AsymML: An Asymmetric Decomposition Framework for Privacy-Preserving DNN Training and Inference

Yue Niu, Ramy E. Ali and Salman Avestimehr

ECE Department
University of Southern California (USC)
{yueniu, reali, avestime}@usc.edu

Abstract

Leveraging parallel hardware (e.g. GPUs) to conduct deep neural network (DNN) training/inference, though significantly speeds up the computations, raises several data privacy concerns. Trusted execution environments (TEEs) have emerged as a promising solution to enable privacy-preserving inference and training. TEEs, however, have limited memory and computation resources which renders it not comparable to untrusted parallel hardware in performance. To mitigate the trade-off between privacy and computing performance, we propose an *asymmetric* model decomposition framework, AsymML, to (1) accelerate training/inference using parallel hardware; and (2) preserve privacy using TEEs. By exploiting the low-rank characteristics in data and intermediate features, AsymML asymmetrically splits a DNN model into *trusted* and *untrusted* parts: the trusted part features privacy-sensitive data but incurs small compute/memory costs; while the untrusted part is computationally-intensive but not privacy-sensitive. Computing performance and privacy are guaranteed by respectively delegating the trusted and untrusted part to TEEs and GPUs. Furthermore, we present a theoretical rank bound analysis showing that low-rank characteristics are still preserved in intermediate features, which guarantees efficiency of AsymML. Extensive evaluations on DNN models shows that AsymML delivers $11.2\times$ speedup in inference, $7.6\times$ in training compared to the TEE-only executions.

1 Introduction

Deep neural networks (DNNs) have been acting as an essential building block in various applications such as computer vision (Simonyan and Zisserman 2014; He et al. 2016) and natural language processing (Devlin et al. 2018; Radford et al. 2018). Efficiently training a DNN model usually requires a large training dataset and sufficient computing resources. In many real applications, datasets are locally collected and not allowed to be publicly accessible, while model training/inference is usually offloaded to parallel hardware (e.g. GPUs, TPUs), or even distributed onto multiple computing nodes. Considering that the communication can be eavesdropped or the memory in parallel hardware can be accessed by third parties, such practice poses serious privacy concerns that the data, especially if it contains sensitive information, might be leaked to the public and misused.

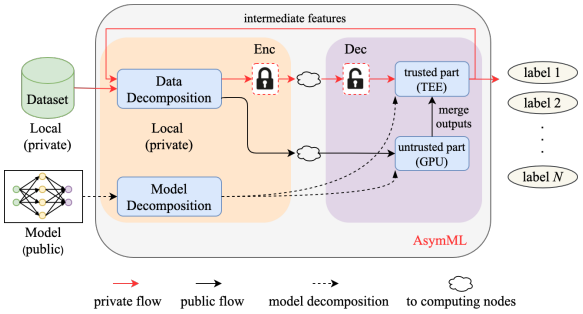


Figure 1: An Overview of AsymML is depicted. Models are asymmetrically decomposed into trusted and untrusted parts. The trusted part keeps most information at small compute/memory cost; while most computation is offloaded onto the untrusted part with little critical information involved.

The needs for data protection motivate various privacy-preserving machine learning algorithms, such as machine learning with differential privacy (DP) (Abadi et al. 2016; Papernot et al. 2018), federated learning (FL) (Konečnỳ, McMahan, and Ramage 2015), and hardware-based solutions (Costan and Devadas 2016; Tramer and Boneh 2018). DP based methods aim to defend membership inference and model-inversion attacks (Shokri et al. 2017; Fredrikson, Jha, and Ristenpart 2015) while the dataset or its noisy version (Wang et al. 2018; Mireshghallah et al. 2020) is still exposed during training. In FL, multiple clients/silos own their private datasets and aim to collaboratively train a global model without sharing their data. In this setting, the clients train and share their local models with a central server, which then obtains a global model by securely aggregating these local models (Bonawitz et al. 2017).

Trusted execution environments (TEEs), such as Intel Software Guard Extensions (SGX) (Costan and Devadas 2016) and Arm TrustZone(Alves 2004), provide a secure runtime environment that directly addresses such challenges in hardware. By performing all computations in TEEs, untrusted access to internal memory is forbidden. However, the computing performance is severely affected when TEEs are solely leveraged due to their limited computation capabilities as a result of not having a GPU support. Such executions are known as TEE-only executions. A natural solution for this problem is to leverage both TEEs and the untrusted

GPUs while protecting the privacy of the data. This idea was leveraged in (Tramer and Boneh 2018) to develop a privacy-preserving framework known as Slalom for DNN inference. Slalom, however, does not support DNN training, which is a more challenging and computationally-intensive task.

To efficiently perform private training and inference in a heterogeneous system with fast *untrusted* parallel hardware and *trusted* TEE-enabled processors, we proposes a novel training and inference framework, AsymML, as shown in Figure 1. By exploiting the low-rank characteristics in the inputs and the intermediate features, AsymML *asymmetrically* decomposes a model into trusted and untrusted parts. The trusted part handles the privacy-sensitive computations, but incurs small computation and memory costs. The untrusted part, on the other hand, is computationally-intensive but not privacy-sensitive. DNN model training and inference are performed by respectively delegating the trusted and untrusted part to TEEs and GPUs, where TEEs protect privacy, and GPUs guarantee computing performance. We also present a theoretical guarantee that states that the intermediate features in DNN models always have low-rank characteristics so that decoupling information from computation is possible. Furthermore, by masking the data in the untrusted part with a small noise, we also show that AsymML achieves more privacy guarantees without obvious accuracy loss. Specifically, our contributions are summarized as follows.

- We propose a privacy-preserving training and inference framework, AsymML, that decomposes a DNN model into two parts which effectively decouples information in data from computation through a lightweight singular value decomposition (SVD) algorithm.
- We provide a theoretical rank bound analysis on the data and the intermediate features that guarantees that the DNN models can be effectively decomposed in an asymmetric manner.
- We implement AsymML in a heterogeneous system with TEE-enabled CPUs and fast GPUs that supports various DNN models.

Our extensive experiments show that, during inference, AsymML delivers $7.7-11.2\times$ speedup in VGG16/VGG19, $5.8-7.2\times$ speedup in ResNet18/ResNet34; during training, AsymML achieves up to $7.6\times$ speedup in VGG variants and $5.8\times$ speedup in ResNet variants compared to TEE-only executions.

2 AsymML

In this section, we present AsymML. We start with our notations, then describe AsymML in detail and finally provide the overhead analysis of AsymML.

2.1 Notations

We use lowercase to denote scalars and vectors, and uppercase to denote matrices and tensors. $\overline{X}$ denotes a 2D matrix flattened from a multidimensional tensor X, while $\left\|\overline{X}\right\|_F$ denotes the Frobenius norm of the matrix $\overline{X}$. $\overline{X}^*$ denotes the transpose of $\overline{X}$. X_i denotes simplified i-th slice of a tensor $X \in \mathbb{R}^{N\times H\times W}$, $X_{i,:,:}$, while $X_{i,j}$ denotes simplified (i,j)-th slice, $X_{i,j,:}$. We use $\circledast$ to denote convolution, and $\cdot$ for matrix multiplication. For a DNN model, given a loss function $\mathcal{L}$, $\nabla_W \mathcal{L}$ denotes the gradients of the loss with respect to the parameter W. We use $\log$ to denote the logarithm to the base 2. Finally, $I(X;Y)$ denotes the mutual information between X and Y (Cover 1999).

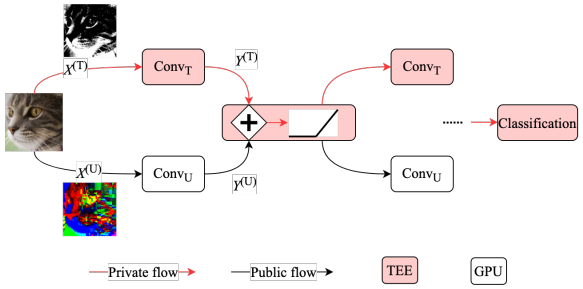


Figure 2: An illustration of the model decomposition in AsymML. The input to the convolutional layer X is decomposed into a privacy-sensitive part $X^{(\mathrm{T})}$ and a non-sensitive part $X^{(\mathrm{U})}$. The convolution of the trusted and the untrusted parts are then performed on the TEE and the GPU, respectively.

2.2 Asymmetric Model Decomposition using SVD

At a high level, AsymML starts with decomposing convolutional layers such that the computation involving sensitive information is performed in TEEs while the non-sensitive part is offloaded to GPUs. Specifically, the input of a convolutional layer, X, is decomposed into a privacy-sensitive part $X^{(\mathrm{T})}$ and a non-sensitive part $X^{(\mathrm{U})}$ as shown in Figure 2. When the convolutions in GPUs and TEEs are completed, the outputs in GPUs, $Y^{(\mathrm{U})}$, are merged with the outputs, $Y^{(\mathrm{T})}$, in TEEs, and then followed by a non-linear layer (and a pooling layer). The outputs of a non-linear layer will be then re-decomposed before proceeding to the next convolutional layer. During the whole forward/backward pass, the privacy-sensitive part in TEEs is never exposed to public environments, which effectively prevents crucial parts of data from being leaked.

For a **convolutional layer** with input $X \in \mathbb{R}^{N\times H\times W}$ and kernels $W \in \mathbb{R}^{M\times N\times k\times k}$ where N and M is the number of input and output channels, the i-th output channel is calculated as $Y_i = \mathrm{Conv}(X, W_i) = \sum_{j=1}^{N} X_j \circledast W_{i,j}$. [1] By splitting inputs X into $X^{(\mathrm{T})}$ and $X^{(\mathrm{U})}$, AsymML decomposes the convolution Conv into a trusted convolution $\mathrm{Conv_T}$ and an untrusted convolution $\mathrm{Conv_U}$ as follows

$$Y_i = \mathrm{Conv_T}(X^{(\mathrm{T})}, W_i) + \mathrm{Conv_U}(X^{(\mathrm{U})}, W_i). \qquad (1)$$

The *asymmetric* decomposition is inspired by the observation that the data and the intermediate features X usually feature a low-rank structure. By exploiting such a low-rank structure, X can be decomposed in a way that a low-rank tensor $X^{(\mathrm{T})}$ keeps most information, while a high-rank tensor $X^{(\mathrm{U})}$ stores the privacy-insensitive residuals. Therefore, complexity of $\mathrm{Conv_T}$ is significantly reduced without sacrificing privacy.

[1] The batch size and the bias are omitted here for simplicity.

To extract a privacy-sensitive low-rank tensor $X^{(\mathrm{T})}$, we first apply singular value decomposition (SVD) to $\overline{X} \in \mathbb{R}^{N \times HW}$ flattened from X as

$$\overline{X} = U \cdot \mathrm{diag}(\boldsymbol{s}) \cdot V^*, \tag{2}$$

where the *transformed* channels are stored in V, while the singular values in $\boldsymbol{s}$ denotes importance for each channels. The j-th channel in $X^{(\mathrm{T})}$ is obtained from the first r most important channels as

$$X^{(\mathrm{T})}_j = \sum_{j'=1}^{r} \boldsymbol{s}_{j'} \cdot U(j, j') \cdot X^{'}_{j'} \equiv \sum_{j'=1}^{r} a_{j,j'} \cdot X^{'}_{j'}, \tag{3}$$

where $X^{'}_{j'} \in \mathbb{R}^{H \times W}$ is a 2D matrix reshaped from the $j^{'}$-th column in V. On the other hand, $X^{(\mathrm{U})}_j$ is calculated as $X^{(\mathrm{U})}_j = X - X^{(\mathrm{T})} = \sum_{j'=r+1}^{N} a_{j,j'} \cdot X^{'}_{j'}$.

With the low-rank input $X^{(\mathrm{T})}$, the forward and the backward passes of $\mathrm{Conv}_{\mathrm{T}}$ can be reformulated in such a way that complexity depends on the number of principal channels r. During the forward pass, the outputs in TEEs are calculated as follows

$$\begin{aligned} Y^{(\mathrm{T})}_i &= \sum_{j=1}^{N} X^{(\mathrm{T})}_j \circledast W_{i,j} = \sum_{j=1}^{N} \sum_{j'=1}^{r} a_{j,j'} X^{'}_{j'} \circledast W_{i,j} \\ &= \sum_{j'=1}^{r} X^{'}_{j'} \circledast \sum_{j=1}^{N} a_{j,j'} W_{i,j} \equiv \sum_{j'=1}^{r} X^{'}_{j'} \circledast W^{'}_{i,j'}, \end{aligned} \tag{4}$$

where $W^{'}_{i,j'} = a_{j,j'} W_{i,j}$.

During the backward pass, given the gradient $\nabla_Y \mathcal{L}$, $\nabla^{(\mathrm{T})}_{W_{i,j}} \mathcal{L}$ from TEEs is calculated as

$$\nabla^{(\mathrm{T})}_{W_{i,j}} \mathcal{L} = X^{(\mathrm{T})}_j \circledast \nabla_{Y_i} \mathcal{L} = \sum_{j'=1}^{r} a_{j,j'} \cdot X^{'}_{j'} \circledast \nabla_{Y_i} \mathcal{L}. \tag{5}$$

On the other hand, the untrusted forward ($Y^{(\mathrm{U})}_i$) and the backward ($\nabla^{(\mathrm{U})}_{W_{i,j}} \mathcal{L}$) passes in GPUs are the same as in classic convolutional layers but with a different input $X^{(\mathrm{U})}$. The final result Y_i and $\nabla_{W_{i,j}} \mathcal{L}$ is obtained by simply adding results from TEEs and GPUs. In addition, since computing $\nabla_X \mathcal{L}$ does not involve the data X, it is safe to be offloaded onto GPUs (See more details in Appendix C).

2.3 Lightweight SVD Approximation

Performing exact SVD on X incurs a significant complexity of $O(NH^2W^2 + N^2HW)$, which goes against our objective of reducing the complexity in TEEs. To reduce the complexity, we propose a lightweight SVD approximation that reduces complexity to only $O(rNHW)$.

SVD essentially is an algorithm that finds two vectors $\boldsymbol{u}^{(i)}$ and $\boldsymbol{v}^{(i)}$ to minimize $\left\| \overline{X}^{(i-1)} - \boldsymbol{u}^{(i)} \cdot \boldsymbol{v}^{(i)*} \right\|_F$, where $\overline{X}^{(i-1)}$ is the remaining $\overline{X}$ with the $i-1$ most principal components extracted. $\left\{\boldsymbol{u}^{(i)}\right\}_{i=1}^{HW}$ and $\left\{\boldsymbol{v}^{(i)}\right\}_{i=1}^{N}$ are both orthogonal vectors. AsymML, however, does not require such orthogonality. The optimization then is formulated as

$$\begin{aligned} \overline{X}^{(\mathrm{T})} = \arg\min_{\overline{X}^{(\mathrm{T})}} \quad & \left\| \overline{X} - \overline{X}^{(\mathrm{T})} \right\|_F^2, \\ \text{s.t.} \quad & \mathrm{rank}(\overline{X}^{(\mathrm{T})}) \le r, \overline{X}^{(\mathrm{U})} = \overline{X} - \overline{X}^{(\mathrm{T})}. \end{aligned} \tag{6}$$

Using alternating optimization (Bezdek and Hathaway 2003), each component i in $\overline{X}^{(\mathrm{T})}$ can be obtained as in Algorithm 1. Due to the fast convergence of alternating optimization, given suitable initial values (e.g. output channels from the previous ReLU/Pooling layer), we have experimentally observed that the maximum number of iterations max_iter to reach the optimal $\left\{\boldsymbol{u}^{(i)}\right\}_{i=1}^{r}$ and $\left\{\boldsymbol{v}^{(i)}\right\}_{i=1}^{r}$ is typically $1 \sim 2$. Finally, it is worth noting that the computational complexity only increases linearly with r.

Algorithm 1: Lightweight SVD Approximation

Input: $r, \overline{X}, \left\{\boldsymbol{u}_0^{(i)}, \boldsymbol{v}_0^{(i)} \quad | \quad i = 1, \cdots, r\right\}, \mathrm{max\_iter}$

Output: $\overline{X}^{(\mathrm{T})}, \overline{X}^{(\mathrm{U})}$

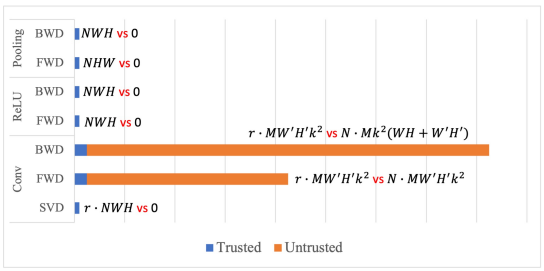
```
Initialize X̄^(T) as zero matrix
for i in 1, ..., r do
    for j in 1, ..., max_iter do
        u_j^(i) = X̄ · v_{j-1}^(i) / ||v_{j-1}^(i)||_F^2   {Alternating optimization}
        v_j^(i) = X̄* · u_j^(i) / ||u_j^(i)||_F^2
    end for
    X̄^(T) = X̄^(T) + u_j^(i) · v_j^(i)*
    X̄ = X̄ − u_j^(i) · v_j^(i)*
end for
X̄^(U) = X̄
```

In the actual implementation, $\overline{X}^{(\mathrm{T})}$ is stored as a list of vectors $\left\{\boldsymbol{u}^{(i)}, \boldsymbol{v}^{(i)} \quad | \quad i = 1, \cdots, r\right\}$, rather than as a matrix form. It should be noted that the approximated SVD in Algorithm 1 does not affect the training or the test accuracy as the sum of $X^{(\mathrm{T})}$ and $X^{(\mathrm{U})}$ always equals X. Hence, the sum of the convolution outputs $Y^{(\mathrm{T})} + Y^{(\mathrm{U})}$ also equals the original output Y, as given in Equation (1). In fact, the approximated SVD algorithm only affects the information distribution in $X^{(\mathrm{T})}$ and $X^{(\mathrm{U})}$, which is analyzed in Section 4.4

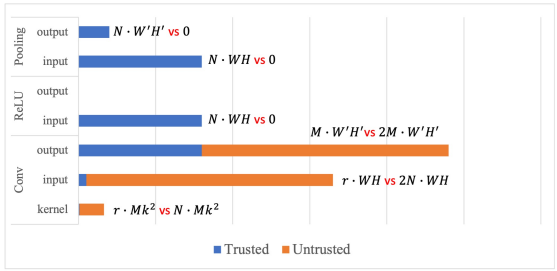
2.4 Overhead Analysis

The computation and the memory costs in TEEs are of great concern as they decide AsymML's performance in a heterogeneous system. In this section, we break down computations and memory usage in a heterogeneous system and compare the costs of the TEEs and the GPUs.

Figure 3 shows the computation and the memory costs in TEEs and GPUs for the case where $r/N = 16$. As described in Section 2.2, the computation complexity of $\mathrm{Conv}_{\mathrm{T}}$ in TEEs increases with the number of principal channels r, while the complexity of $\mathrm{Conv}_{\mathrm{U}}$ in GPUs is the same as the original convolution layer. In addition to $\mathrm{Conv}_{\mathrm{T}}$, all element-wise operations (ReLU, Pooling, etc) and the lightweight



(a) Computation in TEEs and GPUs



(b) Memory in TEEs and GPUs

Figure 3: The computation and memory costs in TEEs (Trusted) and GPUs (Untrusted) ($\frac{r}{N} = \frac{1}{16}$).

SVD are performed in TEEs. As for the memory cost, the inputs and the outputs of all element-wise operations are stored in TEEs, together with the inputs $X^{(\mathrm{T})}$ and the outputs $Y^{(\mathrm{T})}$ of the convolution $\mathrm{Conv}_{\mathrm{T}}$. While the GPU's memory is only used to store the inputs $X^{(\mathrm{U})}$ and the outputs $Y^{(\mathrm{U})}$ of the convolution $\mathrm{Conv}_{\mathrm{U}}$.

3 Rank Bound Analysis

The efficiency of AsymML hinges on the observation that the data and the intermediate features have a low-rank structure. Thus it is necessary to ensure that the low-rank structure is still preserved after DNN layers, especially for convolutional layers. In this section, we first define a metric termed as *SVD-channel entropy* to formally quantify such a low-rank structure. Then, we show how the SVD-channel entropy changes in a DNN model after each layer. This rank analysis serves as a guidance for AsymML to decompose the computation of each layer, however we formally measure the privacy guarantees of AsymML in Section 4.4 through the mutual information metric (Cover 1999).

3.1 SVD-Channel Entropy

Inspired by SVD entropy (Alter, Brown, and Botstein 2000) and Rényi Entropy (Jizba and Arimitsu 2004), we define SVD-channel entropy based on singular values obtained in Equation (2). Given the singular values $\{s_i(X)|i = 1, \cdots, N\}$ for input X with N channels, the SVD-channel entropy μ is defined as follows.

Definition 1. *(SVD-Channel Entropy). The SVD-channel entropy of a matrix X is given by $\mu_X = -\log\left(\sum_{j=1}^{N} \bar{s}_j^2(X)\right)$, where $\bar{s}_j$ is the j-th normalized singular value: $\bar{s}_j(X) = \frac{s_j(X)}{\sum_{j'=1}^{N} s_{j'}(X)}$.*

Lemma 1 and Theorem 1 show that the SVD-channel entropy can be related to how many *principal* channels are needed to represent the original data X.

Lemma 1. *The SVD-channel entropy of an input X with N input channels is bounded as $0 \le \mu_X \le \log N$.*

If we use the first $\lceil 2^{\mu_X} \rceil$ most principal channels to reconstruct X, then Theorem 1 states that such reconstruction is sufficient to preserve most information in X.

Theorem 1. *Given a matrix X with SVD-channel entropy μ_X and if we use the $N' = \lceil 2^{\mu_X} \rceil$ most principal channels to reconstruct X, then $\frac{\sum_{j=1}^{N'} s_j^2}{\sum_{j=1}^{N} s_j^2} \ge 0.97$.*

We provide the proofs of Lemma 1 and Theorem 1 in Appendix A. In the rest of the paper, we regard N' as the sufficient number of necessary channels to represent X.

3.2 SVD-Channel Entropy in DNNs

Given an input data with a low-rank structure, it is crucial to measure how such structure (SVD-channel entropy) changes after each layer so that we can systematically set the number of principal channels r to be used in TEEs. In convolutional neural networks (CNNs), Conv, Pooling and nonlinear layers such as ReLU play important roles in changing the rank structure of the data. Hence, we mainly analyze SVD-channel entropy after those layers.

For a **convolutional layer** with input $X \in R^{N\times H\times W}$, we first bound the SVD-channel entropy after convolution with 1×1 kernels in Theorem 2, then extend the theorem to $k\times k$ kernels in Theorem 3.

Theorem 2. *For a Conv layer, given input $X \in R^{N\times H\times W}$ with SVD-channel entropy μ_X, kernel $W \in R^{M\times N\times 1\times 1}$, then the SVD-channel entropy of the output $Y \in R^{M\times H'\times W'}$ is upper-bounded as follows*

$$\mu_Y \le \log\lceil 2^{\mu_X}\rceil.$$

Theorem 2 implies that low-rank structure is still preserved in the outputs for a convolutional layer with 1×1 kernels. Therefore, before a convolutional layer, if $\lceil 2^{\mu}\rceil$ principal channels are used in TEEs, the same number of channels is still sufficient to keep privacy in outputs.

For a convolutional layer with a $k \times k$ kernel, it can be viewed as an accumulation of 1×1 convolution with k^2 different "input channels". Each "input channel" is a patch of X_j, denoted as $\left\{\hat{X}_{j,q,r}|1 \le q, r \le k\right\}$, as shown in Figure 4. Given $X \in R^{N\times H\times W}$, $W \in R^{M\times N\times k\times k}$, the i-th output channel can be rewritten as follows

$$Y_i = \sum_{j=1}^{N} W_{i,j} \circledast X_j = \sum_{j=1}^{N} \sum_{q=1,r=1}^{k,k} W_{i,j,q,r} \circledast \hat{X}_{j,q,r}, \quad (7)$$

where $W_{i,j,q,r}$ is a 1×1 kernel after this conversion. Then, as stated in Theorem 3, knowing the SVD-channel entropy in

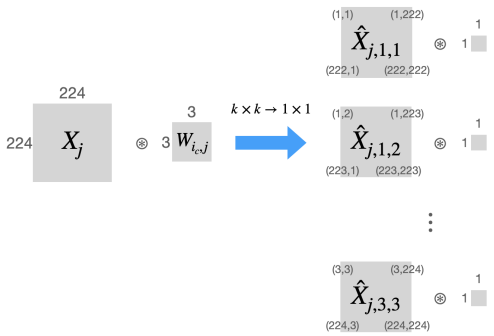


Figure 4: An illustration of the conversion from $k \times k$ convolution to 1×1 convolution.

$\hat{X}_{j,:,:}$ for $j = 1, \cdots, N$, the SVD-channel entropy in outputs of $k \times k$ convolution can be accordingly bounded.

Theorem 3. *Given data X with SVD-channel entropy μ_X, $N^{'} = \lceil 2^{\mu_X} \rceil$, $\mu_{\hat{X}_{j,:,:}} = \hat{\mu}_j$ for $1 \leq j \leq N$ and suppose that $\hat{\mu}_1 \leq \hat{\mu}_2 \leq \cdots \leq \hat{\mu}_N$, then the SVD-channel entropy of $\hat{X}$ satisfies*

$$\mu_{\hat{X}} \leq \log(\sum_{j=1}^{N^{'}} \lceil 2^{\hat{\mu}_j} \rceil) \cong \mu_X + \bar{\mu}, \text{ where } \bar{\mu} = \frac{\sum_{j=1}^{N^{'}} \hat{\mu}_j}{N^{'}}.$$

Furthermore, the SVD-channel entropy of the output Y after convolution with $W \in R^{M \times N \times k \times k}$ is upper-bounded as

$$\mu_Y \leq \log(\sum_{j=1}^{N^{'}} \lceil 2^{\hat{\mu}_j} \rceil)$$

The proofs of Theorem 2 and Theorem 3 are provided in Appendix A.

Based on Theorem 3, with $k \times k$ kernels, the SVD-channel entropy of the outputs increases with $\mu_{\hat{X}_{j,:,:}}$ for $j = 1, \cdots, N^{'}$. Intuitively, if k is larger, more patches are included, then $\mu_{\hat{X}_{j,:,:}}$ will be higher. Therefore, a large kernel size typically adds more entropy into outputs Y. For example, given 3×3 kernels, $\bar{\mu}$ is around 1, which indicates that $2\times$ number of principal channels are needed to represent Y. More numerical analyses are presented in Appendix B. Furthermore, it is straightforward to extend Theorem 2 and Theorem 3 to convolutional layers with a convolution stride larger than 1.

For **ReLU** and **Pooling layers**, we also provide rank analyses in the Appendix A showing that the same number of principal channels as in the input is sufficient to represent the output after ReLU and Pooling layers.

4 Empirical Analysis

In this section, we evaluate AsymML from the following three aspects: overall training/inference performance, runtime breakdown, and information leakage. We use VGG-16/19 and ResNet-18/34, and train and test these models on two datasets: CIFAR10 (Krizhevsky, Hinton et al. 2009) and ImageNet (Deng et al. 2009). We consider a heterogeneous system with an Intel SGX enabled Xeon E-2288G CPUs(Costan and Devadas 2016) as a TEE, while NVIDIA Quadro RTX5000 GPUs work as untrusted accelerators.

4.1 AsymML Implementation

Since there is no existing library that efficiently leverages both TEEs and untrusted GPUs, we first build a library for AsymML as shown in Figure 5. The library reads the model definition from PyTorch (Paszke et al. 2019), splits it and respectively offloads the computations onto TEEs and GPUs. Operations in TEEs (e.g. Conv_T, ReLU, Pooling) are supported by special functions in TEEs.

During training and inference, SGX and GPU contexts are created for trusted and untrusted operations, as described in Sec 2.2. We use PyTorch as an overall coordinator to distribute computations, activate GPU/SGX context, compute loss, and update model parameters (See FWD and BWD control in Figure 5). For convolutional layers, GPUs and TEEs work in parallel; for other operations, sequential execution is required.

During the forward pass, the outputs of a convolutional layer in GPUs and TEEs will be merged in ReLU (and Pooling) layer in TEEs, then followed by a light-weight SVD to decompose activation into $X^{(\text{T})}$ and $X^{(\text{U})}$, which is then fed into the next convolutional layer. During the backward pass, as described in Sec. 2.2, computing $\nabla_X \mathcal{L}$ for ReLU and Pooling layers is fully performed in TEEs, while $\nabla_X \mathcal{L}$ for convolutional layers is fully performed in GPUs. Partial $\nabla^{(\text{T})}_{W_{i,j}} \mathcal{L}$ and $\nabla^{(\text{U})}_{W_{i,j}} \mathcal{L}$ in a convolutional layer is calculated in GPU and SGX respectively, and merged in GPU context.

4.2 Training and Inference Performance

We conduct training and inference on ImageNet, and compare the runtime with three baselines: GPU-only, TEE-only, and Slalom (only in inference). Following the standard data pre-processing procedure, we resize each image into $3 \times 224 \times 224$. The batch size is set to 32. During training, we set the runtime as the average time of a single forward and backward pass, while during inference, we set it as the average time of a forward pass.

Based on Section 3.2, we compute the number of principal channels r in TEEs as follows: for the first convolutional layer, $r = 1$; r is doubled when another convolutional layer (VGG) or Residual block (ResNet) is added, while r remains unchanged when a ReLU or Pooling layer is added. Furthermore, r is at most 32 in all experiments, as it turns out that 32 principal channels are sufficient to keep most information in input data.

Due to the lack of any existing work that supports private DNN training in TEEs except for TEE-only implementation, we compare the runtime performance with the TEE-only and the GPU-only executions. Figure 6a shows the actual runtime (red plots) and the relative slowdowns (bar plots) compared to the GPU-only execution. Compared to the TEE-only method, AsymML achieves $7.5 - 7.6\times$ speedup on VGG-16/VGG-19, and up to $5.8-5.8\times$ speedup on ResNet-18/34. Compared to the GPU-only method, AsymML shows around $15\times$ slowdown, while TEE-only shows at least $91\times$ slowdown. The slowdown arises from the computation cost in TEEs, and the communication overhead between CPUs and GPUs, for which we present a breakdown analysis in Section 4.3.

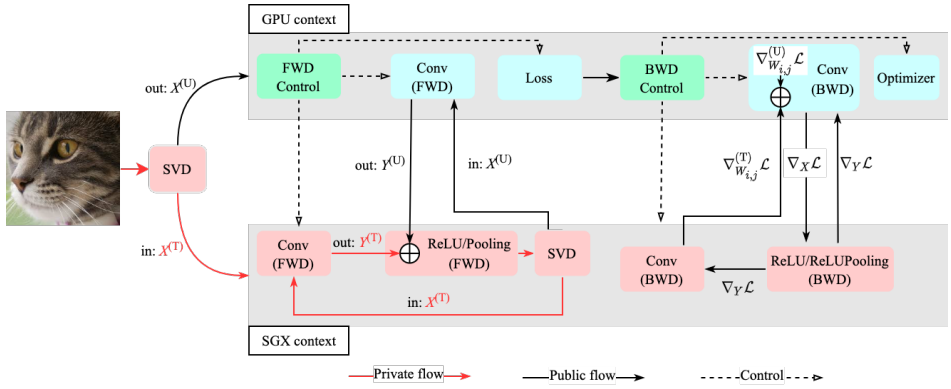
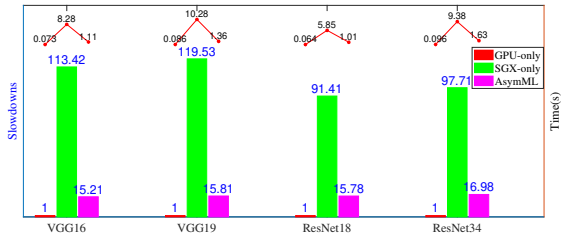
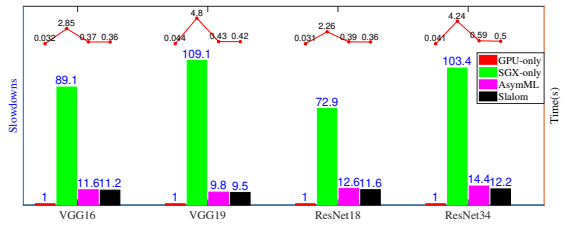


Figure 5: Implementation of AsymML in a heterogeneous system with SGX enabled CPUs and GPUs. The forward and the backward passes in convolutional layers are coordinated by FWD and BWD Control, respectively.



(a) The training time of AsymML compared to the SGX-only and the GPU-only executions is shown.



(b) The inference time of AsymML compared to the SGX-only execution, the GPU-only execution and Slalom is shown.

Figure 6: Training and Inference performance on ImageNet. Bar plot shows slowdowns compared to GPU-only execution; red dotted lines are the corresponding runtime.

For the performance of the inference, in addition to the TEE-only and the GPU-only baselines, we further compare with Slalom (Tramer and Boneh 2018). Figure 6b shows the actual inference time (red plots) and the relative slowdowns (bar plots). Since a small part of the convolution is performed in TEEs, inference using AsymML is slightly slower than Slalom, which is reasonable since Slalom only performs masking/unmasking operations in convolutional layers and non-linear operations. However, as AsymML effectively exploits low-rank data structure, AsymML still does not sacrifice too much performance. When compared to the TEE-only execution, AsymML achieves even more speedups than during training: $7.7 - 11.2\times$ faster on VGG-16/VGG-19, and $5.8 - 7.2\times$ faster on ResNet-18/ResNet-34

4.3 Runtime Breakdown

In this section, we profile the runtime of the forward and backward passes in AsymML. To better identify the main bottlenecks on a heterogeneous platform, we break down the runtime into computation and communication runtimes.

We use VGG16 as an example, as shown in Figure 8. Runtime time for other models are also provided in Appendix B.

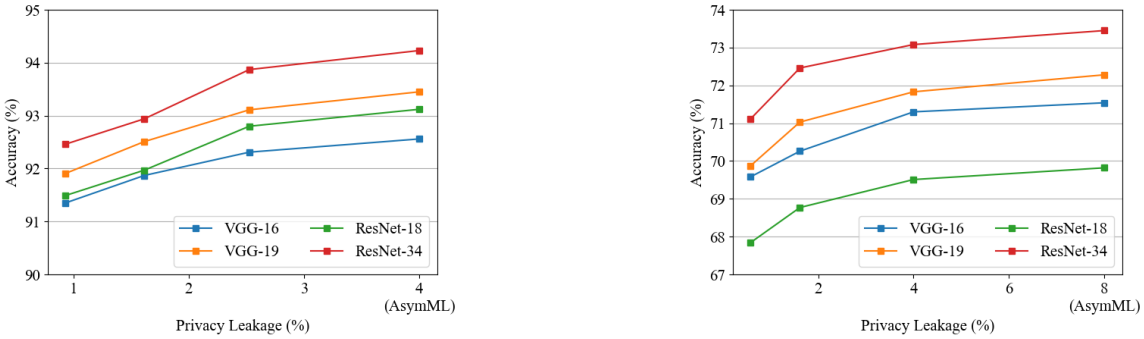
For the forward pass, the blue bars show the time spent in TEEs, while the red bars next to them show time for data movement between CPUs and GPUs. In the early convolutional layers, as the size of the features is large, the communication from GPUs to CPUs (merge output into TEEs) brings notable additional cost, which then becomes marginal as the model goes deeper. In addition, since there is no CPU-GPU communication in the ReLU and the Pooling layers during the forward pass, the additional cost in those layers is only caused by switching running context from GPU (convolution) to SGX context (ReLU, Pooling) (See Figure 5), which contributes little to the total running time.

For the backward pass, the green bars show the time in TEEs, and the red bars show CPU-GPU communication. The additional cost for CPU-GPU communication is much more dominant, especially in early convolutional layers. The reason is that during the backward pass, not only are input gradients $\nabla_X \mathcal{L}$ but also parameter gradients $\nabla_{W_{i,j}} \mathcal{L}$ transferred between CPUs and GPUs. As shown in Figure 5, $\nabla^{(T)}_{W_{i,j}} \mathcal{L}$ in the SGX enclaves is copied from CPUs and accumulated in GPUs, while $\nabla_X \mathcal{L}$ is copied to CPUs to feed backward on the ReLU and the Pooling layers, whose output gradients are then copied back to GPUs for backward on another convolutional layer.

4.4 Privacy Leakage

In AsymML, part of the data $X^{(U)}$ is still exposed in an untrusted GPU environment. While the data has a low-rank structure and $X^{(U)}$ contains much less information compared to $X^{(T)}$, it is still necessary to measure how much information is exposed. Furthermore, to achieve more privacy guarantee, we also mask $X^{(U)}$ using Gaussian noise and present a tradeoff between the privacy leakage and model accuracy.

We use the mutual information (Cover 1999) between X and $X^{(U)}$, denoted by $I(X; X^{(U)})$, as the metric to measure



(a) CIFAR10 (b) ImageNet

Figure 7: Trade-off between privacy leakage and model accuracy on CIFAR10 and ImageNet.

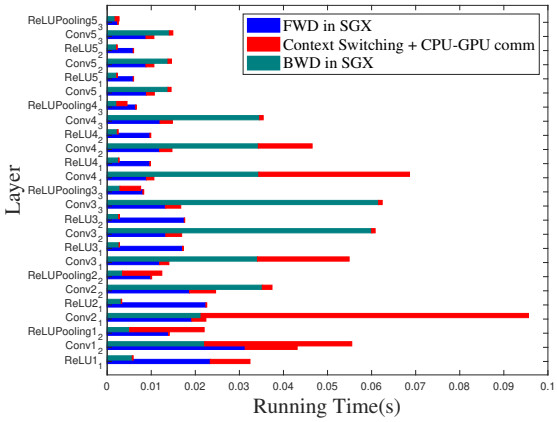


Figure 8: Runtime breakdown in VGG16. Time in data movement is relatively high in early convolutional layers, which then becomes marginal in later convolutional layers.

the privacy leakage in untrusted GPUs. As the probability distribution of X is unknown, we use a mutual information tool(Gao, Ver Steeg, and Galstyan 2015) to numerically estimate $I(X; X^{(\mathrm{U})})$. We use $X^{(\mathrm{U})}$ in the first convolutional layer as experiments show that it causes maximum information leakage compared to the later layers. Furthermore, $I(X; X^{(\mathrm{U})})$ is averaged across all samples in datasets.

Figure 7 shows the trade-off between the relative privacy leakage computed as $\frac{I(X;X^{(\mathrm{U})})}{I(X;X)}$ and the validation accuracy of the model. This privacy leakage is controlled by adding noise with noise-to-signal (nsr) ratio $[0.05, 0.1, 0.2]$. AsymML without noise added already provides strong privacy guarantees. For example, in CIFAR10, only 4% information about X is leaked, while only 8% information is leaked in ImageNet. With small noise added to $X^{(\mathrm{U})}$, privacy leakage is significantly reduced. For example, when nsr $= 0.05$, the privacy leakage is reduced by half, while less than 0.5% accuracy drop is incurred in all models.

5 Discussion

AsymML in this paper reduces the computation and memory costs in TEEs without significant privacy leaks. However, due to the additional communication overhead between GPUs and TEEs, the actual performance improvement does not exactly match the expectation in Figure 3. As Figure 8 shows, this additional overhead even dominates runtime in layers with large-size the features. Architecture such as Unified Memory Access (UMA)(Rogers and FELLOW 2013; Inc. 2021; Harris 2017) can potentially resolve this issue, which AsymML can be fitted into. On the other hand, there is yet no off-the-shelf tensor library designed for the SGX environment. The performance of AsymML can be further exploited with future support for tensor operations in SGX.

AsymML is designed to protect privacy under the assumption that TEEs are secure against potential attacks. The case of side channel attacks (Standaert 2010) might breach TEE's security (Chen et al. 2018) currently is not in our scope. AsymML is compatible with security updates in hardware such as Intel SGX (Costan and Devadas 2016), RISC-V Sanctum (Costan, Lebedev, and Devadas 2016).

6 Conclusion

In this paper, we have proposed an asymmetric decomposition framework, AsymML, to decompose DNN models and offload computations into trusted and untrusted fast hardware. The trusted part aim to preserve the sensitive information in the data with manageable computation cost and the untrusted part undertakes most computation. In such a way, AsymML makes the best use of each platform in a heterogeneous platform. We have then presented a solid analysis of channel entropy after each module in DNNs, which guarantees that AsymML effectively decouples sensitive information from computation, and outsources to the most suitable platform. Our extensive experiments show that AsymML achieves up to $11.2\times$ speedup in model inference and $7.6\times$ in training. We also show that AsymML provides strong privacy protection by measuring the mutual information between original data and data in untrusted GPUs.

References

Abadi, M.; Chu, A.; Goodfellow, I.; McMahan, H. B.; Mironov, I.; Talwar, K.; and Zhang, L. 2016. Deep learning with differential privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 308–318.

Alter, O.; Brown, P. O.; and Botstein, D. 2000. Singular value decomposition for genome-wide expression data processing and modeling. *Proceedings of the National Academy of Sciences*, 97(18): 10101–10106.

Alves, T. 2004. Trustzone: Integrated hardware and software security. *White paper*.

Bezdek, J. C.; and Hathaway, R. J. 2003. Convergence of alternating optimization. *Neural, Parallel & Scientific Computations*, 11(4): 351–368.

Bonawitz, K.; Ivanov, V.; Kreuter, B.; Marcedone, A.; McMahan, H. B.; Patel, S.; Ramage, D.; Segal, A.; and Seth, K. 2017. Practical secure aggregation for privacy-preserving machine learning. In *proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 1175–1191.

Chen, G.; Chen, S.; Xiao, Y.; Zhang, Y.; Lin, Z.; and Lai, T. H. 2018. Sgxpectre attacks: Leaking enclave secrets via speculative execution. *arXiv preprint arXiv:1802.09085*.

Costan, V.; and Devadas, S. 2016. Intel SGX Explained. *IACR Cryptol. ePrint Arch.*, 2016(86): 1–118.

Costan, V.; Lebedev, I.; and Devadas, S. 2016. Sanctum: Minimal hardware extensions for strong software isolation. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 857–874.

Cover, T. M. 1999. *Elements of information theory*. John Wiley & Sons.

Deng, J.; Dong, W.; Socher, R.; Li, L.-J.; Li, K.; and Fei-Fei, L. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, 248–255. Ieee.

Devlin, J.; Chang, M.-W.; Lee, K.; and Toutanova, K. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Fredrikson, M.; Jha, S.; and Ristenpart, T. 2015. Model inversion attacks that exploit confidence information and basic countermeasures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 1322–1333.

Gao, S.; Ver Steeg, G.; and Galstyan, A. 2015. Efficient estimation of mutual information for strongly dependent variables. In *Artificial intelligence and statistics*, 277–286. PMLR.

Harris, M. 2017. NVIDIA Unified Memory Description.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 770–778.

Inc., A. 2021. Apple M1 CPU Description.

Jizba, P.; and Arimitsu, T. 2004. Observability of Rényi's entropy. *Physical Review E*, 69(2): 026128.

Konečnỳ, J.; McMahan, B.; and Ramage, D. 2015. Federated optimization: Distributed optimization beyond the datacenter. *arXiv preprint arXiv:1511.03575*.

Krizhevsky, A.; Hinton, G.; et al. 2009. Learning multiple layers of features from tiny images.

Mireshghallah, F.; Taram, M.; Ramrakhyani, P.; Jalali, A.; Tullsen, D.; and Esmaeilzadeh, H. 2020. Shredder: Learning noise distributions to protect inference privacy. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 3–18.

Papernot, N.; Song, S.; Mironov, I.; Raghunathan, A.; Talwar, K.; and Erlingsson, U. 2018. Scalable Private Learning with PATE. In *International Conference on Learning Representations*.

Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; et al. 2019. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, 8026–8037.

Radford, A.; Narasimhan, K.; Salimans, T.; and Sutskever, I. 2018. Improving language understanding by generative pre-training.

Rogers, P.; and FELLOW, C. 2013. Amd heterogeneous uniform memory access. *AMD Whitepaper*.

Shokri, R.; Stronati, M.; Song, C.; and Shmatikov, V. 2017. Membership inference attacks against machine learning models. In *2017 IEEE Symposium on Security and Privacy (SP)*, 3–18. IEEE.

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

Standaert, F.-X. 2010. Introduction to side-channel attacks. In *Secure integrated circuits and systems*, 27–42. Springer.

Tramer, F.; and Boneh, D. 2018. Slalom: Fast, Verifiable and Private Execution of Neural Networks in Trusted Hardware. In *International Conference on Learning Representations*.

Wang, J.; Zhang, J.; Bao, W.; Zhu, X.; Cao, B.; and Yu, P. S. 2018. Not just privacy: Improving performance of private deep learning in mobile cloud. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2407–2416.

A Proofs, Lemmas and Theorems

This Appendix is organized as follows.

1. Section A.1 presents the proof of Lemma 1.
2. Section A.2 provides the proof of Theorem 1.
3. Section A.3 provides the proof of Theorem 3.
4.
5.
6.

A.1 Proof of Lemma 1

Proof. Let $Q = \sum_{i=1}^{N} s_i$ and $P = \sum_{i=1}^{N} s_i^2$, in which $s_1 \geq s_2 \geq \cdots \geq s_N$.

Since Q satisfies $\sqrt{P} \leq Q \leq \sqrt{N} \cdot \sqrt{P}$, where the left equality holds when all singular values are zeros except s_1 and the right equality holds when all singular values are equal. According to Definition 1, we have $\mu_X = -\log(\sum_{i=1}^{N} \bar{s}_i^2(X)) = -\log(\sum_{i=1}^{N} \frac{s_i^2}{Q^2}) = -\log(\sum_{i=1}^{N} s_i^2) + \log Q^2$

Hence, we get $\mu_X = 2\log Q - \log P$.

According to the inequality between Q and P, we have $2\log\sqrt{P} - \log P \leq \mu_X \leq 2\log\sqrt{NP} - \log P$. Therefore, $0 \leq \mu_X \leq \log N$. □

A.2 Proof of Theorem 1

Proof. Let $Q = \sum_{i=1}^{N} s_i$ and $P = \sum_{i=1}^{N} s_i^2$, and assume $s_1 > s_2 > \cdots > s_N$.

Taking $\{(i, s_i)\}_{i=1}^{N}$ as a series of points from a function, since s_i decreases with i, it is possible to find optimal parameters $(a > 0, 0 < b < 1)$ and a function $f(a, b) = a \cdot b^{i-1}$ to approximate $\{(i, s_i)\}_{i=1}^{N}$. Namely, $s_i \doteq a \cdot b^{i-1}$. Hence, we can write Q and P as: $Q = \sum_{i=1}^{N} s_i \doteq a \cdot \sum_{i=1}^{N} b^{i-1}$ and $P = \sum_{i=1}^{N} s_i^2 \doteq a^2 \cdot \sum_{i=1}^{N} b^{2(i-1)}$.

Summing up the sequences: $Q \doteq a \cdot \frac{1-b^N}{1-b}$, $P \doteq a^2 \cdot \frac{1-b^{2N}}{1-b^2}$, and $\frac{P}{Q^2} \doteq \frac{(1-b)(1+b^N)}{(1+b)(1-b^N)}$.

Let $\eta = \frac{\sum_{i=1}^{N'} s_i^2}{\sum_{j=1}^{N} s_j^2}$, according to the approximation above, η can be written as $\eta \doteq \frac{a^2 \cdot \frac{1-b^{2N'}}{1-b^2}}{a^2 \cdot \frac{1-b^{2N}}{1-b^2}} = \frac{1-b^{2N'}}{1-b^{2N}} \geq \frac{1-b^{2 \cdot 2\mu_X}}{1-b^{2N}} = \frac{1-b^{2\frac{Q^2}{P}}}{1-b^{2N}}$.

By Replacing $\frac{Q^2}{P}$ with b and N, we get $\eta \doteq \frac{1-b^{\frac{2(1-b^N)(1+b)}{(1+b^N)(1-b)}}}{1-b^{2N}}$. Finally by minimizing the function $\eta(b, N)$, we get a minimum of 0.97.

Therefore, with $N' = \lceil 2^{\mu_X} \rceil$ principal channels, the total energy in the reconstructed data is approximately at least 97% of the total energy in the original data X. □

In the later theorems and proofs, we regard N' as the sufficient number of principal channel to reconstruct X. And we use equality sign "=" rather than the approximated one "$\doteq$" when representing X using N' principal channels.

A.3 Proof of Theorem 2

Proof. Let $N' = \lceil 2^{\mu_X} \rceil$, then $X_j = \sum_{j'=1}^{N'} a_{j,j'} \cdot X'_{j'}$, where $X'_{j'}$ is the j'th principle channels of X, and $\overline{X'}_{j'}$ denote a flatten vector from $X'_{j'}$. Then, i-th output channel $Y_i = \sum_{j=1}^{N} W_{i,j} \circledast X_j = \sum_{j'=1}^{N'} (\sum_{j=1}^{N} W_{i,j} \cdot a_{j,j'}) \circledast X'_{j'}$. All channels in Y can be then written as follows

$$\begin{aligned} Y &= \{Y_1, \cdots, Y_M\} \\ &= \sum_{j'=1}^{N'} \left\{ (\sum_{j=1}^{N} W_{1,j} \cdot a_{j,j'}) \circledast X'_{j'}, \cdots \right\}. \end{aligned}$$

Let $Y'_{j'} =$

$\left\{ (\sum_{j=1}^{N} W_{1,j} \cdot a_{j,j'}) \circledast X'_{j'}, \cdots, (\sum_{j=1}^{N} W_{M,j} \cdot a_{j,j'}) \circledast X'_{j'} \right\}$, then $Y = \sum_{j'=1}^{N'} Y'_{j'}$. Since $W_{i,j}$ is a 1×1 kernel, $Y'_{j'}$ can be written as

$$Y'_{j'} = \left\{ (\sum_{j=1}^{N} W_{1,j} \cdot a_{j,j'}) \cdot X'_{j'}, \cdots, (\sum_{j=1}^{N} W_{M,j} \cdot a_{j,j'}) \cdot X'_{j'} \right\}.$$

For any two principal channels, $X^{'}_{j^{'}_1}, X^{'}_{j^{'}_2}, j^{'}_1 \neq j^{'}_2$, $\left\langle \overline{X^{'}}_{j^{'}_1}, \overline{X^{'}}_{j^{'}_2} \right\rangle = 0$. Therefore, $Y^{'}_{j^{'}_1}, Y^{'}_{j^{'}_2}$ are constructed by two orthogonal channels. Y only has at most $N^{'}$ principle channels. Therefore, $\mu_Y \leq \log N^{'} = \log \lceil 2^{\mu_X} \rceil$.

□

A.4 Proof of Theorem 3

In order to prove Theorem 3, we first present a lemma to bound channel entropy generated by a $k \times k$ kernel in one channel.

Lemma 2. *Given original input $X \in R^{N \times H \times W}$ with SVD-channel entropy μ_X, $k \times k$ kernels, then for $\forall 1 \leq q, r \leq k$, SVD-channel entropy in (q, r)-th patch of all channels, $\hat{X}_{:,q,r}$ satisfies:*

$$\mu_{\hat{X}_{:,q,r}} \leq \log \lceil 2^{\mu_X} \rceil$$

Proof. Flatten $\hat{X}_{:,q,r}$, X as $\overline{\hat{X}_{:,q,r}}$ and $\overline{X}$. Then, $\overline{\hat{X}_{:,q,r}}$ can be regarded as $\overline{X}$ with at most $k^2 - (\frac{k+1}{2})^2$ columns reset as zeros. We use set S_0 to denote these columns. Let $N^{'} = \log \lceil 2^{\mu_X} \rceil$, then each row in $\overline{X}$ can be written as $\overline{X_j} = \sum_{j^{'}=1}^{N^{'}} a_{j,j^{'}} \overline{X^{'}_{j^{'}}}$. Therefore, each row in $\overline{\hat{X}_{:,q,r}}$ can be written as $\overline{\hat{X}_{j,q,r}} = \sum_{j^{'}=1}^{N^{'}} a_{j,j^{'}} \overline{\hat{X}^{'}_{j^{'},q,r}}$, where $\overline{\hat{X}^{'}_{j^{'},q,r}}$ is the same as $\overline{X^{'}_{j^{'}}}$ except for values in columns S_0 are zeros. Therefore, $\overline{\hat{X}_{:,q,r}}$ has at most $N^{'}$ principle components. Namely, $\hat{X}_{:,q,r}$ has at most $N^{'}$ principle channels. Hence, $\mu_{\hat{X}_{:,q,r}} \leq N^{'} = \log \lceil 2^{\mu_X} \rceil$. □

With Lemma 2, we can prove Theorem 3 as follows:

Proof. According to Lemma 2, for the (q, r)-th patches in all channels, $\hat{X}_{:,q,r}, \forall 1 \leq q, r \leq k$ can be constructed by at most $N^{'}$ principle channels $\left\{\mathcal{U}_{1,q,r}, \cdots, \mathcal{U}_{N^{'},q,r}\right\}$ as follows

$$\begin{cases} \hat{X}_{1,q,r} &= a_{1,1,q,r}\mathcal{U}_{1,q,r} + \cdots + a_{1,N^{'},q,r}\mathcal{U}_{N^{'},q,r} \\ \quad \vdots & \\ \hat{X}_{N,q,r} &= a_{N,1,q,r}\mathcal{U}_{1,q,r} + \cdots + a_{N,N^{'},q,r}\mathcal{U}_{N^{'},q,r}. \end{cases} \tag{8}$$

For all patches in channel j, $\hat{X}_{j,:,:} \forall 1 \leq j \leq N$ can also be constructed by principle channels $\left\{\mathcal{V}_{j,1}, \cdots, \mathcal{V}_{j,\hat{N}_j}\right\}$, where $\hat{N}_j = \lceil 2^{\hat{\mu}_j} \rceil$:

$$\begin{cases} \hat{X}_{j,1,1} &= b_{j,1,1,1}\mathcal{V}_{j,1} + \cdots + b_{j,p_j,1,1}\mathcal{V}_{j,\hat{N}_j} \\ \quad \vdots & \\ \hat{X}_{j,k,k} &= b_{j,1,k,k}\mathcal{V}_{j,1} + \cdots + b_{j,p_j,k,k}\mathcal{V}_{j,\hat{N}_j}. \end{cases} \tag{9}$$

By combining equation (1) and (2), we can express $\hat{X}_{:,q,r}$ as follows

$$\begin{cases} \hat{X}_{1,q,r} & = a_{1,1,q,r}\mathcal{U}_{1,q,r} + \cdots + a_{1,N^{'},q,r}\mathcal{U}_{N^{'},q,r} \\ & \quad = b_{1,1,q,r}\mathcal{V}_{1,1} + \cdots + b_{1,p_j,q,r}\mathcal{V}_{1,\hat{N}_1} \\ \quad \vdots & \\ \hat{X}_{N,q,r} & = a_{N,1,q,r}\mathcal{U}_{1,q,r} + \cdots + a_{N,N^{'},q,r}\mathcal{U}_{N^{'},q,r} \\ & \quad = b_{N,1,q,r}\mathcal{V}_{N,1} + \cdots + b_{N,p_j,q,r}\mathcal{V}_{N,\hat{N}_N}. \end{cases} \tag{10}$$

In equation above, the coefficients are obtained from SVD, therefore, the determinant of any sub coefficient matrix is not zero. Hence, at most $N^{'}$ sub-equations are needed to derive $\mathcal{U}_{:,q,r}$ from $\mathcal{V}_{:,:}$.

If we pick the equations with least number of principal channels $\mathcal{V}$, then each channel in $\hat{X}$ can be fully constructed by the selected $\mathcal{V}$. At most, the total number of principal channels $\mathcal{V}$ needed is $\sum_{j=1}^{N^{'}} \hat{N}_j$. Then the total number of principle channels needed to construct $\hat{X}$ is at most $\sum_{j=1}^{N^{'}} \hat{N}_j$.

Therefore, $\mu_{\hat{X}} \leq \log(\sum_{j=1}^{N^{'}} \hat{N}_j) = \log(\sum_{j=1}^{N^{'}} \lceil 2^{\hat{\mu}_j} \rceil)$. When $\hat{\mu}_j$ are close, $\mu_{\hat{X}} \doteq \eta + \bar{\mu}$, where $\bar{\mu}$ is the average of for $\hat{\mu}_j$.

Finally, according to Theorem 2, $\mu_Y \leq \log \lceil 2^{\mu_{\hat{X}}} \rceil) \leq \log(\sum_{j=1}^{N^{'}} \lceil 2^{\hat{\mu}_j} \rceil)$. □

A.5 SVD-Channel Entropy after Average Pooling

For average Pooling layers, we also present a theorem to rigorously bound channel entropy in outputs.

Theorem 4. *For a average Pooling layer, given input $X \in R^{N\times H\times W}$ with SVD-channel entropy μ_X, SVD-channel entropy in output Y satisfies*

$$\mu_Y \leq \log \lceil 2^{\mu_X} \rceil$$

Proof. Let $N^{'} = \lceil 2^{\mu_X} \rceil$, then input $X_j = \sum_{j^{'}=1}^{N^{'}} a_{j,j^{'}} \cdot X^{'}_{j^{'}}$, where $X^{'}$ is the principle channels of X. With $k \times k$ average operator, we have

$$\begin{aligned}
&Y_i(h,w) \\
&= \frac{1}{k^2} \sum_{h^{'}=1}^{k} \sum_{w^{'}=1}^{k} X_i((h-1)k+h^{'},(w-1)k+w^{'}) \\
&= \frac{1}{k^2} \sum_{h^{'}=1}^{k} \sum_{w^{'}=1}^{k} \sum_{j^{'}=1}^{N^{'}} a_{i,j^{'}} X^{'}_{j^{'}}((h-1)k+h^{'},(w-1)k+w^{'}) \\
&= \sum_{j^{'}=1}^{N^{'}} a_{i,j^{'}} \cdot \frac{\sum_{h^{'}=1}^{k}\sum_{w^{'}=1}^{k} X^{'}_{j^{'}}((h-1)k+h^{'},(w-1)k+w^{'})}{k^2}
\end{aligned}$$

According to the equation above, each channel in Y can be seen as an accumulation of $N^{'}$ principle channels obtained by conducting average pooling on $X^{'}$. Therefore, Y can be constructed by at most $N^{'}$ principle channels.
Hence, $\mu_Y \leq \log N^{'} = \log \lceil 2^{\mu_X} \rceil$ □

A.6 Numerical Analysis on Nonlinear Operations

For a non-linear layer, ReLU is the most commonly used non-linear operator in DNNs. Theoretically bounding the SVD-channel entropy after ReLU is infeasible since it is a non-linear operator. In addition, the positions with negative value are randomly scattered. Alternatively, we conduct a thorough test on both random and well-trained models. We calculate SVD-channel entropy before and after ReLU layers, and average the difference between them. All tests show the difference is very small, meaning a ReLU layer preserve the low-rank structure in the features.

Another important module is a Pooling operation, which includes Max and Average Pooling. As in ReLU layers, Max Pooling is also non-linear. A similar test as in ReLU layers shows Max Pooling does not greatly change low-rank structure either. Besides, these two pooling operators usually deliver similar performance. As stated in Theorem 4, the SVD-channel entropy after a Average Pooling layer is also less than the one before Pooling. Therefore, Pooling layers still preserve the low-rank structure in the features.

B More Empirical Analysis

B.1 SVD-Channel Entropy Changes with Kernel Size

As Theorem 3 states, the upper bound of the SVD-channel entropy in the outputs only changes with the kernel size in the convolutional layers. Larger kernel size creates more patches in each channel (See Figure 4 in the paper), which generates more additional SVD-channel entropy along the patches. Theoretically bounding the entropy is hard and impractical. Therefore, in this section, we present a statistical result of SVD-channel entropy $\mu_{\hat{X}_{j,:,:}}$ with kernel sizes varying from 1 to 11.

In the experiment, the input data X is randomly sampled from ImageNet. As detailed in Sec 3.2, for $k \times k$ kernels, k^2 input patches are generated. We first obtain the singular values along all patches (each patch is seen as a channel) and compute the SVD-channel entropy according to Sec 3.1. Figure 9 shows the SVD-channel entropy distribution across 10K randomly sampled images using various kernels. The line plot indicates the mean value, while the violin plots visualizes the value distribution. With 3×3 kernels, SVD-channel entropy increases by around 1, which implies that given $N^{'}$ principal channels in the input, $2 \times N^{'}$ principal channels are sufficient to represent the output after convolution with 3×3 kernel. As the kernel size increases, more additional SVD-channel entropy will be introduced. Fortunately, a small kernel size is preferred in typical models as VGG and ResNet. Therefore, the SVD-channel entropy increment is well managed.

B.2 Runtime Breakdown on More Models

The runtime breakdown for VGG19, ResNet18 and ResNet34 are shown in Figure 11. They also share similar observations as in VGG16. The communication costs of the forward and the backward passes are relatively large in the early convolutional layers.

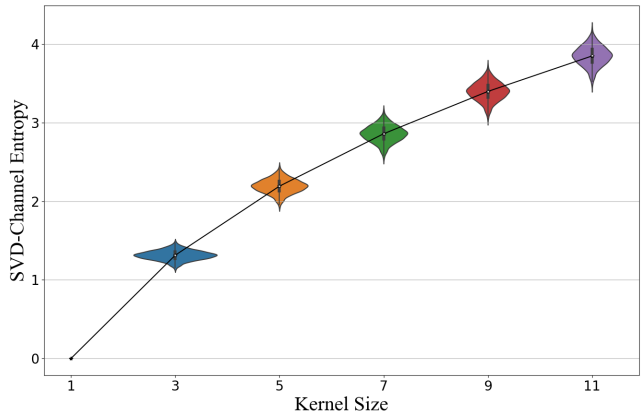


Figure 9: SVD-channel entropy in patches created from various kernel size. Line plot shows the mean value, while violin plot shows value distribution. The width of violin indicates the frequency of images around a particular entropy level.

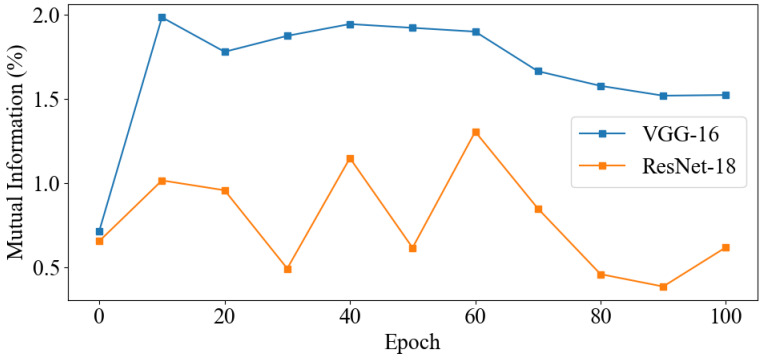
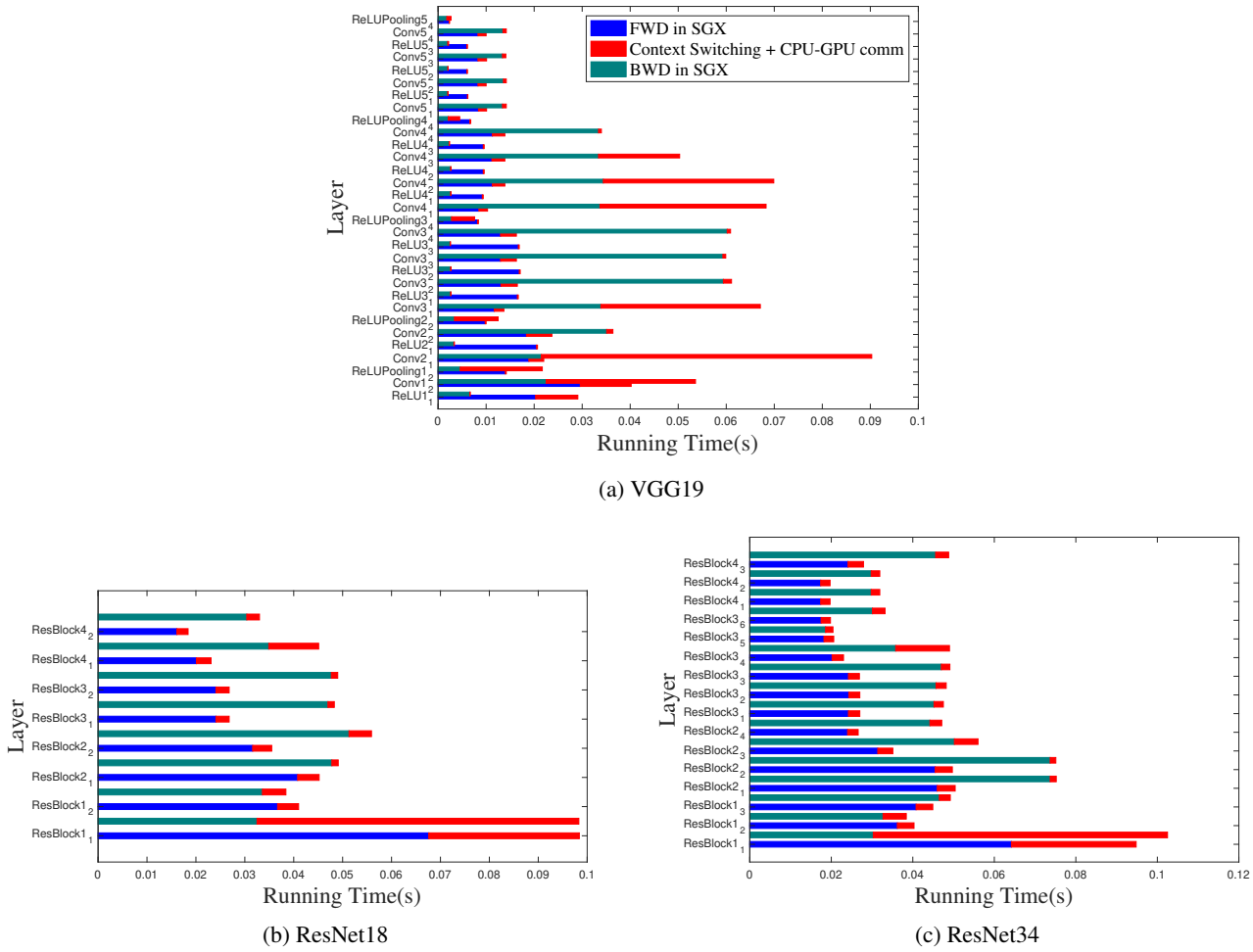


Figure 10: The relative mutual information between X and $\nabla_X \mathcal{L}$ is shown. The relative mutual information is less than 2%, which confirms that $\nabla_X \mathcal{L}$ can be safely offloaded onto untrusted GPUs.

B.3 Information Leakage via Gradients

As stated in Sec 2.2, computing $\nabla_X \mathcal{L}$ is offloaded onto GPUs since it does not directly input data X. However, it is still necessary to conduct a numerical analysis on the information leaked through $\nabla_X \mathcal{L}$. Similar to Section 4.4 in the paper, we use the mutual information between X and $\nabla_X \mathcal{L}$, denoted by $I(X; \nabla_X \mathcal{L})$ to investigate the leakage in $\nabla_X \mathcal{L}$.

We train VGG-16 and ResNet-18 on CIFAR10, and calculate $I(X; \nabla_X \mathcal{L})$ for every 10 epochs. $I(X; \nabla_X \mathcal{L})$ is averaged across all data samples. Figure 10 shows relative mutual information $\frac{I(X;\nabla_X \mathcal{L})}{I(X;X)}$ between the input data X and $\nabla_X \mathcal{L}$ in the first convolution layer. We choose the first convolutional layer as our experiments show that the first convolutional layer gives the largest $I(X; \nabla_X \mathcal{L})$. According to Figure 10, $I(X; \nabla_X \mathcal{L})$ is very small (less than 2%). Therefore $\nabla_X \mathcal{L}$ barely leaks information about X, which presents strong evidence that computing $\nabla_X \mathcal{L}$ can be safely offloaded onto untrusted GPUs.



(a) VGG19

(b) ResNet18

(c) ResNet34

Figure 11: Runtime breakdown in VGG19, ResNet18, and ResNet34. Layers in ResNet18 and ResNet34 are grouped as residual blocks (ResBlock).