MB input, saw approximately 1.8 million page evictions for all three enclaves with less than 123MB of secure memory, but only 30 thousand evictions for the enclave with 168MB of memory.

Third, Darknet's per-layer latency varied dramatically, ranging from more than 17 seconds when thrashing occurred to less than 1 second when thrashing did not occur. As the total number of floating point operations remained constant, this difference resulted from thrashing.

D. Combining Y-Plane and Channel Partitioning

Different factors limit Y-Plane and Channel partitioning. Below we demonstrate those differences using a model with layer sizes that far exceed the available secure memory. We show that a combination of y-plane and channel partitioning

	Weights (MB)	Output (MB)	Y-Plane Latency (s)	Y-Plane Evictions (#)	Channel Latency (s)	Channel Evictions (#)
1	0.01	**49.44**	0.544	48,883	1.387	**123,145**
2	0.14	**49.44**	1.877	62,964	21.552	**1,809,386**
4	0.28	**24.72**	0.891	22,001	9.216	**868,540**
5	0.56	**24.72**	1.638	29,444	18.204	**1,716,377**
7	1.13	12.47	0.770	5,310	1.205	5,301
8	2.25	12.47	1.542	10,283	2.392	7,680
10	4.50	1.64	0.288	1,606	0.166	1,579
11	9.00	1.64	0.583	2,916	0.330	2,727
13	9.00	0.44	0.239	2,434	0.107	2,420
14	9.00	0.44	0.237	2,419	0.107	2,422
16	17.58	0.06	0.112	4,543	0.066	4,602
17	**34.33**	0.06	0.467	**35,689**	0.126	8,901
18	**68.66**	0.12	0.925	**70,979**	0.252	17,885
19	**68.66**	0.06	0.925	**70,878**	0.253	17,762

TABLE III: Per-Layer Latency and Page Evictions for VGG-Large.

allows us to execute this model without thrashing, whereas either scheme would fail if used in isolation.

Methodology. To scale up the model, we preserved most layer parameters (stride, padding, etc.), types, structure, and order of VGG-16. We only scale up two parameters: *(i)* the input resolution, which has an impact on the input and output size in all the layers, and *(ii)* the number of kernels in the first layer, which impacts the inputs, outputs, and weights size in all the layers. We chose an input resolution of 450×450 and 64 kernels in the first layer, so that some layers have either their output or weights larger than the enclave size. We call this model *VGG-Large*.

Results. Table III shows the per-layer page evictions and inference latency for y-plane partitioning and channel partitioning. We make four observations of these results. First, when the output is large compared to the secure memory size, as is the case in the first few layers, channel partitioning will start thrashing. In contrast, y-plane partitioning divides the output and, consequently, saw up to 58X fewer page evictions than channel partitioning.

Second, in the last few layers the weights are larger than secure memory, and y-plane partitioning shows up to 4.0X more page evictions than channel partitioning. This behavior is expected as y-plane partitioning does not divide the weights, but channel partitioning does.

Third, each scheme out-performed the other for a subset of the layers. In other words, using y-plane and channel partitioning in conjunction allows for efficient computations for models that neither y-plane nor channel partitioning could handle without page thrashing. To completely avoid page thrashing with VGG-Large, an enclave of at least 68 MB (resp. 50 MB) would be needed to run this model with y-plane-only (resp. channel-only) partitioning; while the hybrid scheme can run it with just 28MB. This experiment also shows that a large model can be ran with a significantly reduced memory footprint even if it can fit in memory. This result is useful in practice as systems that provide concurrency for secure deep learning inference, like Vessels [9], are limited by the memory footprint of individual models. Therefore, this hybrid scheme is likely to allow for greater concurrency, enabling more models to share the available secure memory efficiently. Of course, our observations are incomplete, and it is essential to consider other factors, such as the specifics of the target model and other potential sources of concurrency-based contention. We leave such explorations for future work.

Lastly, for the layers that can fit both the output and weights in secure memory, channel and y-plane partitioning are comparable in terms of latency and number of page evictions. Further experiments showed that, for these intermediate layers, the slight difference between both schemes is due to the GEMM (matrix multiplication) implementation. When using standard GEMM libraries such as OpenBLAS [17], this difference disappeared. Thus, we do not claim that one scheme is superior to the other for layers that fit in secure memory.

VI. The Decryption Bottleneck

Partitioning alleviates the page thrashing bottleneck. Without thrashing, the transfer of model parameters into the enclave becomes the dominating performance factor due to the overhead of page decryption and integrity checking. This issue, which we call the *decryption bottleneck*, is especially problematic for fully-connected layers with large weight matrices. We explore quantization and compression as possible solutions, reducing the number of pages that need to be transferred.

A. Characterization

For ease of exposition, we refer to the collection of components that handle secure paging—i.e., the eviction, encryption/decryption, and integrity checking of pages—as the *decryption link*.

In fully-connected layers, loading the weights into secure memory is expensive. For example, in our experiments, we observed that the first fully-connected layer of VGG-16 (i.e., layer 19) took 0.028 seconds to execute with Darknet normally, but 1.131 seconds ($\sim$40X) to execute with Darknet when run inside a trusted execution environment. Our experiments show that the difference in execution time was due entirely to the additional 1.102 seconds needed for loading in the weights from main memory—the 0.028 seconds needed for the layer computations was trivial by comparison.

Assuming we cannot modify the hardware to improve secure paging performance, and because the decryption link is already saturated, we turn toward techniques to *reduce the amount of data* that must be transferred over that link. Specifically, we analyze the use of two techniques, quantization and compression, to reduce the size of the weights for fully-connected layers. Further, as these techniques require additional computation, multi-threading can be used to keep the decryption link saturated.

Quantization is the process of converting the set of possible weight values (e.g., 32-bit floats) into a smaller discrete set of values (e.g., 16-bit floats). Some information is lost in this conversion, potentially affecting the model's accuracy, but

Layer	Input (MB)	28MB+1vCPU Latency (s)	28MB+1vCPU Evictions (#)	56MB+2vCPU Latency (s)	56MB+2vCPU Evictions (#)	112MB+4vCPU Latency (s)	112MB+4vCPU Evictions (#)	168MB+8vCPU Latency (s)	168MB+8vCPU Evictions (#)
Outside TEE									
19	392	0.030	-	0.029	-	0.030	-	0.029	-
21	64	0.005	-	0.005	-	0.005	-	0.005	-
23	16	0.001	-	0.001	-	0.001	-	0.001	-
Quant./Comp. in TEE									
19	392	0.542	55,707	0.498	55,258	0.497	16,167	0.464	16,168
21	64	0.090	9,194	0.094	11,157	0.076	2,660	0.075	2,660
23	16	0.023	2,090	0.020	2,500	0.020	715	0.022	715
Unmodified in TEE									
19	392	0.987	101,113	1.158	102,328	1.213	101,270	1.124	103,536
21	64	0.162	16,481	0.191	16,516	0.203	16,458	0.185	16,523
23	16	0.039	4,090	0.046	4,165	0.050	4,030	0.045	4,041

TABLE IV: Latency for Fully-Connected Layers using Quantization and Compression. The compression and quantization ratios are respectively 32:5 and 32:16. Numbers averaged over 30 runs.

			Compression Ratio				
Accuracy	Base.	Quant.	32:10	32:5	32:4	32:3	32:2
Top-1 (%)	70.4	70.4	70.4	70.2	68.1	68.5	26.7
Top-5 (%)	89.8	89.8	89.8	89.8	89.1	89.0	53.6

TABLE V: Model Accuracy with Quantization and Compression. A compression ratio of 32:10 means that a buffer of 32 bytes is compressed into 10 bytes. We omit ratios from 32:9 to 32:6 as they produced the same results as ratio 32:10.

the total memory requirements are halved. The weights are stored quantized and are converted back to 32-bit floats once decrypted. The cost of converting the values back to 32-bits floats was negligible in our experiments.

Similarly, compression also reduces data transfer requirements. We only consider lossy compression here as the compression factor for lossless compression was too small to be useful in our experiments. The amount of information lost is directly related to the compression factor, which can be tuned for many compression algorithms. The computational cost of decompression is higher than quantization, but the workload can be split more easily between virtual CPUs.

B. *Performance of Quantization and Compression*

Table IV shows the execution latency for fully-connected layers. For the 28MB+1vCPU and 56MB+2vCPU enclave configurations, we observe roughly half as many page evictions as unmodified Darknet, and execution took roughly half of the time. This performance difference is due to the quantization scheme, which halves the size of the weight matrix. In separate experiments, we observed that adding more than two threads failed to yield further improvement for quantization, suggesting that two threads are sufficient to saturate the decryption link.

Compression benefits more than quantization from the larger number of virtual CPUs offered by the 112MB+4vCPU and 168MB+8vCPU configurations. When using compression, the number of page evictions decreased to roughly 16% of unmodified Darknet. Once the decryption link was saturated with 6 threads, the compression scheme proved more efficient than quantization in these enclaves.

Lastly, we observe no drop in accuracy from quantization, as shown in Table V. Results will vary by model, and the impact of quantization on accuracy is an active area of research in the AI community [4], [10], [18]. More aggressive quantization strategies could yield even higher performance. For compression in all but the most extreme compression rate, the top-1 accuracy was within 2% of baseline and the top-5 accuracy was within 0.8%.

VII. Conclusions

In summary, we studied the use of partitioning, quantization, and compression to improve the memory efficiency of deep learning inference in trusted execution environments. Partitioning addresses the page thrashing bottleneck, and a combination of the proposed y-plane partitioning scheme and channel partitioning allows for the smallest memory footprint. Quantization and compression reduce the impact of the decryption bottleneck with little impact on model accuracy.

The primary limitation of this study is the limited number of models we consider. While convolution and fully-connected layers are common to a wide range of deep learning models, the benefits of the aforementioned optimizations depend on model specifics. For example, partitioning will not reduce the inference latency for the layers that already fit in memory (e.g. in ResNet). Even so, partitioning allows for a configurable memory footprint. This configurability is especially important in the context of concurrent inference, i.e., multiple enclaves running on a single server. We believe a full study of partitioning and model concurrency is an interesting direction for future work.

It is also important to consider other hardware capabilities when configuring the optimizations, such as secure memory size and decryption speed. For example, one would need to adjust the size of each partition to ensure they fit within secure memory. In the current implementation, a partition can be as small as a single y-plane, which for VGG-16 is at most a few hundred kilobytes. As another example, one might also want to tune the compression and quantization factors based on the decryption speed and available CPU resources.

Finally, TEE-based model inference is a building block for more complex security and privacy guarantees. For example, we could extend TEE-based inference to protect user privacy by hiding the user's input from both the hardware owner and the model provider. Supporting this feature would require additional components, such as a secure communication channel between the TEE and the user to hide both the user's input and the inference results from the hardware provider. Again, we leave such efforts for future work.

References

[1] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *arXiv preprint arXiv:2005.14165*, 2020.

[2] P. J. Denning, "Thrashing: Its causes and prevention," in *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, ser. AFIPS '68 (Fall, part I). New York, NY, USA: Association for Computing Machinery, 1968, p. 915–922.

[3] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale," *arXiv preprint arXiv:2010.11929*, 2020.

[4] G. Dundar and K. Rose, "The effects of quantization on multilayer neural networks," *IEEE Transactions on Neural Networks*, 1995.

[5] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing, "Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy," in *International Conference on Machine Learning (ICML)*, 2016.

[6] J. Gordon, "Microsoft azure confidential computing with intel sgx," https://software.intel.com/en-us/blogs/2018/11/08/microsoft-azure-confidential-computing-with-intel-sgx, 2018.

[7] T. Hunt, C. Song, R. Shokri, V. Shmatikov, and E. Witchel, "Chiron: Privacy-preserving machine learning as a service," *arXiv preprint arXiv:1803.05961*, 2018.

[8] "Intel, Software Guard Extensions (SGX)," https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions.html, Intel Corp., 2015.

[9] K. Kim, C. H. Kim, J. J. Rhee, X. Yu, H. Chen, D. Tian, and B. Lee, "Vessels: efficient and scalable deep learning prediction on trusted processors," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 462–476.

[10] R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference: A whitepaper," *arXiv preprint arXiv:1806.08342*, 2018.

[11] T. Lee, Z. Lin, S. Pushp, C. Li, Y. Liu, Y. Lee, F. Xu, C. Xu, L. Zhang, and J. Song, "Occlumency: Privacy-preserving remote deep-learning inference using sgx," in *The 25th Annual International Conference on Mobile Computing and Networking (MobiCom)*, 2019.

[12] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. A. Popa, "Delphi: A cryptographic inference service for neural networks," in *29th USENIX Security Symposium (USENIX Security)*, 2020.

[13] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical black-box attacks against machine learning," in *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, 2017, pp. 506–519.

[14] I. Shumailov, Y. Zhao, D. Bates, N. Papernot, R. Mullins, and R. Anderson, "Sponge examples: Energy-latency attacks on neural networks," *arXiv preprint arXiv:2006.03463*, 2020.

[15] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[16] S. Tople, K. Grover, S. Shinde, R. Bhagwan, and R. Ramjee, "Privado: Practical and secure dnn inference," *arXiv preprint arXiv:1810.00602*, 2018.

[17] Z. Xianyi, W. Qian, and Z. Chothia, "Openblas," *URL: http://xianyi.github.io/OpenBLAS*, 2012.

[18] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen, "Incremental network quantization: Towards lossless cnns with low-precision weights," in *International Conference on Learning Representations (ICLR)*, 2017.