Are Very Deep Neural Networks Feasible on Mobile Devices?

S. Rallapalli, H. Qiu, A. J. Bency, S. Karthikeyan, R. Govindan, B.S.Manjunath, R. Urgaonkar

In the recent years, the computing power of mobile devices has increased tremendously, a trend that is expected to continue in the future. With high-quality onboard cameras, these devices are capable of collecting large volumes of visual information. Motivated by the observation that processing this video on the mobile device can enable many new applications, we explore the feasibility of running very deep Convolutional Neural Networks (CNNs) for video processing tasks on an emerging class of mobile platforms with embedded GPUs. We find that the memory available in these mobile GPUs is significantly less than necessary to execute very deep CNNs. We then quantify the performance of several deep CNN-specific memory management techniques, some of which leverage the observation that these CNNs have a small number of layers that require most of the memory. We find that a particularly novel approach that offloads these bottleneck layers to the mobile device's CPU and pipelines frame processing is a promising approach that does not impact the accuracy of these tasks. We conclude by arguing that such techniques will likely be necessary for the foreseeable future despite technological improvements.

1. INTRODUCTION

Smart mobile devices, e.g., smartphones, tablets, wearables, are being adopted by users at an unprecedented rate. Moreover, the capabilities of cameras on these devices has also been increasing, to the point where these devices can generate high volumes of visual data in the form of video and images. Video and image data is semantically rich and being able to process this data in near real-time on the mobile device (i.e., without incurring the latency of off-loading processing to the cloud) could be immensely useful for several applications: in a military scenario to catch suspicious activities, or in a civilian scenario like localized advertising, or personal assistants.

In the last few years, computer vision techniques have been revolutionized by the development of Convolutional Neural Networks (CNNs) which can be trained for classification, detection, segmentation, and localization tasks for high accuracy. Being able to perform complex computer vision tasks in-situ on a mobile device is now within the realm of feasibility, with the emergence of a new class of mobile processors that includes an integrated GPU with hundreds of cores [29]. Many CNNs can be run on these GPUs [1].

In recent years, the computer vision community has been developing *very deep* CNNs. Such CNNs exhibit high accuracy in complex detection tasks by using tens of convolutional layers (Section 2): intuitively, the large number of layers enables the network to learn finer features and thereby improves accuracy. Examples of these CNNs include YOLO [26] for object detection (i.e., classification as well as localization of objects), FCN [19] for image segmentation, and VGG [27] for object recognition (i.e., classification). These have between 19 and 26 layers, and have so far been demonstrated to work only on server-class machines.

Our contributions: In this paper, we explore the feasibility of running very deep CNNs on emerging mobile GPUs. We assume that these CNNs will still be trained on server-class machines, and focus on using CNNs for computer vision tasks. To do this, we take a recently developed very deep CNN for object detection, YOLO [26] (Section 2). As shown in Figure 1, YOLO not only classifies the object but also draws bounding box around it. This is a useful primitive in tracking objects in a video (i.e., detecting objects across successive frames): once an object's bounding box has been determined, lightweight trackers can be used to achieve this.

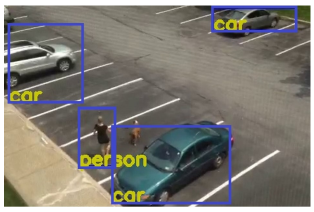

Figure 1: Output of YOLO object detection: classification + localization (Accurate, except the very small dog is not detected).

We find that a primary bottleneck in running very deep CNNs is *memory* (Section 3). The mobile GPU we use, the TK1 [29], has 2 GB of memory, but YOLO fails to run on this device for lack of memory. In part, this is because the mobile GPU is physically addressed and lacks memory management hardware. We also find that one of the fully-connected (FC) layers (these last layers perform the detection and localization) requires almost 80% of the memory required by the CNN.

We then explore a range of memory optimizations (Section 3), some of which leverage CNN structure and properties of YOLO, and quantify their performance (Section 4). These optimizations include: pruning unused variables in the neural network, using "managed" memory in the GPUs, splitting the FC layer into sub-parts each of which is loaded onto memory and executed sequentially, trading-off accuracy by reducing the size of the neural network, and, in a novel twist, offloading the FC layer to the CPU while pipelining CPU and GPU computation.

We find (Section 4) that many of these techniques make it feasible to run YOLO on the TK1 and we are able to process about 3-4 frames per second. Splitting the CNN can incur overheads in loading the sub-parts, and is less effective than CPU offloading with pipelining. Reducing the size of the CNN can reduce accuracy by 3-4%.

We conclude by making the case that such memory optimizations will be needed in the foreseeable future (Section 5). Very deep CNNs are fast becoming the norm, and many of these have memory intensive FC layers. Moreover, the increase in mobile GPU memory is likely to be limited by energy budgets, and developing high-performance virtual memory across CPUs and GPUs may take a few years. Even if these advances accelerate faster than expected, there is still an order of magnitude gap between the frame rate we can achieve today and full-motion video, and our memory optimizations will be required to bridge this gap.

2. BACKGROUND

A Primer on Convolutional Neural Networks: Modern computer vision classification and detection tasks increasingly use deep neural networks. Deep neural networks are typically arranged in layers, with an input layer, an output layer and several hidden layers in between. Usually raw data (e.g., pixels) are fed into the input layer, and the output layer emits the corresponding class or type of object, and additional information (such as the location in an image of a detected object). Each layer in the network contains a number of nodes, and different layers can contain different weights. Links exist between nodes at one layer and the next, and each link is associated with a *weight*.

Each node at a layer processes its input according to a pre-defined function, then transmits a weighted output (based on the link weight between the node pairs) to a set of nodes at the next layer. In turn, nodes at the next layer repeat this process until the output layer is reached. These link weights are learned by training the neural network on a large set of labeled data. Intuitively, each successive layer of the neural network learns higher-level features within an image. In general, the larger the network, i.e., more layers and more nodes per layer, the better the task accuracy. Figure 2 shows 2 layers of deep neural network.

Traditional neural networks involved all layers to be fully connected (Figure 2, left). Learning weights in these networks can be time consuming. Moreover, in cases where the same object resides in different parts of the image, the learned weights can be redundant: two different parts of the neural network would have to learn the same weight for the same object.

Convolutional neural networks (CNNs) were proposed to solve this problem and are predominantly used in machine vision today. For example, in Figure 2 (right), there are only 9 connections between the layers. Furthermore, some of these weights are the same, e.g., in the figure, 3 red dashed lines (w_1), 3 blue dash-dotted lines (w_2) and 3 green dotted lines (w_3) are each same weight. This is because in a CNN, the network learns a filter which is applied at all places in an image by sliding it throughout. For accuracy, several sets of filters are used, but notice that for each filter, it is the same operation applied everywhere on the image. This helps achieve sparsity as well as efficiency. In the figure by applying a filter $< w_1, w_2, w_3 >$ to nodes (A, B, C) gives P, to nodes B, C, D gives Q and so on. The complete CNN is formed by alternating convolutional layers with sub-sampling layers for efficiency. After multiple such alternating layers, we arrive at a set of high-level features on which *fully connected (FC) layers* are applied to perform the final detection task.

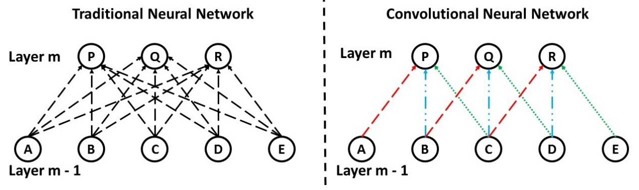

Figure 2: Neural Network Architecture.

Mobile device GPUs: In the near future, it will be possible to run convolutional neural networks on mobile devices, thanks to the recent development of mobile processors with integrated GPUs. An example of such a mobile processor is the Tegra K1, manufactured by nVidia, which contains 192 GPU cores and conforms to the Compute Unified Device Architecture (CUDA) specification. CUDA-capable GPUs can be programmed using a high level programming language that permits easy specification of parallelism. The Tegra K1 has already been incorporated into tablet-class devices such as the Nexus 9 [21] and the Google Tango [28]. In this paper, we use, for our evaluations, the Jetson TK1 board from nVidia (Figure 3), which contains the Tegra K1 mobile processor and an associated CUDA-capable development environment.

Figure 3: nVidia Jetson TK1 board.

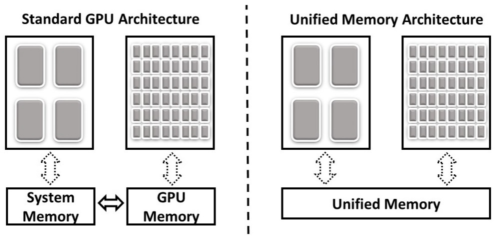

Figure 4: GPU architecture [32].

The memory architecture of CUDA 6 devices, like the TK1, can significantly impact the performance of convolutional neural networks on mobile devices, as we discuss in the rest of the paper. Traditional GPU architectures involve separate GPU and CPU memory, with the programmer having to manage copying between these to perform GPU operations. CUDA 6 devices have unified memory, so the CPU and GPU can share memory, resulting in programming simplicity (Figure 4). However, relative to the memory requirements of very deep convolutional neural networks, devices like the TK1 have limited memory (2GB). A newer mobile GPU platform – TX1 [30] has 4GB memory. However, with vision applications becoming increasingly complex (FCN [19] for semantic segmentation requires 4.78GB memory), the gap between what we would *like* to achieve and what we *can* achieve given the current hardware will remain for the foreseeable future.

Furthermore, the memory architecture in these devices requires the GPU to physically address memory: in other words, GPUs do not have virtual memory and demand paging [12, 17]. So, GPU-based applications that need more memory than is available need to explicitly manage memory usage in order to even be able to execute on these devices. In Section 5, we discuss whether this situation is likely to persist in the future.

Related work: Our work is related to two different lines of research namely: (1) systems issues in running deep neural networks (DNNs) on mobile devices, and (2) detection algorithms proposed in the vision literature.

DNNs on mobile devices: Despite the potential of neural networks to improve the accuracy of analytics tasks, until recently they have been troublesome to train even on server-class machines. Recent advances in GPUs and parallelism have led to the revival of this area. However, till date, very little work has focused on running neural networks on mobile platforms. One pioneering work that explores running deep neural networks on smartphone CPUs is [15]. The network they consider uses a total of 3 layers and is focused on activity detection, emotion recognition and speaker identification using accelerometer and microphone. As the authors point out, it is much simpler than the networks required for image recognition tasks. [16] is another paper from the same group that explores the performance of audio sensing on smartphones using DNNs. The DNN network they consider is smaller than the ones we require for object detection. Another work [1] has benchmarked smaller Caffe-generated CNN (AlexNet) [14], with 5 convolutional layers and 3 FC layers, for vision-based classification-only (no localization) on the TK1, for which they report 30 fps and we have also confirmed this in our experiments. Since none of these works use *very deep* CNNs, they did not have to deal with the issues related to memory constraints. In Section 5, we show that CNNs for some vision tasks can have 20 layers or more. This requires the design of sophisticated techniques to address the resulting bottlenecks, a focus of our work.

Object detection: Object detection involves: (1) classification of objects, and (2) localizing them and drawing bounding boxes as shown in Figure 1. CNN based classification-only schemes achieve high accuracy at real time speeds [11], but top detection systems like R-CNN [8], Fast R-CNN [9] (which have localization capability in addition to classification) exhibit less than real-time performance even on server-class machines. This is because detection is a much harder problem than classification and many detection systems repurpose classifiers to perform detection by evaluating different locations and scales of objects to be able to localize them on the image. Recent approaches like R-CNN use region proposal methods to first generate bounding boxes in an image and then run a classifier on these bounding boxes. Selective search [31] is the most commonly used region proposal method but takes 1-2 seconds per image to generate the proposal boxes even on server-class machines.

In contrast to these algorithms for object detection, YOLO [26] is a one shot CNN-based detection algorithm that predicts bounding boxes and classifies objects. It is very fast, running at 45 fps on an nVidia Titan X server-class GPU with 3072 cores and achieves a reasonable accuracy. It is much faster than existing detection algorithms. Table 2 in [26] summarizes the state-of-the-art detection schemes that we do not repeat here for brevity. The important point to note is that, while YOLO is not the most accurate detection algorithm, it runs very fast with reasonable accuracy, and thus lends itself to be run on mobile platforms. Hence, in this paper, we explore the feasibility of using mobile GPUs like the TK1 on a very deep CNN, like YOLO.

3. VERY DEEP CNNS ON MOBILE GPUS

In this section, we explore the performance issues in executing very deep CNNs on mobile GPUs like the TK1 by specifically focusing on YOLO [26].

Details of YOLO: YOLO is a very deep CNN for performing object detection. It is structured as a 27 layered CNN, with 24 convolutional layers, followed by 2 FC layers and a detection layer. The convolutional layers are responsible for extracting the features that best suit the vision task, while the FC layers use these features to predict the output probabilities and the bounding box coordinates.

Both, convolutional layers as well as the FC layers involve a set of matrix multiplications. In the case of convolutional layers these multiplications simply convolve the filters with the output of the previous layer. In the case of FC layers, these multiplications weigh the features appropriately and map them to the output probabilities. We have already seen a FC layer in Figure 2 (left) while looking at traditional DNNs, because in traditional DNNs all layers are fully connected, whereas in the case of CNNs only the last layer(s) are fully connected. Suppose that in Figure 2 (left) P, Q, R are required outputs, and A - E are the outputs of the final convolutional layer, then $P = w_1 * A + w_2 * B + w_3 * C + w_4 * D + w_5 * E$, $Q = w_6 * A + w_7 * B + w_8 * C + w_9 * D + w_{10} * E$, and so on. The detection layer simply extracts the output of the FC layer into a usable format.

The YOLO network requires 2.8GB memory on TK1 using regular GPU programming workflow described below (after removing allocations that are only required in the training phase). TK1 has physically unified memory, so this includes both CPU as well as GPU memory requirement. Almost 80% the memory allocated to the network parameters, is taken by the first FC layer (also observed by other works [13]). The detailed original YOLO network structure is explained in [26] and uses 448×448 resolution for video frames. The network is *trained offline* on server-class machines: in this paper, we focus on *using the network on mobile GPUs to perform object detection.*

Since GPU is a slave device and CPU is the master, a normal GPU programming workflow involves loading the data to be operated upon (in our case the trained CNN weights) first in CPU RAM, followed by copying them to GPU memory and starting the kernel operation. If we had enough resources to hold the entire CNN model in memory, then multiple video frames can be dispatched for processing in sequence and the computation is very fast. The kernel operation is asynchronous and kernels are designed so that they can operate in parallel on the data and compute different parts of the result. This allows operations on large arrays and matrices to be done very quickly.

However, when we ran YOLO on the TK1 using the workflow described above, it *failed to execute* due to insufficient memory resources. Recall that, on the TK1, memory is shared between the CPU (which runs a full-fledged Linux OS) and the GPU. This motivated us to explore several *application-specific* methods to manage the memory resource constraint on mobile GPUs. In the following paragraphs, we describe these methods. In Section 5, we discuss whether these methods will continue to be relevant in future mobile GPUs and for other very deep CNNs.

Basic memory optimizations: For mobile GPUs, two basic memory optimizations can reduce memory usage. In server-class systems, where memory is not as restricted, it is common to allocate the same neural network for training as well as testing phases, even though not all functionality is required in the testing phase. Specifically, training of neural networks happens through several iterations of forward and backward propagation to learn and refine the weights. Since we aim to only run the testing phase on the mobile devices, we never use back-propagation or the variables used in the process of refining the weights. One optimization we use is to not allocate memory for any variables not used during the testing phase. After doing this, the original YOLO network that required memory as high as 4.6 GB (recall that for TK1 this is GPU + CPU due to physically unified memory), now requires only 2.8 GB. After this point, throughout the paper, we only refer to the memory requirement after removing the un-used variables.

Further, since TK1 follows the Unified Memory Architecture, we have two options for memory management. One approach (discussed in the workflow above) is to allocate memory for the CNN on the CPU, then invoke a `cudaMalloc()` function which performs a deep-copy (adjusts pointers during copy) of the CNN for use by the GPU. A more memory-efficient technique is to use *managed* memory allocation. This allocates memory for a single copy of the CNN, but creates the data structure in such a way that it is accessible both by a virtually-addressed CPU, and a physically-addressed GPU. It was primarily introduced to ease the programmer burden and enable fast prototyping of GPU accelerated algorithms. While it optimizes memory usage and programming efforts, it can potentially slow down GPU performance [22] due to memory translation over-heads. So we evaluate both techniques in Section 4.

Sacrificing accuracy: As mentioned above, the FC layers are the ones that require most memory in a CNN. Thus we *reduce the size* of the FC layer, which can potentially reduce detection accuracy while addressing the memory constraint ([2] explored different configurations of neural networks to reduce memory footprint for a different application: keyword spotting). To reduce the size of the FC layer, we reduce the number of filters in the last convolutional layer (from 1024 (original) to 512 (medium) to 256 (small)). We also reduce the number of outputs (can be thought of as number of nodes) of the first FC layer from 4096 (original) to 2048 (medium) to 1024 (small). After reducing the size, we re-train the new networks. In Section 4 we explore the trade-off between accuracy and running time for different network sizes.

The default YOLO detection network is able to recognize 20 classes of objects, e.g., person, car, dog, bicycle, etc. One way to avoid a potential loss of detection accuracy when the network size is reduced is to also *reduce the number of objects detected by the network*. When this trade-off is feasible (i.e., when an application requires only a subset of the 20 classes), it is a plausible hypothesis that this approach might be able to overcome the memory constraint without reducing detection accuracy. In Section 4, we test this hypothesis.

Split computation in the fully connected layers: In YOLO, the FC layer is a memory bottleneck. To reduce the memory footprint of this layer, we can split the matrix multiplication performed in the FC layer [20], as shown in Figure 5. Specifically, in the FC layer, the input vector of previous layer is multiplied by the weight matrix to obtain the output vector. Suppose the input vector is of size 1×2 and weight matrix is of size 2×2. Suppose we split the computation into two parts, then we can read the weight matrix into memory row by row and multiply with the corresponding element in the input vector. Thus, in Figure 5, the blue parts are computed first, then green parts, and the results are combined. Reading the weight matrix in parts allows us to re-use memory. However, re-using memory and over-writing the used chunks incurs additional file access overheads. We evaluate the impact of this in Section 4.

$$[\boldsymbol{I_1} \quad \boldsymbol{I_2}] \begin{bmatrix} \boldsymbol{W_1} & \boldsymbol{W_2} \\ \boldsymbol{W_3} & \boldsymbol{W_4} \end{bmatrix} = [\boldsymbol{I_1 W_1} + \boldsymbol{I_2 W_3} \quad \boldsymbol{I_1 W_2} + \boldsymbol{I_2 W_4}]$$

Figure 5: Split the matrix multiplication.

CPU offload: While computational offload techniques have been extensively explored [3, 7], we explore a novel form of off-loading motivated by memory constraints: offloading the FC layer computation to the CPU. This offloading technique is not motivated by processor speed considerations, but by the fact that, because the CPU has built-in memory management and demand paging, it can more effectively manage memory over-subscription required for the FC layer.

CPU offload with pipelining: With CPU offloading, we observe that when the CPU is running the FC layer on the first video frame, the GPU cores are idle. So we adopt a pipelining strategy to start running the second video frame on the GPU cores rather than letting them idle. To achieve this kind of pipelining, we run the FC layers on the CPU on a separate thread, as GPU computation is managed by the main thread. This gives us a speed up in processing as we shall see in Section 4.

Speed optimizations: In addition to memory optimizations, our results in the next section incorporate two optimizations for speeding up the computation (recall that, ultimately, our goal is to process frames at as close to full frame rate as possible). First, mobile GPUs are typically set to a lower clock-rate to save power and this clock-rate is variable and ramped up based on the GPU usage. In our case, before the image processing starts, we simply set the GPU clock-rates to the highest supported to ensure best performance. This approach trades off power for increased computation speed. Second, a few CUDA operations incur significant latency when they are invoked for the first time. For example, cuBLAS [5] parallelizes the matrix operations and requires a handle to be created the first time around which takes time. Similarly, random number generation requires initializing the state, which can add a one-time latency. We perform these operations before running the CNN to reduce latency.

4. RESULTS

Methodology: In this section, we evaluate the various performance optimization strategies for object detection discussed in Section 3. Our results focus on three metrics: time to detect objects on a single frame (or, *detection time*), the energy consumed per frame (*detection energy*), and the *detection accuracy*. For time and energy results, we run 25 frames through each strategy and report the average per frame.

Detection time is measured by evaluating elapsed time on the CPU. To measure detection energy, we use the setup shown in Figure 6. We use a CA 60 current clamp [4] and clamp it to the positive wire of the power supply to the TK1 board. By inducing a magnetic field on the conductor and making use of Hall effect sensor, we can compute the current passing through. We use a DataQ DI-149 Data Acquisition Kit [6] that samples at 80 Hz and allows us to save the readings in a file to post-process later.

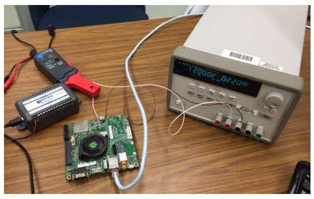

Figure 6: Power measurement set-up.

Detection accuracy is measured as mean average precision (mAP), which captures false positives as well as the false negatives. Precision is the fraction of objects correctly detected out of the total detected objects. Recall is the fraction of objects detected correctly out of all the objects present in the frame. Average precision computes the average of precision across different values of recall. mAP is average precision over all classes (in our case 20: person, dog, car, etc.).

Performance with no accuracy loss: In Table 1 we summarize the timing results of running the various strategies. We explain

	Memory Requirement (GB)	FC layer: resulting size of matrix multiplication	CPU	GPU	Split	CPU Offload	Pipeline
Original	2.8	1 × 50176 * 50176 × 4096	25.026524	N/A	10.249577	0.703449	0.416910
Medium	1.6	1 × 25088 * 25088 × 2048	24.315950	0.272191	0.573386	0.400366	0.261985
Small	1.3	1 × 12544 * 12544 × 1024	24.003447	0.259000	0.394960	0.299940	0.261144

Table 1: Average time taken to run detection per image (seconds).

the results starting from the first row, i.e., *Original network*. This network is already the optimized version which does not allocate memory for redundant variables not used in the testing phase. We make several observations. First, running the computation on CPU takes close to *half a minute* per frame, which is extremely slow, so it is imperative to leverage the GPU cores on the board for accelerated computation. Second, we see that for the original network, the memory restrictions do not allow running it on GPU at all; our subsequent optimizations to reduce the memory foot-print make this possible. Third, splitting the matrix multiplication of the FC layer into 4 parts (*Split*), as explained in Section 3 allows us to run the original network using the GPU cores – it gives much better performance than running on the CPU by bringing the running time to 40% of that on CPU, but still has a large running time of 10s per frame. As explained in Section 3 this is because splitting incurs file access overhead. Fourth, CPU offload brings the computation time to under 1s. Using the CPU for the FC layer allows us to read weight file *only once* for all the frames and store the weights in CPU memory (which supports demand paging). Finally, CPU offload along with pipelining gives the best results; it brings down the computation time to about 0.42s or almost *60 times faster* than running it on the CPU. We see this speed up because although FC layer is memory intensive, the running time even on CPU is not a bottle-neck, in other words, the CPU processing completes within the time that the GPU is processing the next frame. This is interesting because it is achieved without compromising on accuracy.

Trading off accuracy: Next, we reduce the size of the network, by reducing the size of the weight matrix in the FC layer to one fourth of the original size (50176 × 4096 to 25088 × 2048). This *Medium network* (row 2, Table 1) shows a similar trend as the original network with a few differences. The network now fits into GPU memory and can be run fully in the GPU in about 0.27s. Split takes considerably less time than for Original network because the size of the weights to be read is 1/4-th of the original, so the file read overhead incurred per frame is much smaller. CPU offloading along with pipelining is again the best scheme with 0.26s running time. This is about 93 times faster than running the network in the CPU.

Finally, we try a network whose FC layer weight matrix is 1/16 the size of the original network (12544 × 1024) and call it *Small network*. The result trend looks similar to the medium network described above, with one exception: the GPU is now the fastest alternative. This is because FC layer is very small and the overhead of pipelining negates its benefits at this point: this overhead includes (1) thread creation for CPU computation and (2) making the output of the final convolutional layer on the GPU accessible to the CPU. Further reducing the network size shows diminishing returns for running time, so we stop trying smaller networks.

How do these networks degrade accuracy? We evaluate the accuracy on the Pascal VOC 2012 test data-set [23], which is commonly used in the vision community. For the networks we trained (on over 21K images from Pascal VOC 2007 and 2012 train and validation data-sets as well as 2007 test data-set) we obtained 48.3%, 46.1%, 42.9% for the Original, Medium and Small networks respectively. With some changes in training parameters, we believe we can achieve about 58.8% for Original network as this is reported in [26] Table 2. But in this paper, since we focus on the degradation in accuracy as a function of size, we do not worry about the absolute numbers. The key take-away here is that by reducing the connected layer every time to 1/4th, we lose about 2-3% mAP. For applications for which this degradation is acceptable, reducing network size is a good tool in a system designer's toolbox to achieve good detection speed.

Can fewer classes help accuracy? As mentioned in Section 3, YOLO by default works with 20 classes. One hypothesis we tested is that could we do with smaller network sizes if we were to only detect fewer classes. We found that at a particular size of network, working with 10 classes instead of 20 did not buy us accuracy. Hence did not pursue this path further.

The role of managed memory: Finally, in Table 2 we change the memory allocation of YOLO to use CUDA managed memory. This is better for memory efficiency as only one copy of data is required as opposed to keeping 1 in CPU and 1 in GPU, so memory requirement reduces as reflected in the table. However, managed memory increases detection times by around 25%: for e.g., for the medium network, the detection time increases from 0.27s to 0.33s under the GPU only strategy. This increased overhead comes from the fact that managed memory has to present a single memory view to both the virtually addressed CPU and the physically addressed GPU. [10] explains that CPU processing of GPU memory mappings is expensive. Further, [22] nVidia itself recommends using traditional host-to-device, device-to-host transfers (in many cases) for best performance. Notice two exceptions here. First, unlike other schemes, Split does better with managed memory, since the read overheads are smaller (as one copy is avoided). Second, unlike without managed memory, the Original network (after removing redundant variables) now runs entirely on the GPU. However, at this point it is using almost all of the available memory. If we run a slightly bigger network (*Large network*), with a 20% larger FC layer than the Original network, this fails to run on the GPU but has a detection time of less than 0.5s using pipelining.

	Memory Req. (GB)	GPU	Split	CPU Offload	Pipeline
Large	1.8	N/A	11.610	0.818	0.489
Original	1.6	0.407	10.041	0.746	0.423
Medium	1	0.328	0.545	0.449	0.284
Small	0.8	0.315	0.410	0.344	0.268

Table 2: Average time taken to run detection per image (seconds) for CUDA managed allocation.

Power measurement: The detection energy for the various schemes is shown in Table 3. For brevity we only show the measurements from the regular GPU programming work-flow (corresponding to the timing results in Table 1). For the Original network, we see that CPU offloading based schemes are also the best from energy point of view. They bring down the energy requirement by more than 27 times. Furthermore, we also see that smaller CNNs do save energy, but this is at the expense of accuracy as explained above.

	CPU	GPU	Split	CPU Offload	Pipeline
Original	106.9	N/A	17.5	3.78	3.95
Medium	103	2.49	3.23	2.59	2.25
Small	101.4	2.34	2.64	2.36	2.03

Table 3: Average energy consumed for detection per image (J).

5. DISCUSSION

Name (# Layers)	Task	Total Memory (GB)	FC Mem. Share
FCN 32 s (16)	Semantic Segmentation	4.78	89%
VGG 19 (19)	Object recognition	1.34	86%
VGG 16 (16)	Object recognition	1.24	89%

Table 4: CNNs with large memory requirements.

In the previous section, we have evaluated a single very deep CNN on a single mobile GPU. To understand the generality of our results, we need to address two questions: are there other very deep CNNs and does their structure resemble that of YOLO? Relative to these CNNs, will future mobile GPUs be memory constrained, and will they also continue to be physically addressed (i.e., without memory management hardware)?

Given their reported efficacy, it is likely that deep CNNs are here to stay, and will be used for increasingly complex recognition tasks, so their memory requirements will increase in the coming years. Early CNNs used 5 convolutional layers and 3 fully connected layers [14] for image classification. YOLO uses 24 convolutional and 2 connected layers for object detection and requires about 2GB. Table 4 lists a few other large CNNs, which have between 16 and 19 layers, and require between 1.3GB and nearly 5GB. Interestingly, these also have the same property as YOLO: their fully connected layers account for nearly 90% of the overall memory allocated to the network parameters. This shows that all our memory optimizations are likely to remain relevant in the future, including the ones that rely on the FC-layers being the memory bottleneck.

We also believe mobile GPUs are unlikely to have enough memory to accommodate the larger CNNs in Table 4. Although newer mobile GPU hardware (e.g., TX1 [30]) has more memory, mobile GPU memory increases are limited by energy budgets: as [18] points out, DRAM power can account for up to 30% of the total power consumption of smartphones. Future mobile GPUs may get virtual memory hardware. Designing unified virtual memory across heterogeneous processors is still an area of active research [24, 25], and although standards are emerging, the performance of these unified virtual memories (e.g, the costs of TLB and cache invalidation, memory coherence, and address resolution, which are about 10-15% for today's CPUs [24]) are as yet unknown. Even if these techniques exhibit good performance, we believe that application-specific memory management techniques for very deep CNNs, like ours, will continue to be useful for several years, especially since there is a big gap between where we are today (we can process 3-4 frames per second with our most efficient techniques), and the eventual goal (of processing full-motion video at 30 fps).

6. CONCLUSION

Motivated by the advent of powerful mobile GPUs, we explore the feasibility of running very deep neural networks on them for the first time. We find that, relative to the requirements of very deep CNNs, mobile GPUs are memory constrained. We explore several CNN-specific memory management techniques starting with basic memory optimizations, varying sizes of neural networks, splitting the computation to re-use memory, and, using the CPU together with the GPU cores on the board. Interestingly, for some memory intensive tasks it actually is useful to leverage the CPU power on the mobile platforms as they support demand-paging and over-subscription of memory.

With this, we have just scraped the surface, and a lot remains to be done to understand if general purpose video analytics can be run in real-time on mobile devices. For e.g., can we use frequency domain to further improve running time? Convolution in time domain becomes multiplication in the frequency domain, which can be computed much faster due to reduced number of operations. However, studying the trade-off is interesting as this technique involves the over-head of computing fourier transform and the inverse fourier transform. Further, how to use the vision tasks enabled by this paper in an end-to-end system to perform object detection followed by tracking is an open question that will allow us to study how often we really need to run deep CNNs in a continuous video stream.

Acknowledgements

This research is supported by the Network Science Collaborative Technology Alliance sponsored by the US Army Research Laboratory W911NF-09-2-0053. We thank the author of YOLO paper, Joseph Redmon for his tremendous inputs. We would also like to sincerely thank Catherine Crawford, Nirmit Desai, Rajesh Bordawekar for making the necessary hardware available to us and also helping us with many useful discussions.

7. REFERENCES

[1] Caffe on TK1. http://goo.gl/6hgbM6.

[2] G. Chen, C. Parada, and G. Heigold. Small Footprint keyword spotting using deep neural networks. In *Proc. of IEEE ICASSP '14*.

[3] D. Chu, N. D. Lane, T. T.-T. Lai, C. Pang, X. Meng, Q. Guo, F. Li, and F. Zhao. Balancing energy, latency and accuracy for mobile sensor data classification. In *Proc. of Sensys '11*.

[4] Current Clamp. http://pdimeters.com/products/Accessories-Parts/PDI-CA60.php.

[5] cuBLAS. http://docs.nvidia.com/cuda/cublas/#axzz3nbfSUkXx.

[6] Data Acquisition Kit. http://www.dataq.com/products/di-149/.

[7] H. Eom, P. St.Juste, R. Figueiredo, O. Tickoo, R. Illikkal, and R. Iyer. Machine learning-based runtime scheduler for mobile offloading framework. In *Proc. of UCC '13*.

[8] R. Girshick, J. Donahue, T. Darrell, and J. Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proc. of IEEE CVPR '14*.

[9] R. B. Girshick. Fast R-CNN. *CoRR*, abs/1504.08083, 2015.

[10] J. Hestness, S. W. Keckler, and D. A. Wood. Gpu computing pipeline inefficiencies and optimization opportunities in heterogeneous cpu-gpu processors. In *Proc. of IISWC '15*.

[11] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[12] J. Kehne, J. Metter, and F. Bellosa. Gpuswap: Enabling oversubscription of gpu memory through transparent swapping. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '15, pages 65–77, New York, NY, USA, 2015. ACM.

[13] A. Krizhevsky. One weird trick for parallelizing convolutional neural networks. *CoRR*, abs/1404.5997, 2014.

[14] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proc. of NIPS '12*.

[15] N. D. Lane and P. Georgiev. Can deep learning revolutionize mobile sensing? In *Proc. of HotMobile '15*.

[16] N. D. Lane, P. Georgiev, and L. Qendro. Deepear: robust smartphone audio sensing in unconstrained acoustic environments using deep learning. In *Proc. of UbiComp '15*.

[17] J. Lee, M. Samadi, and S. Mahlke. VAST: The Illusion of a Large Memory Space for GPUs. In *Proc. of ACM PACT'14*.

[18] S. Liu, K. Pattabiraman, T. Moscibroda, and B. Zorn. Flikker: Saving dram refresh-power through critical data partitioning. In *Proc. of ACM ASPLOS '11*.

[19] J. Long, E. Shelhamer, and T. Darrell. Fully convolutional networks for semantic segmentation. In *Proc. of IEEE CVPR '15*.

[20] M.-M. Moazzami, D. E. Phillips, R. Tan, and G. Xing. ORBIT: A Smartphone-based Platform for Data-intensive Embedded Sensing Applications. In *Proc. of ACM IPSN '15*.

[21] Nexus 9. `https://www.google.com/nexus/9/`.

[22] nVidia UMA Overhead. `https://devtalk.nvidia.com/default/topic/754874/tk1-memory-bandwidth/`.

[23] Pascal VOC. `http://host.robots.ox.ac.uk/pascal/VOC/`.

[24] B. Pichai, L. Hsu, and A. Bhattacharjee. Architectural support for address translation on gpus: Designing memory management units for cpu/gpus with unified address spaces. In *Proc. ACM ASPLOS '14*.

[25] J. Power, M. D. Hill, and D. A. Wood. Supporting x86-64 address translation for 100s of GPU lanes. In *Proc. of IEEE HPCA '14*.

[26] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. *CoRR*, abs/1506.02640, 2015.

[27] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In *Proc. of ICLR '15*.

[28] Google Tango. `https://www.google.com/atap/project-tango/`.

[29] nVidia Jetson TK1. `https://developer.nvidia.com/jetson-tk1`.

[30] nVidia Jetson TX1. `http://www.nvidia.com/object/jetson-tx1-module.html`.

[31] J. Uijlings, K. van de Sande, T. Gevers, and A. Smeulders. Selective search for object recognition. *IJCV*, '13.

[32] Unified Memory Architecture. `http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/`.