DeepMon: Mobile GPU-based Deep Learning Framework for Continuous Vision Applications

Loc N. Huynh, Youngki Lee, Rajesh Krishna Balan
Singapore Management University
{nlhuynh.2014, youngkilee, rajesh}@smu.edu.sg

ABSTRACT

The rapid emergence of head-mounted devices such as the Microsoft Holo-lens enables a wide variety of continuous vision applications. Such applications often adopt deep-learning algorithms such as CNN and RNN to extract rich contextual information from the first-person-view video streams. Despite the high accuracy, use of deep learning algorithms in mobile devices raises critical challenges, i.e., high processing latency and power consumption. In this paper, we propose DeepMon, a mobile deep learning inference system to run a variety of deep learning inferences purely on a mobile device in a fast and energy-efficient manner. For this, we designed a suite of optimization techniques to efficiently offload convolutional layers to mobile GPUs and accelerate the processing; note that the convolutional layers are the common performance bottleneck of many deep learning models. Our experimental results show that DeepMon can classify an image over the VGG-VeryDeep-16 deep learning model in 644ms on Samsung Galaxy S7, taking an important step towards continuous vision without imposing any privacy concerns nor networking cost.

Keywords

Mobile GPU; Mobile Sensing; Deep Learning; Continuous Vision

1. INTRODUCTION

The popularity of head-mounted augmented reality (AR) devices such as the Microsoft Hololens [4] and the Google Glass [3] has given rise to a new class of continuous mobile vision applications. These range from identifying road signs in real time to provide directions [15], to identifying people in the environment to give guidance to individuals suffering from dementia [12]. In all these use cases, the commonality is the need to perform computer vision algorithms in real time on a continuous video stream provided by the AR devices.

The current state-of-the-art approach to continuous video processing is to use a deep neural network (DNN) approach where the video streams are processed by a large and well-trained convolutional neural network (CNN) or recurrent neural network (RNN). However, these networks require large amounts of CPU and memory resources to run efficiently. It has thus proved challenging to achieve adequate performance when executing large deep learning networks on commodity mobile devices. For example, a commonly used object recognition model, VGG-Verydeep-16 [46], has 13 convolutional layers and three fully connected layers and takes ≈100 seconds to process a single image using CPU on a Samsung Galaxy S7 smartphone.

One way to overcome this limitation is to use cloud resources to run the required DNNs [25]. However, this introduces significant privacy concerns (as the video feed is now available on the cloud server) in addition to possible latency, and energy concerns depending on where the cloud is located and what network interface (LTE etc.) is used.

In this paper, we present a system, called *DeepMon*[1], that uses the graphics processing unit (GPU) on mobile devices to execute the large DNNs required for continuous video processing. *DeepMon* can achieve continuous video processing (at about 1-2 frames per second) of full HD (1080p) video frames using just the memory, CPU, and GPU resources of a commodity smartphone. This speedup allows *DeepMon* to be used, with a larger processing pipeline where *DeepMon* can extract features from video frames that can then be processed by cloud resources to produce a complete knowledge. This greatly reduces the privacy impact of using a cloud (as only features and not actual video frames are sent to the cloud) as well as the latency and energy concerns (the feature set is much smaller than the full video image). However, in this paper, we focus solely on the optimisations and techniques to reduce the local processing time from multiple seconds to ≈600ms per frame and leave the integration with a complete cloud-enabled solution to future work.

Before building *DeepMon*, we analysed various deep learning models (e.g., VGG-Verydeep-16 [46] and YOLO [43]) to identify their performance bottlenecks. We noticed that they commonly adopt a large number of convolutional layers (to extract and refine features) along with a small number of fully connected layers (to make inferences). Our measurement showed that the convolutional layer processing takes a significant portion of the entire processing – e.g. 88.7% for VGG-Verydeep-16 and 85% for YOLO (see Section 4).

We thus focused on techniques to reduce the processing latency of convolutional layers. One clear solution was to offload the DNN convolutional layer computation to the mobile GPU as these layers have highly parallel and repetitive processing structures. However, prior offloading techniques were developed for server-class GPUs and required re-design/optimization for mobile GPUs with much

[1]The git repo of *DeepMon* is available at https://github.com/JC1DA/DeepMon and the videos are at http://is.gd/DeepMon

MobiSys '17, June 19–23, 2017, Niagara Falls, NY, USA.

DOI: http://dx.doi.org/10.1145/3081333.3081360

smaller number of processing cores and memory bandwidth; for instance, NVidia GTX 980 GPU for desktops have 2,048 GPU cores and 224GB/s memory read/write bandwidth while Mali T880 GPU on Samsung Galaxy S7 has 12 GPU cores and 25.6GB/s memory bandwidth.

We developed a suite of optimizations for processing convolutional layers on mobile GPUs. First, we designed a smart caching mechanism specially designed for convolutional layers. The key idea is to exploit the similarity between consecutive frames in first-person-view videos. Our mechanism is unique in that it utilizes the internal processing structure of convolutional layers to reuse the intermediate results of the previous frame to calculate the current frame, instead of just simply reusing its final output. Second, we decompose the matrices used in the convolutional layers to accelerate the multiplications between high-dimensional matrices, which are the bottleneck when running convolutional layers on GPUs. Also, we applied a number of system-level optimizations (described in Section 6 to accelerate the matrix calculation in mobile GPUs).

We implemented *DeepMon* using OpenCL [7] and Vulkan [9] and tested it on various mobile GPUs (Adreno 420, Adreno 430, and Mali T 880) with multiple large DNN models. For developers to adopt various DNN models in *DeepMon*, we also developed a tool that automatically converts pre-trained legacy models and loads them to *DeepMon* with its various optimization strategies applied.

Our results show that *DeepMon* significantly accelerates the processing of large DNNs. For example, the latency of VGG-VeryDeep-16 model-based inference reduces ≈5 times compared to the naive GPU-based implementation with just a marginal reduction in inference accuracy (≈5%). This enables low-latency image classifications (i.e., 3 frames per 2 seconds). Note: VGG-Verydeep-16 is the model many applications such as face recognition (Deep Face from Oxford [42]) and object detection (YOLO [43] and Fast R-CNN [22]) rely on. In addition, we conducted experiments on other models for object detection (such as YOLO) on commodity smartphones (Samsung Galaxy S7, Note 4, etc.). Our results showed that our proposed techniques could achieve a latency of 644ms for VGG-Verydeep-16 and 1,006ms for YOLO on Samsung Galaxy S7.

The contributions of our paper can be summarized as follows:

- To the best of our knowledge, *DeepMon* is the first system to allow large DNNs to run on commodity mobile devices at a low latency. Prior work, such as DeepX [35] and MCDNN [25], has focused on smaller DNNs, cloud computation, and non-commodity more powerful mobile devices such as the Tegra K1.
- We devised a suite of optimization techniques to reduce the processing latency of the convolutional layers of DNNs. Our smart caching mechanism leverages similarities of consecutive images to cache internally processed data within the deep convolutional neural network. Also, we adopted and improved state-of-the-art matrix multiplication techniques such as model decomposition [31] and unfolding [14] to accelerate multiplication operations (the bottleneck operation in convolutional layers) on mobile GPUs.
- We shared lessons about implementing *DeepMon* on OpenCL and Vulkan and scaling it to support various mobile GPUs. Prior work has focused primarily on CUDA [6] which, to the best of our knowledge, is not supported by commodity smartphones. *DeepMon's* OpenCL implementation can be deployed on a variety of Android-based devices with Snapdragon and Exynos chipsets while its Vulkan implementation (the first such implementation we could find) can be deployed on recent iPhone models such as the iPhone 7. Finally, developers can easily load pre-trained legacy models on various mobile GPUs by using *DeepMon's* model converting tool.
- We conducted extensive experiments showing that *DeepMon* can execute very deep models such as VGG-Verydeep-16 on video streams in near real-time, reducing the processing latency to execute one frame from 3 seconds down to 644 ms.

2. RELATED WORK

Deep Learning on Mobile Devices. Lane et al. took important first steps towards the real-time execution of DNN and CNN on mobile devices [35, 36]. DeepEar [36] showed the feasibility of running entire DNNs for audio sensing applications on low-power mobile DSPs. DeepX [35] then enabled the execution of DNN and CNN on mobile devices by splitting computations across multiple co-processors. We believe that *DeepMon* can complement DeepX in the following ways. First, DeepX is effective in reducing the latency of fully-connected layers while our framework focuses on reducing the latency of convolutional layers. Also, *DeepMon* supports OpenCL, Vulkan and GPUs on commercially available mobile devices in the market whereas DeepX was prototyped on more powerful hardware such as Tegra K1 and Snapdragon 801 on external development kits.

DeepSense [28] presented early evidence that using a GPU could help improve the latency of DNN computations – *DeepMon* extends that work by providing many more optimisations, a full implementation, and extensive evaluation. Recently, Glimpse [15] leveraged the cloud to enable real-time object detection and tracking while MCDNN [25] executed deep learning algorithms across mobile devices and clouds. MCDNN [25] proposed efficient optimization techniques such as building multiple smaller DNN models to recognize frequently appearing objects, sharing visual features between applications and optimizing task offloading to the clouds. However, we tackled the problem of executing the full deep learning pipelines solely within a mobile device using its GPU. In addition, while *DeepMon* shares a high-level concept with MCDNN (about re-using the DNN computation), *DeepMon's* caching technique re-uses the intermediate partial results while processing convolutional layers, enabling much more fine-granule sharing of computation across consecutive images.

Mobile Continuous Vision. Gabriel [24] uses cloudlets to support cognitive assistance applications while LiKamWa et al. presented optimization techniques for image sensors to enable continuous mobile vision [38] and Starfish [39] to support concurrent execution of multiple vision applications. *DeepMon* also aims at enabling continuous vision applications by focusing on efficiently executing deep DNNs locally on mobile devices.

Deep Learning Optimization. In the machine learning communities, there has been work to reduce the training time of CNNs and DNNs [40]. There has been some work to optimize inference time; for example, Vanhoucke et al. [50] and Jaderberg et al. [29] presented optimization techniques to reduce the inference latency (e.g., using fixed point arithmetic and SSSE3/SSE4 instructions on x86 machines). *DeepMon* uses these techniques and develops additional optimization techniques to further reduce the latency of mobile-driven continuous vision applications.

Restructuring DNN models has been widely studied to reduce the size of the model and accelerate the inference speed [18, 23, 29, 32, 34, 45]. Recently, Bhattacharya and Lane proposed a frame-

	App	Size (M)	Top-1 Acc. (%)	Top-5 Acc. (%)	Arch.
Deep Models					
VGG-VeryDeep-16	IR	138.4	71.7	90.5	13c,5p,3fc
VGG-Face	FR	145	98.95	-	13c,5p,3fc
YOLO	IR	275	63.4	-	24c,4p,2fc
Shallow Models					
AlexNet	IR	60.8	58.2	80.8	5c,3p,3fc
VGG-F	IR	60.8	58.6	80.9	5c,3p,3fc
VGG-M	IR	102.9	63.1	84.5	5c,3p,3fc
LRCN (CNN+LSTM)	AR	62.5	68.2	-	5c,3p,2lrn, 3fc

Application (IR: image recognition, FR: face recognition, AR: activity recognition),
Size: number of parameters,
Architecture (c: convolutional layers, p: pooling layers, fc: fully connected layers, lrn: local response normalization)

Table 1: Comparison of DNN Models

work to sparsify fully-connected layers and separate convolutional kernels, reducing the memory and computational costs of DNN/CNN significantly for wearables [34]. Kim et al. proposed a *Tucker-2 decomposition* technique [32]. It decomposes a tensor into three smaller ones, accelerating convolutional layer execution for mobile devices. *DeepMon* adopts the Tucker-2 decomposition and tailor it to work with our caching technique to further reduce the execution latency of convolutional layers.

3. MOTIVATING SCENARIOS

The following usage scenarios drove the development of *DeepMon*.

Speaker Identification: In this scenario, a user is wearing a pair of AR glasses and moving through a conference venue. The video stream captured by the glass is continuously processed to identify the faces of the people in view. As people are identified, information about them, such as their name, their last interaction with the user, etc., is overlayed in the glass's display.

Safety For The Elderly: In this scenario, the AR glasses are worn by an elderly person going about their regular daily chores. The glass's video feed is constantly processed to detect objects, such as oncoming traffic if a road is being crossed, or obstacles on the pavement when walking, that need to be brought to the attention of the user. In this way, the user can react early and appropriately to various situations.

Both of these scenarios, face and object detection, share similarities; (a) they both need to be performed in near real-time – identifying people after they have left or hazards after they have been hit are both undesirable outcomes, (b) the image screen may contain images (children, relatives, etc.) that should not be sent to a cloud service, (c) they can still be useful even at low frame rates, for example, face and object recognition that operates at 1 to 2 frames per second can still provide near real-time feedback to a user who is walking, and finally, (d) the state-of-the-art solutions for both problems use deep DNNs – deep DNNs are DNNs that have many computational layers (usually more than 10) that substantially increase the accuracy at the expense of higher latencies and computational requirements. In the rest of this paper, we describe how *DeepMon* enables both face and objection detection deep DNN models to run at 1 to 2 frames per second on commodity smartphones.

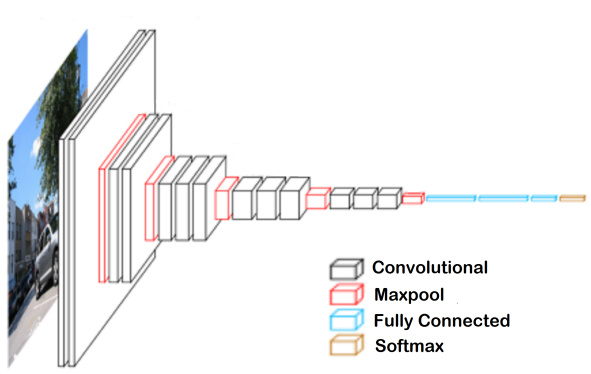


Figure 1: Macroarchitecture of VGG-VeryDeep-16 [1]

4. DEEP LEARNING PIPELINES

Vision applications use many deep learning pipelines. We explored the most popularly used models, such as AlexNet, VGG-F, VGG-VeryDeep-16, YOLO, Fast R-CNN (Region-based CNN), to characterize their computational requirements and performance – summary provided in Table 1. In this paper, we primarily focus on models (VGG-VeryDeep-16 and YOLO in particular) that adopt more than 15 processing layers to achieve higher accuracy.

4.1 Background on Various Models

VGG-VeryDeep-16 and VGG-Face. Figure 1 shows the detailed processing structure of VGG-VeryDeep-16. The architecture is composed of 13 convolutional layers, 5 pooling layers, and 3 fully-connected layers. Convolutional layers are in charge of extracting various features from an image and refining them while fully connected layers make inferences from extracted features. Pooling layers convert the data from the previous layer to feed to the next input layer. The softmax layer is the final layer to aggregate and normalize the scores generated by the last fully connected layer and outputs the final classification result.

VGG-VeryDeep-16 [46] is used to classify images into one of 1,000 different image types with a confidence probability; it outputs top-N image types with the probability per type. VGG-Face [42], is based on VGG-VeryDeep-16, and performs face recognition. We only use VGG-VeryDeep-16 in our evaluation as VGG-Face has the same structural and algorithmic properties.

YOLO [43] recognizes and locates objects in an image. YOLO can be trained with different datasets. For example, YOLO trained with the VOC dataset [2] identifies 20 objects and tracks their locations while YOLO trained with the Pascal VOC dataset [8] can identify and localize 80 different objects. The architecture of YOLO is composed of 24 convolutional layers and two fully connected layers, resulting in higher computational requirements compared to VGG-VeryDeep-16 or VGG-Face.

Other Models. There are other smaller-sized but popular models used for image classification, such as VGG-F [13], AlexNet [33], and VGG-M [13]. Their architecture incorporates a much smaller number of layers; for example, they use just 5 convolutional layers to extract features and 4 fully connected layers for inference. These models are much smaller than VGG-VeryDeep-16 or YOLO with correspondingly lower accuracies given the same train and test data. We omit these shallow models from the rest of the paper as (i) they have already been studied by prior work [35, 36], and (ii) higher accuracy object and face recognition would be more usable for end user applications.

	Conv. (ms)	FC. (ms)	Pooling (ms)	Total (ms)
VGG-VeryDeep-16	2,647	294	40	2,984
YOLO	3,345	536.1	44.9	3,935
(CNN+LSTM)	5,488.8	161.7	2,158.8	8,301

Table 2: Latency Breakdown

At the other end, some extremely deep models achieve even higher accuracies. For instance, ResNet-152 [26] has 152 layers and achieves 3.62% higher accuracy compared to VGG-VeryDeep-16. However, we noticed that the accuracy improvements of such models are marginal compared to the models that have 15 to 25 layers while incurring much higher computational costs. We do not expect those extremely deep models can be run on mobile devices in near real-time and thus exclude them from this work.

There are other models such as Faster-RCNN [22] for object detection and *Long-term Recurrent Convolutional Networks* (LRCN [19]) for activity recognition – LRCN is the combination of CNN and *Long Short Term Memory* (LSTM [27]). These models have some common characteristics with VGG-VeryDeep-16 or AlexNet and also modify the structures to achieve better performance and accuracy. Even though they are applied in different scenarios, we noticed that they have lots of commonality with VGG-based models and our workload characterization and optimization techniques apply well to these models.

Effect of the model depth on accuracy and latency. In general, the deeper the model becomes, the higher accuracy it achieves for the same classification task. This increase in accuracy has been validated by recent results [49] (although there are a few special cases where a shallow network is equally accurate). For instance, AlexNet with 5 convolutional layers achieves 80.8% top-5 accuracy to recognize an image while VGG-Verydeep-16 with 13 convolutional layers achieves 90.5% top-5 accuracy. Also, ResNet-152 with 152 layers shows 94.3% top-5 accuracy. On the other hand, deeper models impose much higher computational or memory requirements; For example, the number of operations required to execute VGG-VeryDeep-16 is 21 times more than that of AlexNet while ResNet-152 requires 4 times more memory space than VGG-VeryDeep-16.

4.2 Workload Characterization

We noticed important common characteristics in the workloads of deep deep-learning models that drove the optimisations in *DeepMon*. First, each deep model has a large number of computational layers – with the accuracy of the model increasing as more layers (convolutional layers in particular) were added. Second, the majority of the layers are convolutional layers. Convolutional layers play a critical role to extract useful features from images and then refine them; in particular, they apply various types of *filters* over the small blocks of an image to abstract out visual features such as edges and shapes. Table 1 confirms that, in deep models, the most processing layers are convolutional layers, with a small number of fully connected layers and pooling layers.

Hence, it is likely that most of the processing time is spent in convolutional layers. To check if this was the case, we measured the running time of different deep learning models on a Samsung Galaxy S7 and broke down the processing latency per layer type. To do this, we implemented a GPU-based deep learning execution framework (without any optimization techniques applied).

Table 2 shows the execution time broken down per layer type (i.e., convolutional, fully-connected, and pooling). It indicates that the convolutional layers dominate the processing time. For VGG-VeryDeep-16, over 88.7 % of the processing time is occupied by the convolutional layers followed by 9.8% and 1.3% for fully-connected layers and pooling layers, respectively. For the YOLO model, over 85% of computation time is spent in convolutional layers. The reasons for these time breakdowns are (i) there are many more convolutional layers than other layers in deep models, and (ii) the total number of addition and multiplication operations within convolutional layers is much higher compared to fully connected layers and pooling layers (e.g. VGG-VeryDeep-16 requires 15,346M addition and multiplication operations for convolutional layers while only 123M operations are necessary for fully connected layers). These results suggest that optimizing the processing time of convolutional layers would lead to huge improvements in overall model processing latencies.

5. DESIGN CONSIDERATIONS

We developed *DeepMon* with the following design goals:

No cloud offloading: Our primary design goal, for this paper, was to use local phone resources only without any cloud offloading to process deep DNNs as this area has compelling use cases without any viable solutions. There are also scenarios, such as processing of sensitive video feeds or video processing in places with poor or expensive networking connectivity, where offloading is either unwanted (due to privacy concerns) or impossible (due to networking issues). We do plan to extend our solution to support cyber foraging (e.g. MAUI [16] and Chroma [10, 11]), where local and cloud resources are used in a dynamic fashion.

Near real-time latency: Our intended application scenarios, described in Section 3, both require near-real time processing of image streams to give on-the-fly feedback to the users. However, we do not aim to provide strict real-time support (e.g., < 50ms with strict inter-frame timings) as we do not believe this is possible with current commodity smartphones and deep DNNs. Instead, we aim to push the research boundary to provide 1-2 frames per second processing capability (the current state-of-the-art is 1 frame every 3-4 seconds).

Minimal accuracy loss: While achieving near-real-time processing latencies is good, it cannot be done at the cost of accuracy – otherwise improving latency becomes trivially easy. We thus require *DeepMon* to be only about 5% less accurate than running the same model on a desktop PC.

Efficient power use. Minimizing the energy use of *DeepMon* is essential as we aim at running complex deep learning pipelines on mobile devices. In this paper, we focused on reducing the power consumption of executing deep learning pipeline on a mobile device and rely optimising the power consumption of the video camera (to capture and store continuous video feeds) to prior work [38].

Support a wide range of mobile GPUs and programming APIs: There has been prior work [25, 35] that used external mobile development boards, such as the Tegra K1, to test their solutions. We designed *DeepMon* to work well on commodity smartphones and tested it across a range of mobile phones and programming APIs (the full list of test devices is shown in Table 3). In particular, *DeepMon* supports both the OpenCL [7] and Vulkan [9] programming APIs.

6. IMPLEMENTATION

In this section, we first show the overall architecture of *DeepMon*, and then describe, in detail, the various techniques we adopted to optimize the execution of deep learning pipelines.

Phone	GPU	APIs	# GPU Cores (#ALUs)	Memory Size (GB)	Memory Bandwid. (GB/s)
Samsung S7	Mali T880	OpenCL/ Vulkan	12	4	25.6
Samsung Note 4	Adreno 420	OpenCL	4 (128)	3	12.8
Sony Z5	Adreno 430	OpenCL	4 (192)	3	12.8

Table 3: Specs for Commodity Mobile GPUs

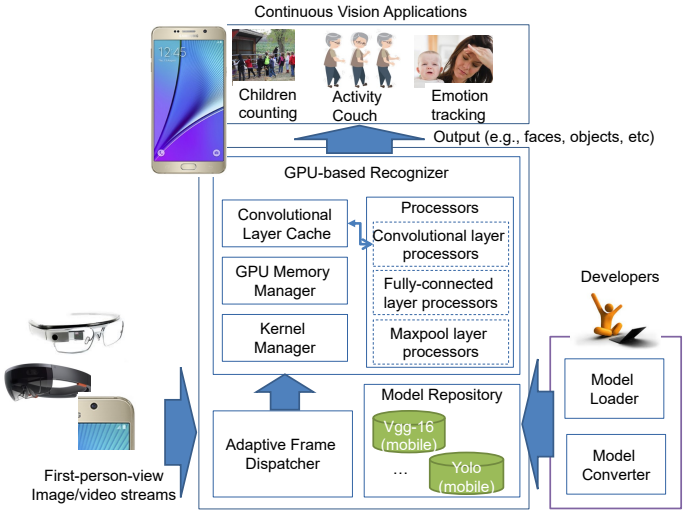


Figure 2: *DeepMon* System Architecture

Techniques	Description	Evaluation
Model Conversion/Loading	Section 6.2	-
Convolutional Layer Caching	Section 6.3	Section 7.2–7.5& Section 7.8–7.9
Layer Decomposition	Section 6.4	Section 7.2–7.5
Convolution Optimizations	Section 6.5	Section 7.2–7.5
Scaling to various GPUs/APIs	Section 6.6&6.2	Section 7.6&7.7

Table 4: Summary of *DeepMon's* techniques

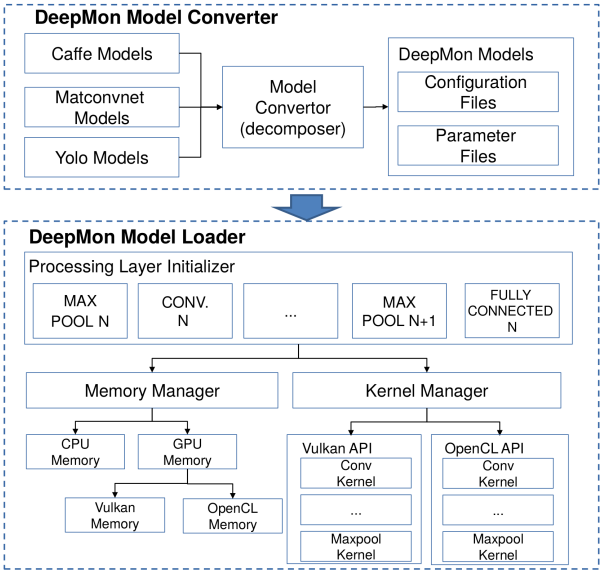


Figure 3: The Flow of Model Conversation and Loading

6.1 Architecture Overview

Figure 2 shows the overall architecture of *DeepMon*, and Table 4 summarizes our techniques. *DeepMon* works through two different phases: (1) the model conversion phase to convert existing models to run on mobile GPUs, and (2) the inference phase to process image streams using the converted model to recognize useful information.

Model conversion and loading. To use *DeepMon*, developers first need to convert existing deep learning models (built for desktop GPUs) to fit on mobile GPUs. For this, we provide *model converter* and *model loader* tools – the current *DeepMon* prototype can convert a variety of existing models including the ones described in Table 1. The model converter adapts the configurations and parameters of an existing model and generates a new model that can run efficiently on mobile GPUs (See Section 6.4). The model loader then loads the generated model on *DeepMon*– it allocates adequate memory spaces to lay out input data for efficient convolution computation and structures the processors for all the layers composing the model (See Section 6.2)

DeepMon currently supports the models from three different deep learning frameworks, namely *Caffe* [30], *Matconvnet* [51] and *YOLO* [43].

Real-time Inference. During the inference phase, *DeepMon* takes a stream of first-person-view images as its input. The *frame dispatcher* selects important frames to recognize and feeds them to the GPU-based recognizer. Then, the *GPU-based recognizer* executes the deep learning pipeline and outputs its inference results to the applications of interest. During the execution, it applies a suite of optimization techniques, such as convolutional layer caching and matrix multiplication optimizations, to boost the recognition speed (explained in detail in Sections 6.3 and 6.5).

DeepMon supports both OpenCL and Vulkan and was tested on phones with Adreno and Mali GPUs. We present our evaluation results for various GPUs and Vulcan in Section 7.6 and 7.7.

6.2 Loading Models into Mobile GPUs

Figure 3 shows the detailed flow of the model conversion and loading process. First, the model convertor decides how to layout the input data into the memory space. The challenge here occurs mainly because the memory space is linear while the input data are multi-dimensional matrices. The wrong unfolding of the multi-dimensional data into a linear space would result in huge fragmentation of the data, which will slow down the convolution processing significantly. Intuitively, the model converter lays out the data such that matrix multiplications can be done by reading consecutive memory blocks and reusing them as much as possible once they are in memory. This is particularly important for devices with low memory bandwidth (e.g. Samsung Galaxy Note 4 and Sony Xperia Z5).

Once the data layout is decided, the *model loader* initializes all the necessary additional layers (e.g. convolutional, pooling, fully-connected, etc.) within the *DeepMon's* recognizer. During initialization, *DeepMon* performs two important tasks: (a) memory allocation and (b) kernel code compilation.

First, upon layer initialization, *DeepMon* needs to allocate memory spaces to store the metadata (e.g. size of filters, input size,

(a) An image at time t_0 (Left)
(b) An image at time $t_0 + 500ms$ (Middle)
(c) The same image blocks marked as black (Right)

Figure 4: Example First-Person-View Images

output size, etc.) or parameter values. *DeepMon* stores all the metadata in the host memory (or CPU memory) for easy data access and stores all the parameters in the device memory (or GPU memory). The GPU memory space is allocated based on the API used (OpenCL or Vulkan). The memory space for the actual input and output data is also allocated in the GPU memory for efficient computation. This space can be mapped to the host memory when necessary (e.g. to return final output to application).

Second, a specific *kernel code*, containing the code block to be parallelized by the layer, needs to be built and loaded. Building these kernel code is handled differently for OpenCL and Vulkan. For OpenCL, a kernel is written in the OpenCL C-like language. It does not require pre-built binary code for any specific device – Instead, it supports compilation capabilities on the target device itself, making it easy to be ported to other OpenCL-enabled devices. Vulkan, on the other hand, uses SPIR-V (Standard Portable Intermediate Representation), an intermediate language for graphics and parallel computation. In Vulkan, SPIR-V code can be loaded onto various Vulkan-enabled devices without building binary code. *DeepMon* prepares two separate convolutional implementations in advance and compiles the kernel code on demand, based on the chosen API, when a layer is initialized and loads the kernel into memory.

6.3 Convolutional Layer Caching

As shown earlier (Section 4), the convolutional layers are the main performance bottlenecks. To accelerate the computation of these layers, we designed a caching mechanism optimised for convolutional layers. Our key observation is that first-person-view images tend not to change much over a short time duration. For example, Figure 4 shows three first-person-view images; the left and middle images were taken at time t_0 and $t_0 + 500ms$ while the rightmost image, taken at time $t_0 + 500ms$ shows the same image blocks (marked as black).

In particular, the background of images across multiple continuous image frames often remains still while foreground objects tend to move. Such commonality in images incurs heavy repetition in the execution of convolutional layers as applying the full pipeline on one image at a time applies the same convolution computations on many different "repeated" frames and sub-frames.

Our caching mechanism reduces this repetitive computation significantly. A plausible caching approach would be to reuse the final result from the previous frame when the difference between frames is under a certain threshold (Chen et al. [15] proposed a similar idea). However, this approach would not work in many cases as foreground objects (that take a small portion of the entire image but are important to recognize) tend to change noticeably while the background images do not. This makes the previously cached results either stale (on a cache hit) or incurs lots of cache misses. To overcome this, we *cache the partial results of convolutional layers* – i.e., we reuse the convolution outputs for the unchanged blocks of an image while recalculating convolutions for the changed blocks.

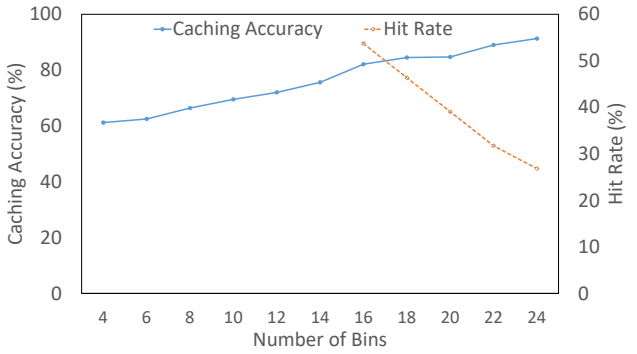


Figure 5: Effect of the Number of Bins on Caching

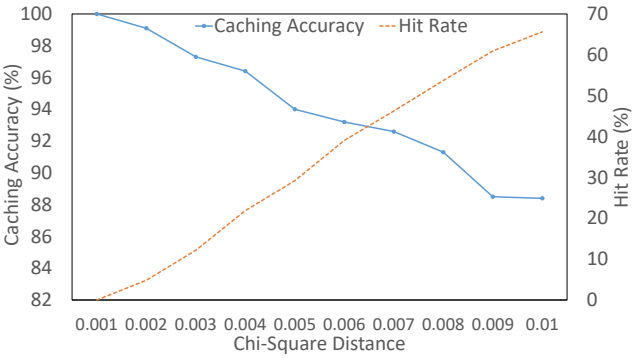


Figure 6: Effect of the Distance Values on Caching

6.3.1 Caching Mechanism

The overall flow of our caching mechanism is as follows. First, we divide the image into a grid (e.g. an 8x8 grid) where each grid block contains a fixed number of pixels. During the execution, we compare corresponding blocks, $b_{(t-1)}$ and b_t of two consecutive images to determine if the outputs of the previous convolutional layer, $b_{(t-1)}$, are reusable (i.e., it is a cache hit). Upon a cache hit, *DeepMon* skips the convolution computation on the pixels within the entire block. *DeepMon* caches the convolution outputs for the first N convolutional layers only (where N is determined empirically for every model) since the computation for the later convolutional layers are often quite small, and the caching overhead is higher than the benefit. Cached values expire after a certain duration – for example, we set the default expiration times, determined empirically, to 650ms for VGG-VeryDeep-16 and 1000ms for YOLO.

However, the key challenge with this caching scheme is that it is a non-trivial task to determine if the two image blocks are similar or not. Indeed, if the image comparison is too heavyweight, the caching overhead will quickly exceed its benefit. There are a few image comparison algorithms with high comparison accuracy, for example SIFT-based [41] and Hog-based [17] algorithms. However, their computational cost is high and not suitable for our cache design (See Section 7.9 for the relevant results.).

To solve this problem, we adopted a light-weight algorithm based on colour histograms. For the two image blocks to compare, we compute the histogram of the colour distribution and compute a *chi-square distance metric*. If the distance is less than a pre-defined threshold, the cell is marked as "reusable".

For efficient caching, it is important to choose the right number of bins (to calculate the histogram) and the distance threshold. We carefully chose the right parameters through empirical studies using AlexNet. First, we investigated the effect of the number of bins by fixing the distance parameter to 0.005. Figure 5 shows that as the number of bins increases, the accuracy increases while the number of "cacheable" blocks decreases; in the figure, the caching accuracy indicates how closely *DeepMon* outputs the final classification results with respect comparing to the original model. We also explored the effect of distance threshold – the number of bins was set to 16. Figure 6 shows the trade-off between accuracy and cache hit rates for various distance threshold values. We use the cross-over points of the accuracy and hit rate to decide a plausible number of bins and distance threshold value.

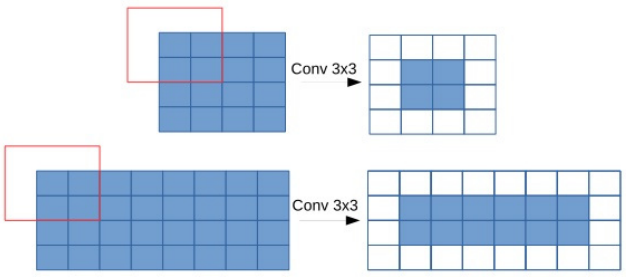
To make caching work efficiently along with our GPU-based recognizer, we carefully re-implemented our GPU-specific kernels. Intuitively, we first initialize all the memory spaces (that need to contain the output of a convolutional layer) with the cached results. Only for those blocks with cache misses, *DeepMon* maps the new outputs into the corresponding memory spaces. This makes updating uncached results easy.

When reusing cached results, we had to be careful about the edges of an image block. Figure 7 shows two examples of caching applied on a block size of 4x4 in a convolutional layer with a filter size of 3x3. Figure 7(a) shows an example where a convolutional filter is applied to the edge of the cached block. In this case, the output value becomes non-cachable as the 3x3 block being calculated may refer to non-cachable data (data outside of cached block). For that reason, *DeepMon* will not reuse the results for the edges of the block. However, when two or more consecutive blocks can be cached, as shown in Figure 7(b), *DeepMon* reuses the cached results for the edges that are shared by the cacheable consecutive blocks. Importantly, for the models we are considering, the block size is quite large for the first few convolutional layers (e.g. 28x28 pixels for the first layer for VGG-VeryDeep-16), making this caching technique effective for all those layers.

6.4 Convolutional Layer Decomposition

We further optimize convolutional layers by decomposing the convolutional parameters. Convolutional layers are well-known to have redundant parameters [29], making them computationally inefficient on resource-constrained devices. Prior research have provided a few different methods (such as the tucker decomposition [32] and CP decomposition [37]) to decompose a convolutional layer into three smaller convolutional layers so that the total computation of the decomposed layers is less than that of the original layer.

DeepMon adopts a variance of the Tucker decomposition named *Tucker-2* [32] over other alternatives since it is a better match to *DeepMon's* caching algorithm. The weights of a convolutional layer are often represented as a tensor T of size *[N x C x D x D]* in which N and C are the numbers of input and output channels, respectively, while D is the size of the filters. Tucker-2 decomposes T into three smaller tensors *T1, T2, T2* with the sizes of *[C' x C x 1 x 1]*, *[N' x C' x D x D]*, *[N x N' x 1 x 1]* respectively, where the number of new input and output channels (i.e. N' and C') are reduced compared to those in the original tensor (i.e. N and C). Intuitively, the decomposition reduces the number of dot product operations from *(N x C x D x D)* to *(C' x C) + (N' x C' x D x D) + (N x N')*, enabling *DeepMon* to further reduce the latency.



(a) Example with a 4x4 Block and a 3x3 Filter (Top)
(b) Example with two 4x4 Consecutive Blocks and a 3x3 Filter (Bottom)

Figure 7: Caching on the Edge of an Image Block

Tucker-2 decomposition is more appropriate to be used with our caching technique due to its unique characteristic – two of the decomposed layers have the filters with the size of *[1 x 1]*. *[1 x 1]* filters do not reduce the input size to the subsequent layers, keeping the cacheable block size across layers; note that if the block sizes get reduced, the overhead to compute cache hit/miss will increase, compromising the benefit of caching. On the other hand, other decomposition methods use filters larger than *[1 x 1]*, reducing the size of *cachable* blocks and making the caching less effective. Moreover, the *[1 x 1]* filter does not require separate handling of the edges of cacheable blocks (as shown in the Figure 7). This enables us to develop a more efficient GPU-kernel to reduce the latency further.

The non-trivial problem, here, is to choose the right *N' and C'*. In practice, manual trial and error is still a common yet inefficient approach that requires a lot of effort. Instead, we devised a *double binary search algorithm* to reduce the amount of effort needed. The key idea behind the algorithm is to find *N' and C'* that maximizes the variance when we reconstruct the original tensor (e.g. similar to principle component analysis). We define the desired variance that we need to sufficiently reconstruct the tensor and then use binary search to find the parameters that best produce the required variance. Finally, we fine-tune the model to recover from the possible loss in its accuracy.

6.5 Optimizing Convolutional Operation

The execution of a deep learning pipeline heavily relies on matrix multiplication. However, linear algebra libraries for OpenCL (such as ClBlast and ViennaCL used in Caffe) are tuned for desktop GPUs and do not perform efficiently on smartphones.

To accelerate convolutional operation, existing frameworks use a technique called *unfolding* that converts inputs into a large matrix and then uses matrix multiplication on the unfolded input and filters to compute the result [14]. The unfolding technique requires a large amount of memory and bandwidth when executing convolutional layers. Unfortunately, the memory bandwidth on mobile GPU is quite small compared to server GPU. This makes the unfolding technique unsuitable for *DeepMon*.

Deeper observations showed that convolutional operations performed without unfolding tend to consume less bandwidth for memory access. However, it also stores the data, in memory, in a non-contiguous fashion, making it inefficient when running on memory-constrained mobile GPUs. Our second observation is that carefully laying out the convolutional weights in the format of *[N x D x D x*

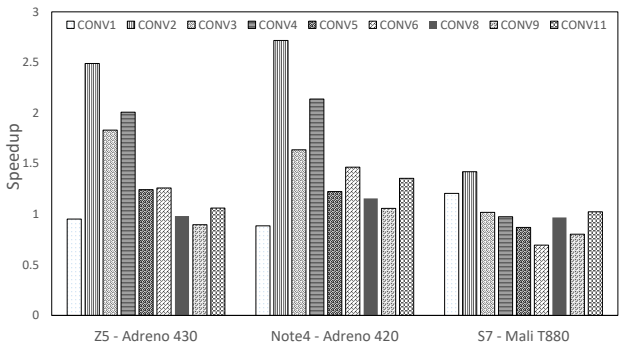


Figure 8: Speedup Comparison with ClBlast

C] and its input in *[H x W x C]* makes it more GPU friendly as we can read multiple items at the same time using OpenCL functionality. We also note that Caffe and YOLO use the format of *[N x C x D x D]* for the weights of convolutional layers.

Figure 8 shows the speedup between our implementation and conventional unfolding approach. We benchmark two approaches using convolutional layers extracted from VGG-VeryDeep-16. We drop convolutional layer 7, 10, 11 and 12 from our benchmark since they have similar parameters to the other layers. We extract unfolding kernel from Caffe and use ClBlast library (one of three linear algebra used in Caffe) to do convolutional operations. Results show that on lower bandwidth devices (Note 4 and Z5), our approach almost provides the better latency (speedup > 1). However, on S7, since the device has integrated LPDDR4 which has doubled bandwidth comparing to two other devices, conventional approach starts to benefit at some layers. We additionally validated the latency of convolutional layers with another commonly-used library, ViennaCL (on Caffe). We found out that ViennaCL performs slower than ClBlast on Samsung Galaxy S7 – mainly due to its lack of support to optimize various parameters.

We further reduce the processing latency of the convolutional operations by using half floating point precision in OpenCL. Since the memory bandwidth is limited on the mobile devices (compared to desktop machines), it is highly useful to reduce the size of memory reads and writes by half by dropping the last half digits of the data. Our results, shown in Section 7, indicate that this optimisation is effective at reducing latency without any significant impact on the accuracy.

6.6 Scaling to Various Mobile GPUs

We implemented a number of techniques to allow *DeepMon* to support various types of mobile GPUs. The most important consideration was to adapt to the different memory architectures of different mobile GPUs and the ways in which they read/write data from the main memory.

Mobile GPUs support unified memory access that allows GPUs to directly access the main memory and use it as its own memory. However, the main memory is shared among the many components of a mobile device and its data read/write bandwidth is limited. This limited bandwidth could slow down the processing of *DeepMon* as DNN execution usually requires the GPU to read a large amount of data from the main memory.

One possible solution is to use *local memory* on the GPU chipset itself. The local memory is a small memory (for instance 8KB on Adreno 330 and 32KB on Adreno 430) which is used as a cache to accelerate memory access during computation (data is first loaded into local memory and is reused during computation). However, the size and architecture of the local memory vary across different GPUs. For example, different Adreno boards have different sizes of local memory while Mali GPUs have no local memory. Such differences are the key challenge in making *DeepMon* support different mobile GPUs.

	conv_1 (ms)	conv_2 (ms)	conv_3 (ms)	conv_4 (ms)
Host memory	78.66	667.10	340.59	757.12
GPU local memory	63.98	526.9	262.57	584.80

Table 5: Benefit of using GPU Local Memory

We address this issue by building kernel codes that can exploit different amounts of local memory (including a kernel code for no memory) and dynamically uses the appropriate code at runtime. In particular, when executing convolution layers, if the memory requirement for a single filter fits into the small local memory, we adaptively use kernel code that supports that amount of local memory. Otherwise, we use the non-local-memory version.

We also build the kernel code in a way that the filters within a convolutional layer are shared to evaluate all input values. Accordingly, for the first layer of VGG-VeryDeep-16, we can fit all 64 filters with the size of *[3 x 3 x 3]* into the 8KB local memory of the Xperia Z5. For the deeper layers that require more than available local memory, *DeepMon* loads a subset of filters into the local memory and compute partial outputs at a time. We also find out that the half floating point approximation reduces the size of filters by half, allowing *DeepMon* to load more filters into the local memory. Table 5 shows the processing time for the four first convolutional layers while executing VGG-VeryDeep-16 on the Sony Xperia Z5 phone. It indicates that the use of local memory accelerates the processing time by 23-30%.

7. EXPERIMENTS

7.1 Experimental Setup

We extensively measured the performance of *DeepMon* with a variety of deep learning models and mobile GPUs.

Workloads. We used a variety of deep learning models as shown in Table 1. We mainly report the results for two deep models, VGG-VeryDeep-16 and YOLO, and report the results for other models only when they are significant. We used the VGG-VeryDeep-16 model trained with the ILSVRC2012 train dataset [44] and YOLO trained with the Pascal VOC 2007 train dataset [8].

Metrics and datasets. We used processing latency, accuracy, and powerer consumption as our key evaluation metrics. For the latency, we measured the duration to process an image, i.e., $t_1 - t_2$ where t_1 is the time that *DeepMon* outputs the inference result and t_2 is the time that *DeepMon* receives the input image. For the latency evaluation, we used two test datasets: (i) the *UCF101* dataset [48] comprising 13,421 short videos (less than a minute long) created for activity recognition and (ii) LENA dataset [47] consists of 200 first-person-view videos captured from Google Glasses, and report the average latency across all processed frames along with the 95% confidence interval. We used the UCF101 dataset by default while we report the performance for LENA dataset in Section 7.8 and Section 7.9.

For accuracy, we measured the percentage of accuracy drop compared to the original models. We focused on the drop as our goal is not to improve the accuracy but to keep it close to that of the original models while accelerating inference speed. Note: unlike prior

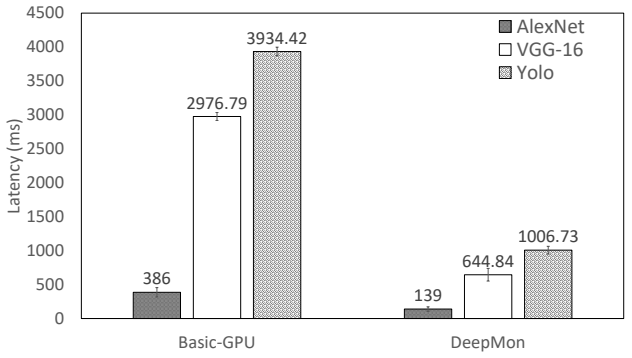


Figure 9: Overall Processing Latency

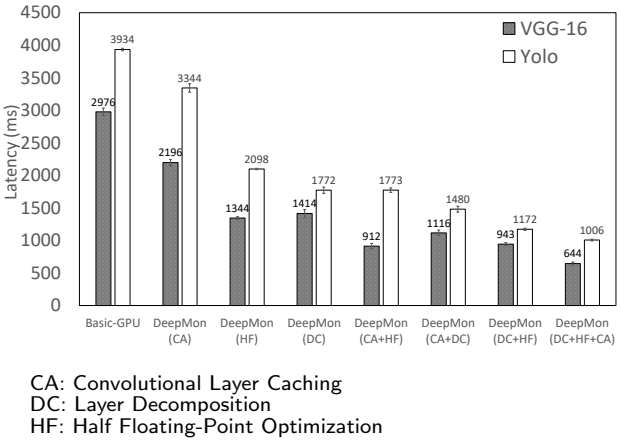


Figure 10: *DeepMon* Latency Breakdown

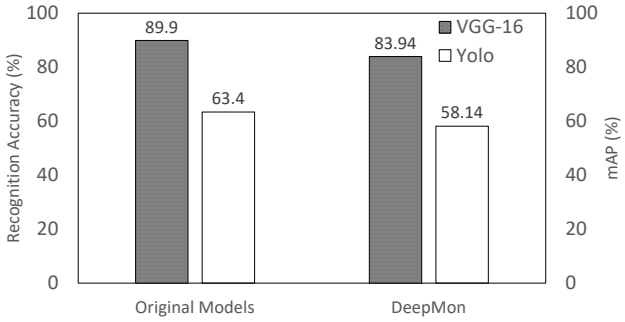


Figure 11: Recognition Accuracy

work [25], we did not reduce the total number of possible output categories (e.g., the number of objects that can be recognized by the model). For the accuracy evaluation, we used the ILSVRC2012 [44] validation dataset for VGG-VeryDeep-16 and the Pascal VOC 2007 test dataset [20] for YOLO, and calculated the average accuracy over each test dataset. For YOLO, we used the mean average precision (mAP), which is a standard metric to evaluate the YOLO's accuracy regarding both object recognition and localization [21].

Finally, we measured the power consumption using the Monsoon power monitor [5]. We reported the average energy consumption of the smartphone while processing an image in uAh by measuring the baseline energy consumption before running the processing logic and deducting the baseline from the measured value. For energy evaluation, we used the UCF101 dataset (the same dataset used in the latency evaluation), and report the average energy consumption across all processed frames along with the 95% confidence interval.

Alternatives. We compared the performance of *DeepMon* with other plausible smartphone-based alternatives such as *basic-CPU* and *basic-GPU*, and a few cloud-based alternatives. *basic-CPU* only uses the mobile CPUs to compute the full deep learning pipelines while *basic-GPU* utilizes the mobile GPUs for all processing layers without optimization. For the cloud-based approaches, the mobile device sends images to a cloud server, the server processes the images and return the results back to the mobile device (details of the cloud-based alternatives are explained in Section 7.4). Also, to look into the benefit of *DeepMon*, we applied different combinations of the optimization techniques presented in Section 6 such as convolutional layer caching (denoted as CA in the figures), layer decomposition (DC), and half floating-point calculation (HF).

Devices and APIs. We used a Samsung Galaxy S7 (with Mali T880 GPU), a Samsung Galaxy Note 4 (with Adreno 420), and a Sony Xperia Z5 (with Adreno 430) as our experiment devices. Unless mentioned, we used the S7 as the default device. Also, we used the OpenCL implementation of *DeepMon* by default while we measured the performance of the Vulkan implementation in Section 7.7.

7.2 Processing Latency

We first study the overall processing latency of *DeepMon* in comparison with naive approaches. Figure 9 shows the results, on an S7, for the three models: AlexNet (trained with the ILSVRC2012 train dataset), VGG-VeryDeep-16 and YOLO.

The figure shows that *DeepMon* accelerates the processing of deep learning models by 3-5 times compared to *basic-GPU*. *DeepMon* processes VGG-VeryDeep-16, a model with 13 convolutional layers and 3 fully-connected layers, at the latency of 644ms, enabling near real-time processing of continuous image streams. YOLO takes about 1 second as it includes more number of convolutional layers to track their locations of the objects.

For smaller models such as AlexNet (or equivalents such as VGG-F or VGG-M with 5 convolutional layers and 3 fully-connected layers), *DeepMon* can process an image with just 139 ms of latency. Note: The processing time of *basic-CPU* is slower by one or two orders of magnitude depending on the model. It takes 6345ms for *basic-CPU* to process an image using AlexNet, which is 45.6 times slower than *DeepMon*.

Digging deeper, we analysed which *DeepMon* techniques contribute to the processing benefits. Figure 10 shows the latency breakdown for VGG-VeryDeep-16 and YOLO while cumulatively applying the various optimization techniques. The results show that all techniques significantly contribute to the latency reduction for VGG-VeryDeep-16. For YOLO, the benefit of the caching was smaller than VGG-VeryDeep-16 as the layer decomposition technique highly optimizes the first few convolutional layers, making the reuse of the cached results less beneficial.

7.3 Recognition Accuracy

Next, we investigate how much accuracy *DeepMon* compromises in return for the latency benefits. Figure 11 shows the classification accuracy of the original VGG-VeryDeep-16 and the mAP of YOLO as well as the converted models optimized to run on *DeepMon*. The figure shows that *DeepMon* drops about 5-6% of ac-

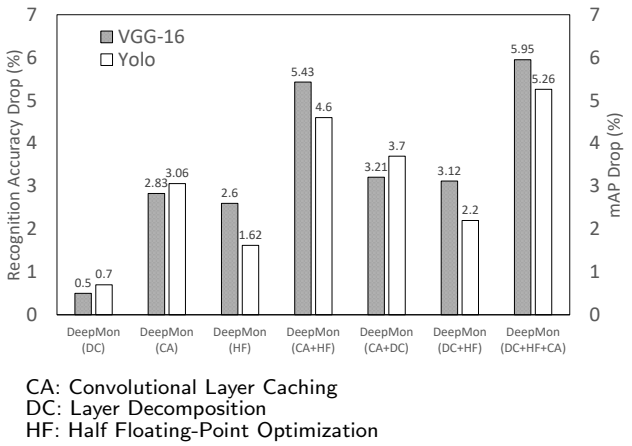


Figure 12: Breakdown of *DeepMon* Accuracy Drop

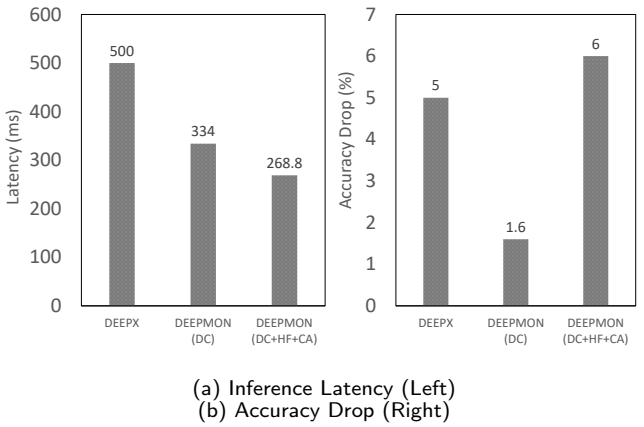


Figure 13: Comparison to DeepX on Samsung Galaxy S5

curacy while accelerating the latency 4-5 times. We designed our techniques to keep the properties of the original architecture, thus minimizing the impact on the recognition output. Note that Fast-YOLO [43], a lightweight version of YOLO shows the lower mAP of 52.7%, which is 5.44% lower than that of *DeepMon*, while the latency benefit of Fast-YOLO was similar to *DeepMon* (i.e., ≈4.5 times when experimented on Samsung S7).

We further analysed which of *DeepMon's* components contributed to the accuracy drop. Figure 12 shows the results by applying the three different techniques that affect the accuracy. The accuracy drop by layer decomposition is marginal, indicating that our binary-search-based decomposition selects suitable decomposition parameters. Also, the convolutional layer caching reduces accuracy by about ≈3%, showing that the use of cached results marginally affects the accuracy for video streams.

7.4 Comparison with Other Approaches

We now compare the processing latency of *DeepMon* with DeepX, the state-of-the-art mobile deep learning inference engine. Figure 13 shows the latency and accuracy drop of DeepX and *DeepMon*; we ran AlexNet using the SnapDragon 801 processor. DeepX consumes 500ms to process an image with an accuracy drop of 5%. *DeepMon's* latency was 269ms, twice as fast as DeepX, when all techniques are applied while its accuracy drop was 1% higher at 6%. *DeepMon* can be adjusted to only use the layer decomposition

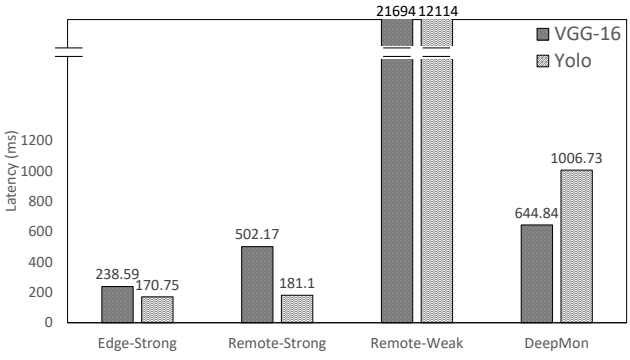


Figure 14: Comparison with the Cloud-based Approach

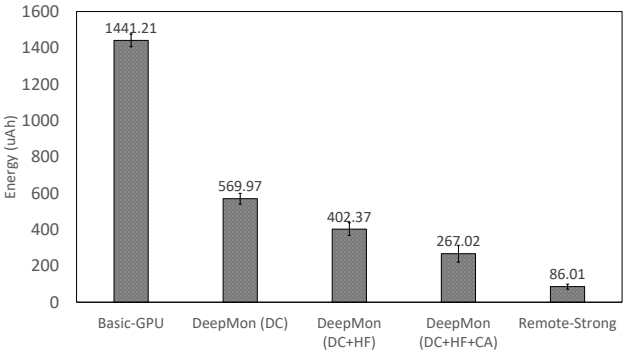


Figure 15: Overall Power Consumption

method which achieves 333ms latency (≈33% faster than DeepX), but with only a 1.6% accuracy drop.

We also compared the latency of *DeepMon* with the cloud-based alternatives. Figure 14 shows the results. We used three different cloud variants: *edge-strong*, *remote-strong*, and *remote-weak*. For, *edge-strong*, the mobile phone and the server was connected through the local Wi-Fi network while the server is equipped with a NVidia GTX 980 GPU (2,048 GPU cores, 8GB memory size and 224GB/s memory bandwidth). For *remote-strong* and *remote-weak*, we used Amazon EC2 servers (in particular g2.2xlarge and t2.medium instances respectively) located in the EC2 Asia Pacific (Singapore) datacenter. *remote-strong* was equipped with a K520 GPU (with 8 cores and 15 GB of memory) with while *remote-weak* had no GPU. We used the Caffe [30] and YOLO [43] frameworks to run the models on the cloud.

edge-strong is 2.7 times faster than *DeepMon* while *remote-strong* is only 28% faster than *DeepMon*for VGG-VeryDeep-16. The latencies of *remote-weak* were 33.6 and 12 times slower than *DeepMon* , respectively, due to its CPU-based execution of deep learning models. This suggests that we can leverage cloud services for home- or office-based applications where the user can offload the data safely to the edge servers with low networking latency and fewer privacy concerns. On the other hand, we need to be careful about using the remote clouds even when the users are willing to send the data. The cost for *remote-strong* (using g2.2xlarge server instance) is 1 USD per hour, imposing huge service cost for continuous vision applications. We can use less powerful instances, although doing so might not improve the latency as indicated by the numbers for *remote-weak*.

7.5 Power Consumption

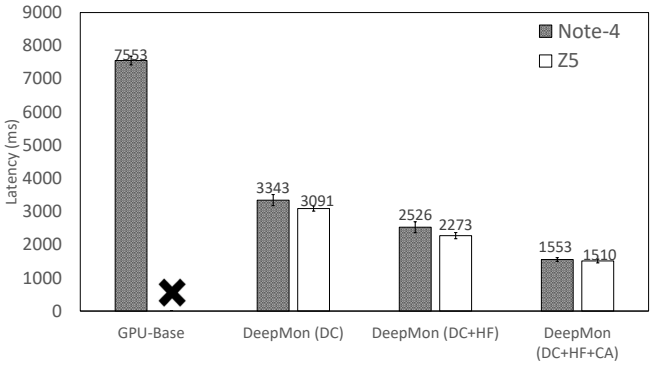


Figure 16: Processing Latency for Different GPUs

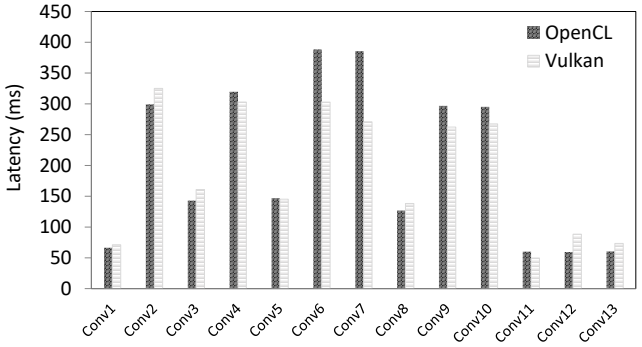


Figure 17: Performance of Vulkan

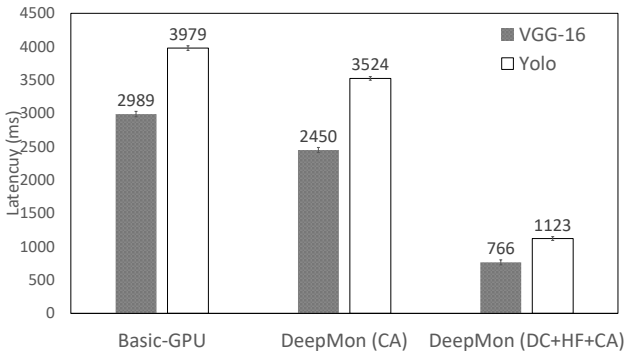


Figure 18: Latency on the LENA Dataset

We now investigate the power consumption of *DeepMon* in comparison with *basic-GPU*, *remote-strong*, and *remote-weak*. Figure 15 shows the overall power consumption for each approach along with the breakdown. All *DeepMon* measurements were done on Samsung Note 4 as it has a detachable battery that could be tapped with the Monsoon power meter.

The figure shows that *DeepMon* is lower than the power consumption of basic-GPU by more than 5 times for both VGG-VeryDeep-16 and YOLO. This savings is mostly from the reduced processing time. *remote-strong* consumes 3 times lesser power as the mobile device consumes power only to send the image to the cloud and then goes into power saving mode until it receives the result. However, as stated earlier, you need a large expensive server instance to see small latency benefits compared to *DeepMon*.

7.6 Latency on Other Mobile GPUs

We next studied the processing latency of *DeepMon* across different GPUs. We used a Samsung Galaxy Note 4 (with Adreno 420) and a Sony Xperia Z5 (Adreno 430). Figure 16 shows the results. While the latency reduction pattern by all our optimization strategies remains similar, the absolute processing latency increases by 2.4 times for the Note 4 and 2.34 times for the Z5, compared to the Samsung Galaxy S7 (with Mali T 880). Even though the direct comparison between Mali and Adreno is non-trivial, Mali's faster performance is likely to result from having more GPU cores and higher memory bandwidth compared to Adreno 420 and 430. We also noticed that the original VGG-VeryDeep-16 model cannot be run on Z5 due to the limitations of the heap memory size – although it can run after the decomposition technique reduces the model size by half.

7.7 Latency of Vulkan

We also explored the performance of the Vulkan implementation of *DeepMon*. We used the Samsung Galaxy S7 that supports both Vulkan and OpenCL. Figure 17 shows the processing time per convolutional layer for VGG-VeryDeep-16. Even though there are small differences in processing time per layer (compared to OpenCL), all our techniques are equally effective on Vulkan as well, resulting in similar overall processing times.

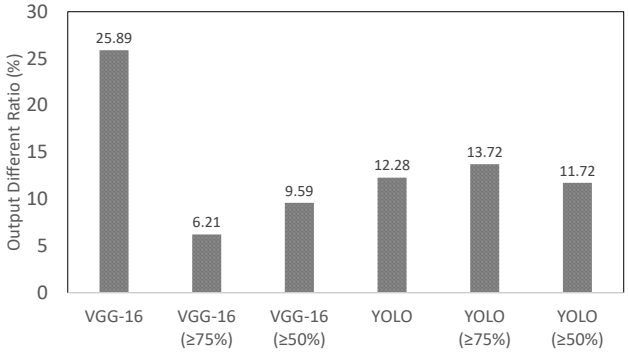
7.8 Performance on First-Person-View Videos

We further evaluated the latency and accuracy of *DeepMon* over the first-person-view dataset, LENA, which could be the typical workload for *DeepMon*. For accuracy, we reported the percentage of frames that the base model and *DeepMon* outputs the different classification result – we define this as the output difference ratio. For VGG-VeryDeep-16, we consider that the output is different when the top-1 classification results of the base model and *DeepMon* are different. For YOLO, we consider that the output is different when the positions of the detected object (indicated as rectangles on the image) overlap less than 50% (i.e., *Intersection-Over-Union (IoU)* < 50%).

Figure 18 shows the latency of *DeepMon* on the entire LENA dataset. *DeepMon* shows ≈4 times of overall latency reduction, which is comparable to the benefit over the UCF101 dataset. In particular, our caching technique reduced ≈22% and ≈13% of the total execution times of VGG-Verydeep-16 and YOLO, respectively. The reduction rate was slightly decreased compared to that of the UCF101 dataset since the first-person-view videos tend to have more frequent changes in the recorded scenes due to continuous head movement. However, the results show that our caching technique is still effective for the first-person-view videos.

Figure 19 shows the output difference ratio. *DeepMon* produces different outputs for 25.89% and 12.28% of the total frames compared to the base VGG-VeryDeep-16 and YOLO models, respectively. We empirically looked into such differently-classified frames and found out that most of those frames are not correctly classified or do not have a matching class in the base model, margin-ally affecting the actual accuracy.

Interestingly, the output difference ratio of VGG-Verydeep-16 is much higher than that of YOLO. This is because VGG-Verydeep-16 always outputs one of the 1,000 pre-trained classes even though the target frame is unlikely to be one of the 1,000 classes; for the consecutive frames with low classification confidences, their top-1 classified objects vary sensitively from one frame to another (al-



VGG-16(≥75%) indicates the accuracy evaluated only on the videos with the average confident score above 75%. Similar explanation applies to VGG-16(≥50%), YOLO(≥75%), and YOLO(≥50%).

Figure 19: Accuracy on the LENA Dataset

	SIFT-based	Histogram-based
Overhead (ms)	2,580	4.77
Overall latency change (ms)	2,064 (increased)	-534 (decreased)
Output difference ratio (%)	3.875	6.21
Cache hit rate (%)	31.4	35.52

Table 6: Caching Performance Analysis

though the frames include the same object), making our caching results different from the newly calculated ones. We further calculated the output difference ratio only over the videos that have the average classification confidence higher than 75% and 50%, and the output difference ratio was reduced to 6.21 and 9.59, respectively. For YOLO, the output difference ratio did not vary much since the model eliminated "others" when its classification confidence was below a certain threshold.

7.9 Convolutional Layer Caching Performance

We further studied how our caching technique performed over the LENA dataset. We used Vgg-VeryDeep-16 for this study. Table 6 shows the results on the videos with the average confidence score over 75%. *DeepMon* (with its histogram-based caching) shows the average latency reduction of 538 ms. The benefit comes from 35.52% of cache hits, significantly reducing unnecessary recalculation of convolution operations. We noticed that the latency reduction was ≈20% less than that of the UCF101 dataset. As expected, the cache hit rate over LENA, the first-person-view dataset, was lower compared the cache hit rate over the UCF101 dataset. This is mainly because head-mounted cameras tend to move more than third-person-view cameras, resulting in bigger differences between the two consecutive images.

We also compared our proposed histogram-based caching algorithm against an alternative using SIFT features [41]. Although SIFT-based algorithm provides the lower output difference ratio (3.875%) than the ratio of the histogram-based algorithm (6.21%), extracting SIFT features from multiple blocks of an image is highly time-consuming; it took over 2.5 seconds to calculate SIFT features for an image (across all convolutional layers). Due to high overhead to calculate the SIFT features, it cannot be used to compare image blocks for caching. On the other hand, our histogram-based approach can compare blocks of an image within 5 ms, making it much more suitable to be adopted for our caching algorithm.

	Base	Base+HF	DC	DC+HF
VGG-Verydeep-16(MB)	578	289	517	258.5
YOLO(MB)	1,116	558	1,002	501

"Base" indicates the original model.

Table 7: Memory Footprint

7.10 Memory Footprint

Table 7 shows the memory footprint for VGG-Verydeep-16 and YOLO. The memory usage is well within the available memory spaces of commodity mobile devices, showing that *DeepMon* manages its memory usage efficiently. Also, the decomposition and half-floating point approximation reduce the memory usage of *DeepMon*; they reduce the memory usage from 578MB and 1116MB down to 258.5MB and 501MB for VGG-Verydeep-16 and YOLO, respectively. For the models that require large memory spaces, other optimization techniques such as *Singular Value Decomposition* (SVD) [35] can be applied to further reduce the memory usage.

DeepMon mainly uses the memory to load the model and stores input and output of a layer. *DeepMon* stores the entire model within system memory for efficient inference since it is time-consuming to load the model on-demand from the external memory. On the other hand, *DeepMon* only stores input and output of the currently executing layer – it discards all output data from previous layers once they become of no use to keep memory usage as low as possible. Accordingly, memory usage of *DeepMon* is capped at the size of the model and the largest input and output size of a single layer.

8. CONCLUSION

In this paper, we present *DeepMon*, a system for enabling the low-latency execution of deep DNNs on a mobile GPU. *DeepMon* uses various optimisation techniques including the convolutional layer caching, decomposition, and matrix multiplication optimizations to achieve significant speedups (over 3-4x) compared to state-of-the-art current solutions. In particular, *DeepMon* allows DNN-based face/object detection models, such as VGG-VeryDeep-16 and YOLO, to process video frames at 1 to 2 frames per second. We implemented *DeepMon* in both OpenCL and Vulkan and tested it's effectiveness using three different mobile GPUs (Adreno 420, Adreno 430, and Mali T 880) and against alternative solutions such as state-of-the-art research systems (DeepX) and plausible cloud-based solutions. Our results show that *DeepMon* significantly outperforms all local-computation-only based solutions with a marginal drop in accuracy and that *DeepMon's* latency and accuracy combination can only be bettered by using very large (and expensive) cloud server instances with very good networking connectivity. Videos of *DeepMon* in action are available at http://is.gd/DeepMon. Also, *DeepMon*'s source code can be found at https://github.com/JC1DA/DeepMon.

9. ACKNOWLEDGEMENTS

We sincerely thank our shepherd, Matthai Philipose, for providing excellent iterative feedback that greatly improved our paper. We also thank the anonymous reviewers for their valuable comments. This research is supported by the National Research Foundation, Prime Minister's Office, Singapore under its IDM Futures Funding Initiative.

10. REFERENCES

[1] A brief report of the heuritech deep learning meetup 5. https://blog.heuritech.com/2016/02/29/a-brief-report-of-the-heuritech-deep-learning-meetup-5/. Accessed: 2016-12-8.

[2] Common objects in context. http://mscoco.org/. Accessed: 2016-12-8.

[3] Google glass. https://developers.google.com/glass/. Accessed: 2016-12-8.

[4] Hololens. https://www.microsoft.com/microsoft-hololens/en-us. Accessed: 2016-12-8.

[5] Monsoon power monitor. https://www.msoon.com/LabEquipment/PowerMonitor/. Accessed: 2016-12-8.

[6] Nvidia cuda toolkit. https://developer.nvidia.com/cuda-toolkit. Accessed: 2016-12-8.

[7] Opencl. https://www.khronos.org/opencl/. Accessed: 2016-12-8.

[8] Pascal visual object classes. http://host.robots.ox.ac.uk/pascal/VOC/. Accessed: 2016-12-8.

[9] Vulkan. https://www.khronos.org/vulkan/. Accessed: 2016-12-8.

[10] Balan, R. K., Gergle, D., Satyanarayanan, M., and Herbsleb, J. Simplifying cyber foraging for mobile devices. *Proceedings of the 5th International Conference on Mobile Systems, Applications and Services*, MobiSys '07, pages 272–285, New York, NY, USA, 2007. ACM.

[11] Balan, R. K., Satyanarayanan, M., Park, S. Y., and Okoshi, T. Tactics-based remote execution for mobile computing. *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services*, MobiSys '03, pages 273–286, New York, NY, USA, 2003. ACM.

[12] Boger, J., Hoey, J., Poupart, P., Boutilier, C., Fernie, G., and Mihailidis, A. A planning system based on markov decision processes to guide people with dementia through activities of daily living. *IEEE Transactions on Information Technology in Biomedicine*, 10(2):323–333, 2006.

[13] Chatfield, K., Simonyan, K., Vedaldi, A., and Zisserman, A. Return of the devil in the details: Delving deep into convolutional nets. *arXiv preprint arXiv:1405.3531*, 2014.

[14] Chellapilla, K., Puri, S., and Simard, P. High Performance Convolutional Neural Networks for Document Processing. In Lorette, G., editor, *Tenth International Workshop on Frontiers in Handwriting Recognition*, La Baule (France), Oct. 2006. Université de Rennes 1, Suvisoft. http://www.suvisoft.com.

[15] Chen, T. Y.-H., Ravindranath, L., Deng, S., Bahl, P., and Balakrishnan, H. Glimpse: Continuous, real-time object recognition on mobile devices. *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, SenSys '15, pages 155–168, New York, NY, USA, 2015. ACM.

[16] Cuervo, E., Balasubramanian, A., Cho, D.-k., Wolman, A., Saroiu, S., Chandra, R., and Bahl, P. Maui: Making smartphones last longer with code offload. *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 49–62, New York, NY, USA, 2010. ACM.

[17] Dalal, N. and Triggs, B. Histograms of oriented gradients for human detection. *Proceedings of 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 1 of *CVPR '05*, pages 886–893. IEEE, 2005.

[18] Denton, E. L., Zaremba, W., Bruna, J., LeCun, Y., and Fergus, R. Exploiting linear structure within convolutional networks for efficient evaluation. *Advances in Neural Information Processing Systems*, pages 1269–1277, 2014.

[19] Donahue, J., Anne Hendricks, L., Guadarrama, S., Rohrbach, M., Venugopalan, S., Saenko, K., and Darrell, T. Long-term recurrent convolutional networks for visual recognition and description. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '15, pages 2625–2634, 2015.

[20] Everingham, M., Eslami, S. A., Van Gool, L., Williams, C. K., Winn, J., and Zisserman, A. The pascal visual object classes challenge: A retrospective. *International Journal of Computer Vision*, 111(1):98–136, 2015.

[21] Everingham, M., Van Gool, L., Williams, C. K., Winn, J., and Zisserman, A. The pascal visual object classes (voc) challenge. *International Journal of Computer Vision*, 88(2):303–338, 2010.

[22] Girshick, R. Fast r-cnn. *Proceedings of the IEEE International Conference on Computer Vision*, pages 1440–1448, 2015.

[23] Grasedyck, L., Kressner, D., and Tobler, C. A literature survey of low-rank tensor approximation techniques. *GAMM-Mitteilungen*, 36(1):53–78, 2013.

[24] Ha, K., Chen, Z., Hu, W., Richter, W., Pillai, P., and Satyanarayanan, M. Towards wearable cognitive assistance. *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, pages 68–81, New York, NY, USA, 2014. ACM.

[25] Han, S., Shen, H., Philipose, M., Agarwal, S., Wolman, A., and Krishnamurthy, A. Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints. *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '16, pages 123–136, New York, NY, USA, 2016. ACM.

[26] He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[27] Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[28] Huynh, L. N., Balan, R. K., and Lee, Y. Deepsense: A gpu-based deep convolutional neural network framework on commodity mobile devices. *Proceedings of the 2016 Workshop on Wearable Systems and Applications*, WearSys '16, pages 25–30, New York, NY, USA, 2016. ACM.

[29] Jaderberg, M., Vedaldi, A., and Zisserman, A. Speeding up convolutional neural networks with low rank expansions. *CoRR*, abs/1405.3866, 2014.

[30] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. Caffe: Convolutional architecture for fast feature embedding. *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[31] Kim, Y., Park, E., Yoo, S., Choi, T., Yang, L., and Shin, D. Compression of deep convolutional neural networks for fast and low power mobile applications. *CoRR*, abs/1511.06530, 2015.

[32] Kim, Y.-D., Park, E., Yoo, S., Choi, T., Yang, L., and Shin, D. Compression of deep convolutional neural networks for fast and low power mobile applications. *arXiv preprint arXiv:1511.06530*, 2015.

[33] Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.

[34] Lane, N. and Bhattacharya, S. Sparsifying deep learning layers for constrained resource inference on wearables. *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems*, pages 176–189. ACM, 2016.

[35] Lane, N. D., Bhattacharya, S., Georgiev, P., Forlivesi, C., Jiao, L., Qendro, L., and Kawsar, F. Deepx: A software accelerator for low-power deep learning inference on mobile devices. *2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks*, IPSN '16, pages 1–12. IEEE, 2016.

[36] Lane, N. D., Georgiev, P., and Qendro, L. Deepear: Robust smartphone audio sensing in unconstrained acoustic environments using deep learning. *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '15, pages 283–294, New York, NY, USA, 2015. ACM.

[37] Lebedev, V., Ganin, Y., Rakhuba, M., Oseledets, I., and Lempitsky, V. Speeding-up convolutional neural networks using fine-tuned cp-decomposition. *arXiv preprint arXiv:1412.6553*, 2014.

[38] LiKamWa, R., Priyantha, B., Philipose, M., Zhong, L., and Bahl, P. Energy characterization and optimization of image sensing toward continuous mobile vision. *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, pages 69–82, New York, NY, USA, 2013. ACM.

[39] LiKamWa, R. and Zhong, L. Starfish: Efficient concurrency support for computer vision applications. *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '15, pages 213–226, New York, NY, USA, 2015. ACM.

[40] Mathieu, M., Henaff, M., and LeCun, Y. Fast training of convolutional networks through ffts. *CoRR*, abs/1312.5851, 2013.

[41] Morel, J.-M. and Yu, G. Asift: A new framework for fully affine invariant image comparison. *SIAM Journal on Imaging Sciences*, 2(2):438–469, 2009.

[42] Parkhi, O. M., Vedaldi, A., and Zisserman, A. Deep face recognition. *Proceedings of 2015 British Machine Vision Conference*, volume 1, page 6, 2015.

[43] Redmon, J., Divvala, S., Girshick, R., and Farhadi, A. You only look once: Unified, real-time object detection. *arXiv preprint arXiv:1506.02640*, 2015.

[44] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[45] Sainath, T. N., Kingsbury, B., Sindhwani, V., Arisoy, E., and Ramabhadran, B. Low-rank matrix factorization for deep neural network training with high-dimensional output targets. *Proceedings of 2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, ICASSP '13, pages 6655–6659. IEEE, 2013.

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

[47] Song, S., Chandrasekhar, V., Cheung, N.-M., Narayan, S., Li, L., and Lim, J.-H. Activity recognition in egocentric life-logging videos. *Asian Conference on Computer Vision*, pages 445–458. Springer, 2014.

[48] Soomro, K., Zamir, A. R., and Shah, M. Ucf101: A dataset of 101 human actions classes from videos in the wild. *arXiv preprint arXiv:1212.0402*, 2012.

[49] Urban, G., Geras, K. J., Kahou, S. E., Aslan, Ö., Wang, S., Caruana, R., Mohamed, A., Philipose, M., and Richardson, M. Do deep convolutional nets really need to be deep (or even convolutional)? *CoRR*, abs/1603.05691, 2016.

[50] Vanhoucke, V., Senior, A., and Mao, M. Z. Improving the speed of neural networks on cpus. *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*, 2011.

[51] Vedaldi, A. and Lenc, K. Matconvnet: Convolutional neural networks for matlab. *Proceedings of the 23rd ACM International Conference on Multimedia*, pages 689–692. ACM, 2015.