Augur: Modeling the Resource Requirements of ConvNets on Mobile Devices

Zongqing Lu, *Member, IEEE*, Swati Rallapalli, *Member, IEEE*, Kevin Chan, *Member, IEEE*, Shiliang Pu, and Thomas La Porta *Fellow, IEEE*

Abstract—Convolutional Neural Networks (ConvNets/CNNs) have revolutionized the research in computer vision, due to their ability to capture complex patterns, resulting in high inference accuracies. However, the increasingly complex nature of these neural networks means that they are particularly suited for server computers with powerful GPUs. We envision that deep learning applications will be eventually widely deployed on mobile devices, *e.g.*, smartphones, self-driving cars, and drones. Therefore, in this paper, we aim to understand the resource requirements of CNNs on mobile devices in terms of compute time, memory and power. First, by deploying several popular CNNs on different mobile CPUs and GPUs, we *measure* and *analyze* the performance and resource usage for the CNNs on a layerwise granularity. Our findings point out the potential ways of optimizing the CNN pipelines on mobile devices. Second, we *model* resource requirements of core computations of CNNs. Finally, based on the measurement, and modeling, we build and evaluate our modeling tool, *Augur*, which takes a CNN configuration (descriptor) as the input and estimates the compute time, memory, and power requirements of the CNN, to give insights about whether and how efficiently a CNN can be run on a given mobile platform.

Index Terms—Convolutional neural networks, mobile devices, modeling.

1 Introduction

Deep learning has become the norm of state-of-the-art learning systems, especially in computer version. Convolutional Neural Networks (ConvNets/CNNs) have demonstrated impressive performance on various computer vision tasks from classification and detection to segmentation and captioning. A CNN consists of different types of layers (*e.g.*, convolutional, pooling, fully connected), where each layer performs certain transformation on the input data and outputs the results to the next layer. Different CNNs for computer vision tasks have been designed, from a few layers to a thousand layers. The core of these networks are the convolutional layers, which consist of a set of learnable kernels that are convolved across the length and width of the input image to produce output features. There are several frameworks that support the training (forward and backward pass) and inference (only forward pass) phases of CNNs, including Caffe [1], TensorFlow [6], Torch [8], Theano [7], *etc*. All of these frameworks are designed and optimized for both training and inference on computers with powerful GPUs.

However, we envision that deep learning applications will be eventually widely deployed on mobile devices. It is also expected that for computer vision tasks mobile devices will only perform inference (forward pass), since training can be carried out offline by computers with powerful GPUs. In the rest of this paper, the terms "inference", "test" or "forward pass", mean the same.

Since both the frameworks, as well as the CNN models are designed for computers with powerful GPUs, they may not effectively and efficiently work on mobile devices due to several factors, *e.g.*, constrained memory and energy and limited computing capability. Neural networks for vision tasks are very complex – for example, VGGNet [31] has 528M parameters and requires over 15G FLOPs (FLoating-point OPerations) to classify a single image. Due to the large amount of parameters and FLOPs, and the need to enable running these CNNs on resource-constrained mobile devices, several works focus on accelerating the computation of CNNs on mobile devices by compressing parameters [24], [34], by cloud offload [16], and by distributing computation to heterogeneous processors on-board [26]. Complementary to these techniques, our goal is to model the computation of CNNs in terms of performance (compute time) and resource usage (memory and energy). Our motivation is that our system can provide guidelines to decide when performance optimizations, offloading, *etc*. are required to successfully run analytics tasks on mobile devices. For instance, using the output of our models, one could decide to run all the convolutional layers on the mobile device while offloading the fully connected layers to the cloud so as to cut down on the memory requirement on the mobile device. Although accurately modeling the performance and resource usage of CNNs is very hard, we make progress towards achieving it.

This paper overviews the workflow of CNNs, shares the experiences of deploying CNNs on mobile devices, gives the measurements and analysis of performance and resource usage, and models the inference phase of the CNNs on mobile devices. In doing so we face significant challenges. (*i*) *Profiling overhead*: to measure timing of GPU computations, we need to add a synchronization call that waits for all the results to come back before recording the time. As pointed out by [3], this causes an overhead, as some cores may be

- *Z. Lu is with the Deparment of Computer Science, Peking University. E-mail: zongqing.lu@pku.edu.cn. S. Rallapalli is with IBM Research. Email: srallapalli@us.ibm.com. K. Chan is with Army Research Laboratory. E-mail: kevin.s.chan.civ@mail.mil. S. Pu is with Hikvision. Email: pushiliang@hikvision.com. T. La Porta is with the Department of Computer Science and Engineering, Pennsylvania State University. E-mail: tlp@cse.psu.edu.*

idling while waiting for the rest of the cores to complete the computation. We address this challenge by amortizing this measurement cost by executing the computing task a large number of times and averaging the running time. This ensures that the overhead per iteration is negligible. (*ii*) *Different types of layers*: CNNs are composed of various types of layers, so to model the computation of all the different types is challenging. On the other hand, since the main computation of all these layers boils down to matrix multiplication, we are able to model the different layers by abstracting out the details and focusing on the core of the computation. (*iii*) *How matrix multiplication scales*: as the core of the computation of CNNs, it is important to understand how the computation scales with the sizes of matrices in terms of the compute time and resource usage. Due to the large number of combinations of matrix sizes, this can be very challenging. However, by extracting the matrix multiplication sizes of the popular CNNs, we observe that all of them result into a small set of matrix sizes and thus we are able to accurately model them for different mobile platforms.

Contributions: (*i*) We deploy the popular CNN models including AlexNet [25], VGGNet [31], GoogleNet [32], and ResNet [17] using the Caffe framework [22] on mobile platforms (*i.e.*, NVIDIA TK1 and TX1), where the inference phase is run on both CPUs and GPUs (Section 3). (*ii*) We measure and analyze the performance and resource usage of the inference phase of these CNN models on a layerwise granularity. Our findings point out the potential ways of optimizing the computation of CNNs on mobile devices (Section 4). (*iii*) We profile and model the performance and resource usage of CNNs. We build a modeling tool, *Augur*, that takes a CNN model descriptor as the input and estimates the performance and resource usage of the CNN so as to give insights on how well the CNN can be run on a mobile platform without having to implement and deploy it (Section 5).

2 Background

2.1 Overview of CNNs

Our goal is to model the resource requirements of the forward pass of a CNN. The CNN architecture is typically composed of convolutional, normalization, and subsampling layers – optionally followed by fully connected layers. We overview these layers below, as it lays the foundation for the rest of our work.

Convolutional Layer: The convolutional (CONV) layers form the core of CNNs. The parameters of this layer are a set of kernels (weights) and biases learned during the training phase. During the forward pass, kernels are convolved across the width, height, and depth of the input, computing the dot product between the kernel and the input and producing the output volume. Since the main operation is dot product between the kernels and local regions of the input, the forward pass of a CONV layer can be formulated as a matrix multiplication. For the input volume, each local region (a block of pixels) is stretched into a column of a matrix, and the number of columns is the total number of local regions (im2col). The kernel is stretched into a column of another matrix, and the number of columns is the number of kernels. Finally, the product of the matrix multiplication is reshaped to the output volume with a depth equal to the number of kernels (col2im). For example, the input of AlexNet $[227 \times 227 \times 3]$ (width × height × depth) is convolved with 96 kernels at size $[11 \times 11 \times 3]$ and with a stride 4, and hence there are 55 locations along both width and height. So, the matrix for the input is $[3025 \times 363]$, the matrix of the kernels is $[363 \times 96]$, and the produced matrix is $[3025 \times 96]$ and finally reshaped to $[55 \times 55 \times 96]$.

The CONV layer is commonly implemented using the matrix multiplication function of Basic Linear Algebra Subprograms (BLAS) on CPUs and cuBLAS [2] on CUDA GPUs for acceleration. However, as many values in the input volume are replicated multiple times in the matrix stretched from the input volume, it uses more memory than the input volume itself.

Pooling Layer: The pooling (POOL) layer commonly sits between CONV layers and performs downsampling to reduce the spatial size (width and height) of the features. The pooling is performed on local regions with the kernel size defined by a CNN model. The most common pooling operation in the state-of-the-art CNN models is max pooling. The pooling layer independently operates on the input volume without parameters, and hence its implementation is simple.

Normalization Layer: Two types of normalization layers are commonly used in CNNs: local response normalization (LRN) and batch normalization (BatchNorm). However, LRN's role has been outperformed by other techniques, such as BatchNorm, and thus here we only detail BatchNorm.

BatchNorm is introduced to reduce the internal covariant shift (change in distribution of inputs due to the weight updates in the previous layer) during training [21]. During test phase, BatchNorm normalizes the input volume on each dimension (weight × height), *e.g.*, for the i-th dimension, as follows,

$$\widehat{x}^{(i)} = \frac{x^{(i)} - \mathrm{E}[x^{(i)}]}{\sqrt{\mathrm{Var}[x^{(i)}]}},$$

where $\mathrm{E}[x^{(i)}]$ and $\mathrm{Var}[x^{(i)}]$ are learned during the training phase for dimension i.

Fully Connected Layer: Each neuron in a fully connected (FC) layer is connected to all activations in the previous layer. Due to the full connectivity, there are a huge number of parameters, which places heavy burden on memory usage and computation. Recently, FC layers have fallen out of favor, *e.g.*, the latest CNNs, *i.e.*, GoogleNet and ResNet, only have one fully connected layer as the classifier. This dramatically reduces the number of parameters, *e.g.*, 26MB parameters in GoogleNet while 233MB in AlexNet, and it was found that FC layers of VGGNet can be removed with no performance reduction. Therefore, it is anticipated that CNNs will eliminate the use of FC layers. The forward pass of FC layers is also implemented as a matrix multiplication.

Besides these four layers, rectified linear unit (ReLU) layer that applies an elementwise function, *e.g.*, $\max(0, x)$, on the input volume, is also commonly used in CNNs. However, ReLU is simple, has no parameters, and does not change the size of input volume. Thus we skip the detail of ReLU layer.

TABLE 1
CNN models

Layer	AlexNet	VGGNet	GoogleNet	ResNet	Inception	SqueezeNet	MobileNet
CONV	5	13	57	53	94	26	28
POOL	3	5	14	2	14	4	1
NORM	2		2	53	94		27
ReLU	7	15	57	49	94	26	27
FC	3	3	1	1	1		
Concat			9		11	8	
Scale				53	94		27
Eltwise				16			
Total	20	36	140	227	402	64	110

2.2 Related Work

Although CNNs have been applied to various computer vision applications on different computing platforms, only a few works consider running CNNs on mobile devices, which we envision to be a significant future area for the deployment of deep learning applications.

Among these works, many focus on accelerating the computation of CNNs, *e.g.*, by compressing parameters [15], [24], [34], by cloud offload [16], [23], and by distributing computation to heterogeneous processors (GPU, DSP) onboard [12], [19], [26]. Some consider reducing the memory usage [13], [20], [30] and energy [11] while maintaining high inference accuracy. The resource bottlenecks of running CNNs on mobile devices are preliminarily investigated in [27]. Different CNNs are benchmarked in [9], but it does not consider how to model the compute requirements of CNNs.

While CNNs grow from a few layers to a thousand layers, the computational capability of mobile devices continues to improve. As a result, different mobile devices perform differently on different CNNs, and hence custom optimization and offloading may or may not be needed, depending on whether and how efficiently a CNN can be run on a given mobile platform. This question motivates our work.

3 Measurement Set-Up

To understand the resource requirements of the forward pass of CNNs, we deployed several CNN models on two mobile platforms using the popular deep learning framework – Caffe.

Platforms: Although some frameworks (*e.g.*, Caffe, Torch) can run on Android or iOS, they do not support GPU acceleration on off-the-shelf mobile devices, such as smartphones or tablets. To understand the performance and resource usage of CNNs on both mobile CPUs and GPUs, in this paper, we focus on two developer kits for low power edge devices – NVIDIA TK1 and TX1.

TK1 is equipped with a 2.3GHz quad-core ARM Cortex-15A 32bit CPU, 192 CUDA cores Kepler GPU, and 2GB DDR3L RAM. TX1 is more powerful and has a 1.9GHz quad-core ARM Cortex-A57 64bit CPU, 256 CUDA cores Maxwell GPU, and 4GB LPDDR4 RAM. The system-on-chip (including CPU and GPU) of TK1 and TX1 also appears in many off-the-shelf mobile devices, such as Google Nexus 9 and Pixel C, but none of these devices are enabled to support CUDA, on which deep learning frameworks are built for GPU acceleration. Thus, for ease of experimentation we choose NVIDIA TK1 and TX1, the results of which should indicate the performance of CNNs on mobile devices. Moreover, to mitigate the effect of slow start on CPU and GPU, we disable CPU scaling and force the CPU cores to run at max performance and also set the GPU frequency to the highest. Note that in the early version of this work [29], we have investigated the case with frequency scaling.

Framework: There are several frameworks for deep neural networks. As mentioned before, most of the frameworks use BLAS on CPUs and cuBLAS on GPUs for the CNN computations and thus show similar performance. In this paper, we use the popular Caffe framework, where the choice of BLAS is OpenBLAS [5].

CNN Models: For the measurement, we consider the most popular CNN models including AlexNet, VGGNet (VGG-16), GoogleNet, and ResNet (ResNet-50). We also investigate several recent CNN models including Inception (Inception-v3) [33], SqueezeNet [20], and MobileNet [18]. Although the architectures of these models are quite different, from several layers to more than one hundred layers and from regular stacked layers to branched and stacked layers, they are mainly built on the basic layers of CNNs. Table 1 shows how many these layers each model contains.

4 Initial Measurement Study

In this section, we investigate the computation, compute time, and resource usage of the most popular deep learning models on different mobile platforms and then understand the constraints of running CNNs on mobile platforms.

4.1 Timing

First, we measure the timing of each CNN model on different platforms using CPU and GPU in terms of (*i*) complete forward pass: *i.e.*, timing is measured for the entire forward pass and (*ii*) as summation of individual layer times. We also calculate the number of FLOPs for each model and each type of layer.

AlexNet has the fewest layers among these models and indeed requires the least amount of computation in terms of FLOPs, *i.e.*, 729M, where CONV and FC layers take more than 99%, as shown in Table 2. On the CPU of both TK1 and TX1, the summation of layerwise timing perfectly matches with that of a full forward pass, which are about 600ms on TK1 CPU and 300ms on TX1 CPU. The CONV layers on TX1 run much faster than on TK1 (more than 4x), but, surprisingly, the FC layers are slower. Since the basic computation of both CONV and FC is matrix multiplication, the

TABLE 2
Timing benchmarks on AlexNet

Platform		Layerwise Pass (ms)					Total (ms)	Forward Pass (ms)
		CONV	POOL	LRN	ReLU	FC		
TK1	CPU	318.9±0.3 **51.58%**	6.1±0.0 0.98%	103.7±0.0 16.78%	4.6±0.0 0.75%	186.7±0.1 29.87%	618.2±0.4	618.8±0.3
	GPU	15.1±0.0 34.46%	0.7±0.0 1.50%	1.0±0.0 2.27%	0.8±0.0 1.71%	26.0±0.1 **59.20%**	43.9±0.1	43.2±0.4
TX1	CPU	74.9±1.5 22.24%	8.3±0.5 2.48%	51.6±0.3 15.31%	1.8±0.0 0.54%	200.2±0.1 **59.42%**	336.9±1.5	335.5±2.7
	GPU	6.7±0.5 33.19%	0.4±0.0 1.89%	0.7±0.0 3.53%	0.5±0.0 2.40%	11.7±0.0 **57.71%**	20.3±0.5	19.7±0.6
FLOPs		666M **91.36%**	1M 0.14%	2M 0.27%	0.7M 0.10%	59M 8.09%	729M	

TABLE 3
Timing benchmarks on VGGNet

Platform		Layerwise Pass (ms)				Total (ms)	Forward Pass (ms)
		CONV	POOL	ReLU	FC		
TK1	CPU	7151.7±0.5 **93.00%**	60.1±0.1 0.78%	95.5±0.0 1.24%	382.8±0.2 4.96%	7690.2±0.5	7689.6±0.3
	GPU	245.0±0.3 **77.50%**	4.1±0.0 1.29%	9.9±0.1 3.12%	57.8±0.0 17.95%	316.2±0.3	316.4±0.3
TX1	CPU	1100.2±2.4 **68.34%**	70.2±0.3 4.36%	36.9±0.1 2.29%	402.5±0.2 25.00%	1609.9±2.4	1608.2±1.0
	GPU	109.0±0.7 **72.77%**	2.2±0.1 1.46%	5.9±0.3 3.91%	32.4±0.1 21.61%	149.73±1.3	147.8±0.8
FLOPs		15360M **99.08%**	6M 0.04%	14M 0.09%	124M 0.79%	15503M	

results seem contradictory at first. However, we investigate the problem and give the reasonable explanations below.

First, even though the clock is slower on TX1 compared to TK1 – 1.9 GHz vs. 2.3 GHz, TX1 runs more instructions per clock cycle compared to TK1 (3 vs. 2) and hence the performance of TX1 CPU is expected to be better than TK1 CPU as we see for the CONV layers. Second, FC layers have many more parameters than the CONV layers. Therefore, FC layers are more bottlenecked by the memory whereas CONV layers are more compute bound. Third, the L1 data cache size is 32 KB on both and L2 cache is larger on TK1 compared to TX1. Even if cache size is same on both, as the address is longer on TX1 (64 bit vs. 32 bit), more memory is used for addressing and thus less memory is available to save the data itself on the cache. This means that the CPU needs to fetch data from RAM to the cache more often while executing the FC layers on TX1 due to the large number of parameters which causes the slow down.

GPUs can greatly accelerate the computation of a CNN and thus significantly improves the performance over CPUs as expected, *i.e.*, more than 10x faster on TK1 and TX1. The more advanced TX1 GPU outperforms TK1 GPU on all types of layers as expected and has better performance (about 2x) in both the summation of layerwise passes and a full forward pass. Moreover, the summation of layerwise timing also matches with that of a full forward pass on GPUs.

VGGNet has 2x CONV layers compared to AlexNet (Table 1). However, the number of operations is 20x that of AlexNet, as shown in Table 3, mainly because VGGNet uses much larger feature maps. While other results a follow similar pattern as AlexNet, the throughput of both CPU and GPU on VGGNet is higher than on AlexNet. For example, the throughput of TK1 CPU on AlexNet is 1 GFLOPS (GFLOPs per Second) and of VGGNet is 2 GFLOPS. This is mainly because both CPU and GPU have better throughput on matrix multiplication with larger size.

GoogleNet has more than 50 CONV layers, many more than AlexNet. However, the CONV layers have only two times more FLOPs than that of AlexNet. The main reason is that the size of the kernels and feature maps is small, which dramatically reduces the number of operations. GoogleNet also employs LRN that significantly affects the performance of the CPU on both TK1 and TX1, similar to AlexNet. For example, it takes more than 33% of total time on TX1 CPU.

However, we face one challenge: the summation of layerwise timing does not match the timing of the full forward pass on GPUs any more. The reason for the mismatch in the timing is that CUDA supports asynchronous programming. Before time measurement, an API (*i.e.*, cudaDeviceSynchronize) has to be called to make sure that all cores have finished their tasks. This explicit synchronization is the overhead of each time measurement on GPUs.

The difference between layerwise timing and full forward pass on GoogleNet is much larger than AlexNet and VGGNet (they are negligible) as shown in Table 4. GoogleNet has one hundred more layers than AlexNet and VGGNet and thus much higher measuring overhead on the GPUs. This is a motivation for us to devise measurement techniques that can overcome these measurement overheads as will be discussed later in Section 5.

TABLE 4
Timing benchmarks on GoogleNet

Platform		Layerwise Pass (ms)						Total (ms)	Forward Pass (ms)
		CONV	POOL	LRN	ReLU	Concat	FC		
TK1	CPU	753.3±0.1	68.7±0.0	214.2±0.0	22.8±0.0	2.0±0.0	2.7±0.0	1064.0±0.1	1063.8±0.2
		70.80%	6.46%	20.13%	2.14%	0.19%	0.25%		
	GPU	47.60±0.2	5.6±0.1	1.8±0.0	5.8±0.1	2.2±0.1	0.5±0.0	64.2 ±0.3	57.6±0.3
		74.17%	8.65%	2.81%	9.03%	3.36%	0.79%		
TX1	CPU	159.6±3.3	92.7±0.8	136.8±0.6	8.9±0.0	1.6±0.0	2.7±0.0	402.5±2.1	399.1±1.3
		39.66%	23.03%	33.98%	2.21%	0.39%	0.68%		
	GPU	26.0±2.4	3.0±0.2	1.0±0.0	3.6±1.0	1.3±0.4	0.2±0.0	35.6±4.1	31.9±2.5
		73.03%	8.39%	2.88%	10.06%	3.66%	0.70%		
FLOPs		1585M	13M	3M	3M		1M	1606M	
		98.80%	0.80%	0.20%	0.20%		0.06%		

TABLE 5
Timing benchmarks on ResNet

Platform		Layerwise Pass (ms)							Total (ms)	Forward Pass (ms)
		CONV	POOL	BatchNorm	ReLU	Scale	Eltwise	FC		
TK1	CPU	1826.0±0.2	8.8±0.0	97.0±0.1	64.0±0.0	42.0±0.1	24.8±0.1	5.3±0.0	2068.0±0.2	2067.0±0.1
		88.30%	0.42%	4.69%	3.09%	2.03%	1.20%	0.26%		
	GPU	82.4±0.4	1.5±0.0	59.2±0.3	9.7±0.1	19.6±0.2	12.9±0.1	1.0±0.0	187.8±0.5	148.6±0.4
		43.86%	0.81%	31.52%	5.18%	10.46%	6.88%	0.54%		
TX1	CPU	346.1±0.7	13.7±0.0	77.1±1.2	24.8±0.0	41.8±0.5	19.4±0.6	5.3±0.0	528.4±1.1	525.6±0.7
		65.50%	2.60%	14.60%	4.69%	7.90%	3.67%	1.01%		
	GPU	39.2±3.3	0.6±0.0	31.4±3.3	5.0±0.4	11.5±0.6	7.9±0.2	0.4±0.0	96.6±5.4	77.7±0.7
		40.56%	0.61%	32.54%	5.15%	11.88%	8.22%	0.45%		
FLOPs		3866M	2M	32M	9M	11M	6M	2M	3922M	
		98.59%	0.05%	0.81%	0.23%	0.27%	0.14%	0.05%		

The measuring overhead is much less than that on GPUs with frequency scaling in [29]. This is because the frequency of GPUs is scaled down before performing time measurement (during the computing of a layer) and thus the computation of each layer takes longer time than in a full forward pass where the frequency of GPUs is more steady.

GoogleNet has a layer, named Concat, that does not involve any computation, but concatenates the outputs from previous layers, thus involving memory operations only. On the shared-memory architecture of mobile devices, the memory operations of GPUs may be slower than CPUs due to the slower frequency, *e.g.*, 852MHz compared to 2.3GHz on TK1, and synchronization. This is reflected in the timing measurements on Concat layers. In Table 4, we can see that TK1 CPU slightly outperforms GPU on Concat layers. During a full forward pass, the GPU has to wait for a Concat layer is fully completed (*i.e.*, an implicit synchronization) before proceeding to next layer. Therefore, the measurement should reflect the actual cost of the Concat layer.

ResNet has more than two hundred layers. ResNet includes BatchNorm, Scale, and Eltwise that are not commonly used by other models. These layers are not expensive in term of FLOPs (less than 1% for each type) as shown in Table 5. However, BatchNorm takes a much larger fraction of total time, especially on the GPUs, *i.e.*, about 30% on both GPUs. This is because BatchNorm cannot be accelerated by the GPUs as much as CONV, and it is only about 2x faster than the CPUs, as shown in Table 5. Moreover, as ResNet has more layers than GoogleNet, the measurement overhead increases and hence the difference between the summation of layerwise passes and full forward pass enlarges. As mentioned before, we develop schemes to overcome this overhead in Section 5.

4.2 Memory

The memory requirement to run a CNN comes from three major sources: (*i*) the memory that holds the parameters of the CNN; (*ii*) the memory that stores intermediate data of the CNN; and (*iii*) the workspace for computation. A majority of the CNN parameters come from CONV and FC layers (*i.e.*, weights and biases). Intermediate data is the output of each layer (*i.e.*, the input of next layer), *e.g.*, feature maps. Some types of layers require additional space to perform computation, *e.g.*, on CONV layers, additional memory is needed to hold the matrix stretched from the input data for matrix multiplication. The workspace memory is mostly consumed by the matrix multiplication of CONV layers. The NVIDIA CUDA Deep Neural Network library (cuDNN) [4] can reduce the workspace by sacrificing the computation speed on GPUs. However, as the workspace is not the most significant part, cuDNN cannot reduce the memory usage of CNNs dramatically.

Table 6 shows the memory of weights and biases of CONV and FC layers, intermediate data, and workspace of CONV layers for each CNN. The memory size of each type can be easily determined by parsing the model descriptor (*e.g.*, a prototxt file in Caffe). Table 6 also gives the measured memory usage of Caffe, running each CNN on these platforms. One can see that deeper CNNs (from AlexNet

TABLE 6
Memory of CNN models on platforms (MB)

Type/Platform		AlexNet	VGGNet	GoogleNet	ResNet
Weights & Biases		233	528	26	97
Data		8	110	53	221
Workspace		11	168	46	79
TK1	CPU	324	972	161	409
	GPU	560	1508	196	533
TX1	CPU	362	1013	200	453
	GPU	589	1537	226	562

to ResNet) may not require more memory, especially for GoogleNet, which takes the least memory among them. Memory usage on TX1 is more than TK1, because TK1 is running a 32-bit OS while TX1 is running a 64-bit OS, which incurs more memory usage for the framework itself.

To speed up the running time of CNNs, all memory should be allocated beforehand and not released during the computation. Although existing frameworks (*e.g.*, Caffe[1]) follow this rule, they are designed for training and testing on workstations with powerful GPUs, and thus not quite suitable for mobile devices in terms of memory management.

Unified Memory Architecture: Unlike workstations[2] where GPUs have dedicated memory, mobile platforms usually have a unified memory architecture, where the GPU shares system memory with the CPU. On workstations, in the current implementation of Caffe, data is transferred to and from the memory of GPU for access, which is efficient on workstations. However, on united memory architecture, *e.g.*, TK1 and TX1, memory transfer from CPU to GPU simply generates a redundant data copy on the system memory. As shown in Table 6, on both TK1 and TX1, the memory usage on GPU is always more than CPU, and the additional memory is actually used to hold a redundant copy of the parameters of the CNNs (mostly weights and biases). For example, running AlexNet on TK1 GPU requires 560MB memory, which is 236MB more than on CPU, while the weights and biases of AlexNet are 233MB in total. This also stands for other CNNs.

Mobile GPUs can directly access data by mapping host memory without degrading performance and incurring memory transfer (*i.e.*, *zero-copy* memory). Existing frameworks, including Caffe, Torch, and Theano, do not take into consideration the unified memory architecture for mobile platforms. On the contrary, the unified memory architecture can be exploited to design a tailored computing framework for mobile devices. (*i*) We can eliminate memory transfers between CPU and GPU. (*ii*) We can compute a CNN in the most efficient way; *i.e.*, each layer can be executed on the most efficient unit, switching back and forth between GPU and CPU, without incurring additional memory transfer overhead.

4.3 Power and Energy

The power supply of TK1 and TX1 provides power for all the components of the boards. Therefore, it is difficult to capture the power variation of CPU and GPU by measuring the power supply. Even if it could capture the variation, it is also difficult to synchronize the timing for the power measurement, since the computation of CNN models is instantaneous, especially on GPUs. Therefore, it is preferred to use the power monitor on-board to measure the power of CPU and GPU. However, the TK1 board is not equipped with such power monitors (only for the power of the board), and thus we only give the power and energy measurements for TX1.

The power is measured by reading the output of INA3221 monitors for CPU and GPU on TX1. The output of INA3221 monitors is accessed about every two milliseconds and every one millisecond for the computation on TX1 CPU and GPU, respectively, such that it is able to capture the variation of the power and result in negligible overhead. In addition, as the compute time of some layers is less than the time interval between reading accesses, the energy usage of CNNs is calculated for individual forward passes instead of individual layers.

The power and energy of forward passes of AlexNet, VGGNet, GoogleNet, and ResNet is illustrated in Fig. 1, 2, 3, and 4, respectively.

AlexNet. When AlexNet is run on TX1 CPU, as depicted in Fig. 1a, the CPU power gradually increases and hits a plateau, which is about 5000mW. Although the power fluctuates on the plateau, the fluctuations are small. As the CPU power is not always at the peak when performing the first forward pass, it consumes less energy than the following forward passes (*i.e.*, 1.39J vs. 1.62J). On TX1 GPU, as shown in Fig. 1b, the sum power of CPU and GPU is about 6000mW. As the computation is performed on the GPU, the CPU power is only about 1000mW. The energy expenditure per forward pass is only 0.11J, about one fifteenth of TX1 CPU. This is because the computation finishes much faster on the GPU.

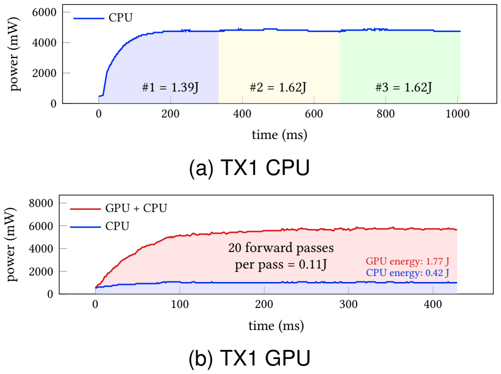
VGGNet. The peak power of VGGNet on both TX1 CPU and GPU is higher than AlexNet, as illustrated in Fig. 2a and 2b. This is mainly because the throughput of CPU and GPU is higher on VGGNet than AlexNet as discussed in Section 4.1.

Different from AlexNet, at the end of a forward pass, the CPU power drops dramatically in Fig. 2a, and the GPU power also slides downward in Fig. 2b. This behavior can be explained as follows. First, VGGNet has much larger FC layers than AlexNet, the compute time of which is also much longer than AlexNet, as shown in Table 2 and 3. In addition, as discussed before, FC layers incur a lot of memory accesses and thus the computation of FC layers is not as intensive as CONV layers. Therefore, given the less intensive workload and longer compute time, the throughput of CPU and GPU goes down when running FC layers at the end of a forward pass of VGGNet, and so does the power.

GoogleNet and ResNet. On TX1 CPU, due the unique architecture, the power of GoogleNet and ResNet also exhibits a unique pattern, and the variation pattern is relatively consistent across forward passes, as depicted in Fig. 3a and 4a.

[1] Caffe allocates the memory for intermediate data on demand (lazily) during the first run, and thus it takes longer time than later runs.

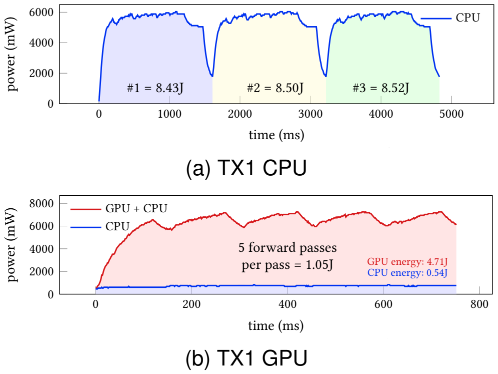
[2] Although GPUs on workstations can also directly access host memory over PCIe, *e.g.*, CUDA kernels, reading data over PCIe is limited by PCIe bandwidth (upto 32GB/s) which is much slower than reading data from GPU memory (limit 200GB/s).



(a) TX1 CPU

(b) TX1 GPU

Fig. 1. Power and energy of AlexNet on TX1.



(a) TX1 CPU

(b) TX1 GPU

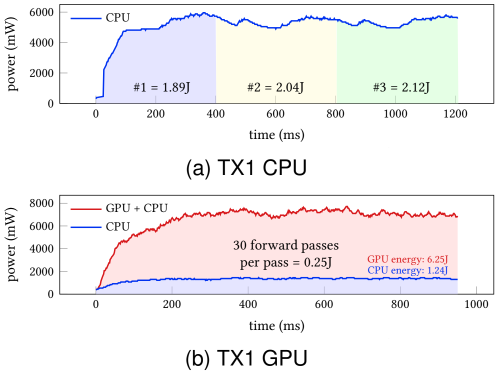
Fig. 2. Power and energy of VGGNet on TX1.

GoogleNet and ResNet have one hundred more layers than AlexNet and VGGNet, which means more workload for the CPU to shift computation to the GPU when using GPUs for acceleration. Therefore, in Fig. 3b and 4b, the CPU power of both GoogleNet and ResNet is higher that of other two models. Moreover, the sum power of CPU and GPU of both GoogleNet and ResNet fluctuates more frequently, since the architecture of GoogleNet and ResNet is largely different from AlexNet and VGGNet (*i.e.*, branched and stacked vs. stacked).

4.4 Analysis

FLOPs. As the throughput of both CPU and GPU is higher on the CNN with more FLOPs (*e.g.*, the throughput of TX1 CPU and GPU on VGGNet is about 10 and 100 GFLOPS, respectively, while 2 and 30 GFLOPS on AlexNet, from Table 2 and 3) and a significant amount of memory operations are involved with the computation of a CNN, FLOPs cannot accurately reflect the compute time of a CNN. Therefore, estimating the compute time of CNNs directly from their FLOPs is not feasible.

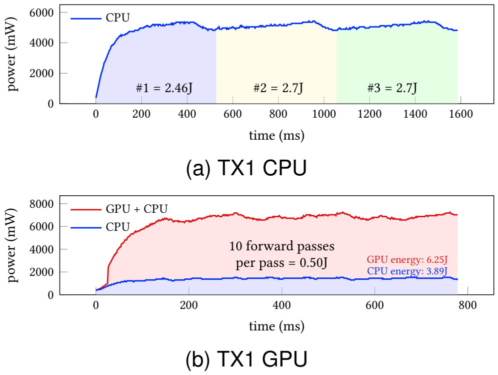
CONV and FC Layer: The computation of CONV and FC layers in most models accounts for a majority of FLOPs (more than 98% in all four models) and running time. Moreover, the power of CPU and GPU is mostly steady during a forward pass, and thus the computing of CONV and FC layers also consumes the most energy. A natural question is whether we can measure these layers instead of the entire network? However, this apparently encounters other difficulties, *i.e.*, layerwise measuring overhead on GPUs, and we have no way to know the exact overhead for each layer, which is hidden by GPUs.



(a) TX1 CPU

(b) TX1 GPU

Fig. 3. Power and energy of GoogleNet on TX1.



(a) TX1 CPU

(b) TX1 GPU

Fig. 4. Power and energy of ResNet on TX1.

Matrix Multiplication: The core of CONV and FC layers are matrix multiplications. Therefore, rather than going into the details of each of the individual layers, if we are able to extract the matrix multiplication component of the layer, we would be able to accurately capture the running time and energy of these layers.

4.5 Additional CNN Models

The measurement of Inception, SqueezeNet, and MobileNet is shown in Table 7. Inception is a more accurate version of GoogleNet with similar branched and stacked layers. However, as shown in Table 7, Inception takes more memory, longer compute time, and more energy, compared with GoogleNet. SqueezeNet has AlexNet-level accuracy with 50x fewer parameters. Indeed, it takes much less memory, but the compute time and energy of SqueezeNet on TX1 GPU is still similar to AlexNet. Different from other CNN models, MobileNet and ShuffleNet [35] respectively employ depthwise convolution and channel shuffle to reduce the computation of CONV layer. However, both require a customized CONV layer. Although depthwise convolution can be performed based on regular CONV layer in Caffe, the computation is not optimized and thus MobileNet (without tuning computation) runs slow as shown in Table 7.

5 Augur

We aim to build a modeling tool that can estimate the compute time, memory, and energy usage of any given CNN descriptor on specific mobile platforms without implementation and deployment. This way, we can take all the factors into consideration during the design phase of a

TABLE 7
Performance of additional CNN models on TX1

		Inception (v3)	SqueezeNet	MobileNet (v1)
FLOPs		5791M	875M	599M
Memory (MB)	Weights & Biases	91	5	16
	Data	196	51	78
	Workspace	180	29	10
Forward Pass (ms)	CPU	852.3±56.3	152.2±3.1	219.9±25.6
	GPU	94.0±0.8	18.1±1.0	137.3±0.6
Energy (mJ)	CPU	3842.2±202.2	643.9±32.9	953.0±35.2
	GPU	739.7±31.9	112.0±11.6	656.7±18.0

CNN. This is critical when designing CNNs for resource-constrained mobile devices.

5.1 Profiling

The basic idea is simple. We first find the matrix multiplications that form the core of the CNN computation. Then we measure their performance based on the BLAS and cuBLAS libraries, which are commonly employed to perform matrix multiplication on CPU and GPU, respectively, in the existing frameworks.

Extract matrix sizes: To find all matrix multiplications and their sizes, we need to parse the descriptor of a CNN. The dimension of input (*e.g.*, images and feature maps) and network parameters (*e.g.*, convolution kernels) determines two matrix sizes (that are to be multiplied) at a CONV or FC layer. As the dimension of feature maps can be changed by some other layers, *e.g.*, POOL layers, we need to trace the dimension of feature maps layer by layer. However, this can be easily done by parsing parameter settings at each layer, such as zero-padding (P), stride (S), the number of output feature maps (N). For instance, in case of a CONV layer, let I denote the spatial dimension of the input feature map, O denote the spatial dimension of the output feature map, K denote the 3D volume of the convolution kernels. Then, we have:

$$O_w = \lfloor (I_w - K_w + 2P)/S \rfloor + 1$$
$$O_h = \lfloor (I_h - K_h + 2P)/S \rfloor + 1.$$

Then, the matrix multiplication at the CONV layer is $[(O_w \cdot O_h) \times (K_w \cdot K_h \cdot K_d)][(K_w \cdot K_h \cdot K_d) \times N]$.

Mitigate measurement overhead: Layerwise timing incurs overhead on GPUs and may cause a large deviation from a full forward pass. Moreover, the overhead is not fixed and varies over each measurement. As illustrated in Table 4 and 5, the measurement overhead (the difference between the summation of layerwise measurements and a full forward pass) of GoogleNet (131 measurements) on TX1 GPU is 3.7 ms (more than 10% of a full forward pass), while the overhead of ResNet (227 measurements) is 18.9 ms (more than 20%). Therefore, we need a way to mitigate the measuring overhead on GPUs for accurate timing and energy of matrix multiplication.

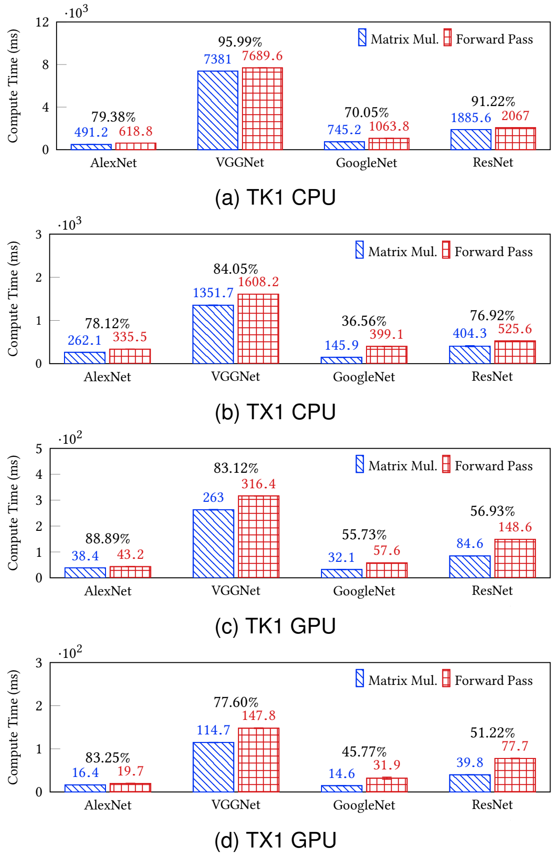
Timing measurements on GPUs can only been recorded after all cores finish their tasks. In a full forward pass, timing is only recorded at the last layer. Therefore, a core may be assigned with the computation of following layers and thus it can continuously perform the computation without synchronization. For example, after finishing the multiply-add operations for the matrix multiplication at a CONV layer, a core can continue to calculate the *max* function of next ReLU layer on the output of multiply-add operations. If layerwise timing is recorded, all cores have to wait until all multiply-add operations of the CONV layer have been completed.

The idea of mitigating the measurement overhead is simple. To benchmark a matrix multiplication, we keep GPUs iteratively running the matrix multiplication task, in a way that GPU cores can continuously perform multiply-add operations without synchronization, before recording the end time. Then, the measurement overhead is amortized over all the iterations, giving accurate timing estimates. When the number of iterations is large enough, the overhead is negligible. Our approach shares a similar basic idea with [10] and [14]. In our experiments we measure the timing of a large number of computing iterations on a matrix multiplication and use the averaged value of each iteration as the running time of the matrix multiplication.

Approximate a forward pass by matrix multiplications: In Fig. 5, we study the fraction of the time spent by matrix multiplications (*matmuls*) in a full forward pass. We do so, by extracting the matmuls, measuring them, and then comparing with the measurement of the full forward pass. Note that due to the averaging methodology explained above, the measurement overhead for matmuls in this section is negligible.

First, as seen in Fig. 5a, matmuls on TK1 CPU take a large portion of the forward pass time – 79.38%, 95.99%, 70.05%, and 91.22% for AlexNet, VGGNet, GoogleNet, and ResNet, respectively. Note that this is very close to the time taken by CONV and FC layers from Table 2, 3, 4, and 5 (81.45%, 97.96%, 71.05%, and 88.56%). Second, the trend is similar on TX1 CPU, as depicted in Fig. 5b, except GoogleNet (only about 36% time spent on matmuls), which is caused by the particular combination of the architecture of TX1 CPU and GoogleNet as discussed in Section 4.1. However, matmuls of GoogleNet still approximate the time taken by CONV and FC layers, which is about 40% from Table 4. Third, the trend on TK1 and TX1 GPUs is similar to the trend on TK1 and TX1 CPUs for AlexNet and VGGNet, as seen in Fig. 5c and 5d. In addition, on TK1 and TX1 GPUs, matmuls take a similar fraction of the forward pass time for each CNN, *e.g.*, 88.89% on TK1 GPU and 83.25% on TX1 GPU for AlexNet.

One thing to note is that while matmuls of GoogleNet



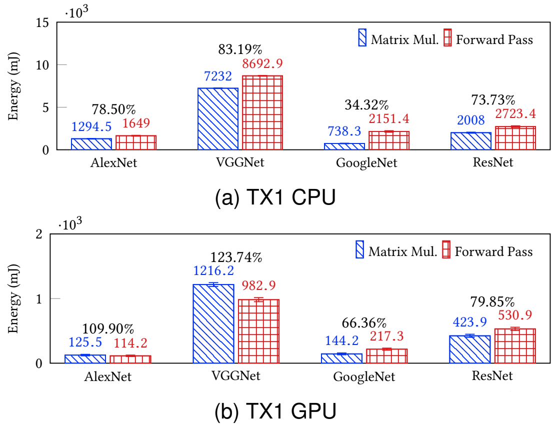
(a) TK1 CPU

(b) TX1 CPU

(c) TK1 GPU

(d) TX1 GPU

Fig. 5. Compute time of matrix multiplication and forward pass of AlexNet, VGGNet, GoogleNet, and ResNet on mobile platforms.



(a) TX1 CPU

(b) TX1 GPU

Fig. 6. Energy of matrix multiplication and forward pass of AlexNet, VGGNet, GoogleNet, and ResNet on TX1.

on GPUs only take about 50% of the time of a forward pass, our previous measurement in Table 4, showed that CONV and FC layers take more than 70%. We believe this is because the matmuls are performed on GPUs without taking into account dependencies, whereas, GoogleNet consists of inception components, each of which has four branches of CONV layers in parallel. Before proceeding to next inception component, all four branches of CONV layers have to be competed. How to handle such dependencies is part of our future work.

Fig. 6 illustrates the comparison between the energy spent by matmuls and by a forward pass. First, as seen in Fig. 6a, on TX1 CPU, the energy consumed by matmuls is about 78%, 83%, 34%, and 73% of the forward pass for AlexNet, VGGNet, GoogleNet, and ResNet, respectively, which matches the fraction of compute time taken by matmuls in Fig. 5b, as expected. However, on TX1 GPU, as illustrated in Fig. 6b, the fraction of energy spent by matmuls is more than the fraction of compute time taken by matmuls for all the CNNs, and surprisingly matmuls even spend more energy than the full forward pass for AlexNet and VGGNet. The results seem conflicting, but we investigate and explain the reason below.

First, the throughput of CPUs is limited, compared to GPUs. Thus, the computing of either a forward pass or matmuls of a CNN can easily reach the maximum throughput of a CPU for the CNN. This means the CPU power when performing a forward pass and matmuls is similar. Therefore, the ratio of energy spent by matmuls over that of a forward pass matches that of compute time.

Second, as matmuls take the most computation of CNNs, the power of TX1 GPU when performing matmuls should be higher than performing other operations (other layers rather than CONV and FC). Therefore, the ratio of energy consumed by matmuls over that of a forward pass is higher than that of compute time.

Third, for AlexNet and VGGNet, the compute time of CONV and FC layers takes a large proportion of forward passes, *i.e.*, 90.9% and 94.38%, respectively, as in Table 2 and 3 (the measurement overhead is negligible for AlexNet and VGGNet). When only the matmuls of the two CNNs are run, the compute time is decreased to 83.25% and 77.60%, respectively. This indicates that the throughput of TX1 GPU when running the matmuls is higher than while running the forward passes, and so is the power. As the compute time of the matmuls of AlexNet and VGGNet is still close to the forward passes, it is possible that matmuls consume more energy than the forward passes. However, it is expected the difference between them is minor, because when the power (throughput) goes up, the compute time goes down.

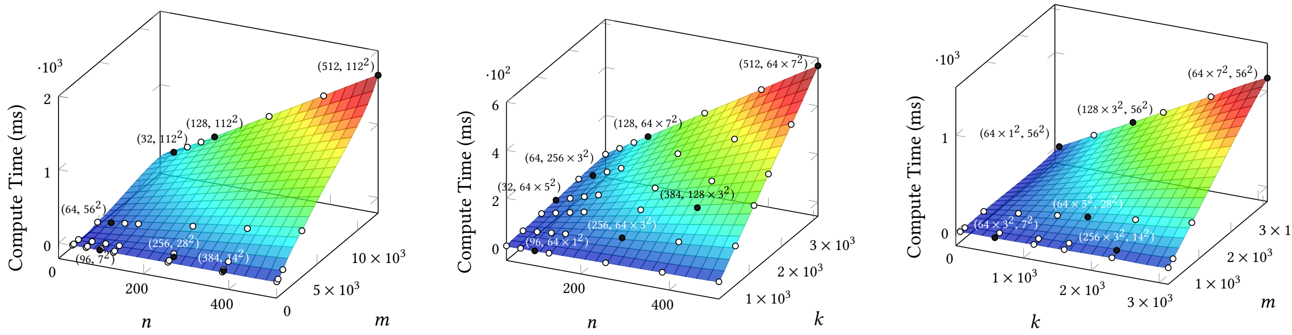
In summary, for most cases, the compute time and energy of matmuls is close to that of a forward pass of a CNN on mobile platforms. Thus, we can predict the compute time and energy of matmuls, to be able to approximately estimate that of a CNN.

5.2 Modeling

So far, we have exactly measured the compute time and energy of matmuls of the CNNs. In this section, we aim to model them, to be able to predict the compute time and energy, just from the matrix sizes. To do so, we benchmark several matrix sizes, as explained below, to understand the relationship between the size of the matrices and the compute time and energy. Based on the relationship, we can estimate the compute time and energy of any matmul.

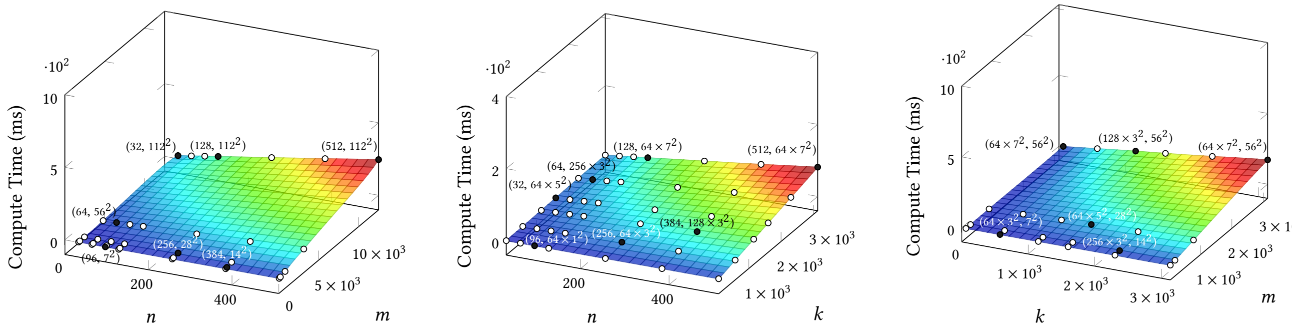
Given the matmul of $[n \times k]$ and $[k \times m]$ (the number of FLOPs is $n \times m \times k$) performed by a CONV layer, n is the number of kernels, k is the size of a kernel in 3D (width $\times$ height $\times$ depth, where depth is the number of input feature maps), and m is the spatial size (width $\times$ height) of output feature maps.

CNNs follow special rules on these parameters of CONV layers. The number of kernels n is usually a multiple of 16, commonly from 32 to 512. The spatial size of a kernel is



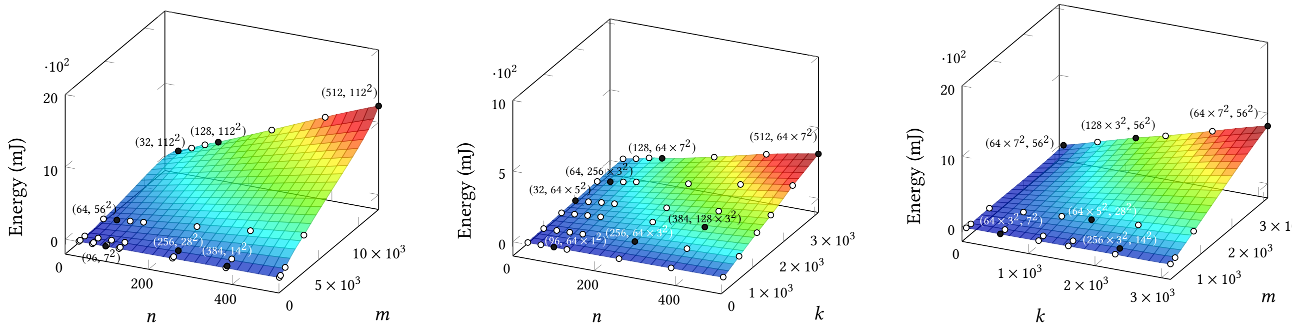
(a) effect of n and m, where $k = 576$ (b) effect of n and k, where $m = 28^2$ (c) effect of m and k, where $n = 256$

Fig. 7. Compute time of matrix multiplication on TK1 CPU with varying n, m, and k.



(a) effect of n and m, where $k = 576$ (b) effect of n and k, where $m = 28^2$ (c) effect of m and k, where $n = 256$

Fig. 8. Compute time of matrix multiplication on TX1 CPU with varying n, m, and k.



(a) effect of n and m, where $k = 576$ (b) effect of n and k, where $m = 28^2$ (c) effect of m and k, where $n = 256$

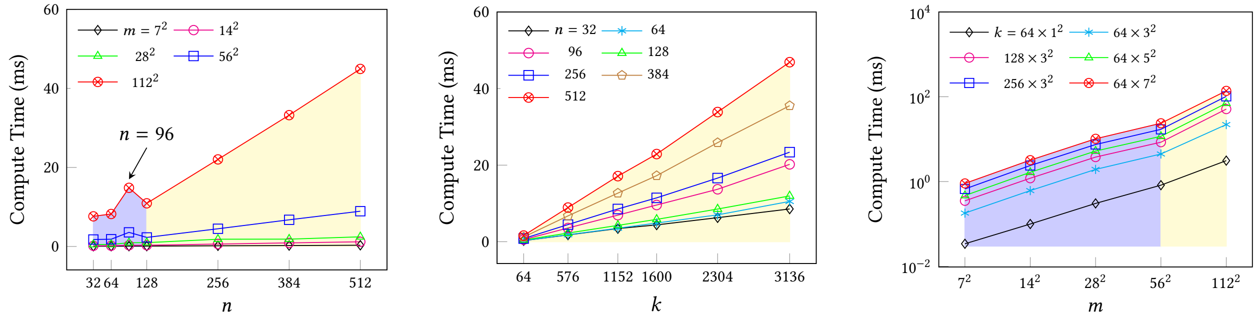
Fig. 9. Energy of matrix multiplication on TX1 CPU with varying n, m, and k.

commonly 1^2, 3^2, 5^2, 7^2, or 11^2. The depth of a kernel is usually the number of kernels in previous CONV layer and hence also a multiple of 16, except the first CONV layer, where the depth is the number of channels of the input image and thus three. The spatial size of output feature maps of a CNN m gradually reduces; it is commonly 112^2, 56^2, 28^2, 14^2, or 7^2, though AlexNet has slightly different ones, *i.e.*, 55^2, 27^2 and 13^2. Based on these parameters in CNNs, we carried out experiments on matmuls with varying n, m, and k, corresponding to the common settings of these parameters. The FC layer is currently used in CNNs only as a classifier (*e.g.*, in GoogleNet and ResNet) and thus its compute time is negligible comparing to the forward pass. Therefore, we do not consider the size of matrices for FC layers in the modeling.

Simple linearity on CPU: Fig. 7 and 8 illustrate the compute time of matmuls on TK1 CPU and TX1 CPU, respectively. Fig. 9 depicts the energy of matmuls on TX1 CPU. The settings of n, m, and k are: $n = [32, 64, 96, 128, 256, 512]$, $m = [7^2, 14^2, 28^2, 56^2, 112^2]$, and $k = [64 \times 1^2, 64 \times 3^2, 128 \times 3^2, 64 \times 5^2, 256 \times 3^2, 64 \times 7^2]$. In each figure, we fix one of three parameters and vary other two; data points are shown as small circles; black circles are labelled with coordinates to highlight the setting of varying parameters.

From Fig. 7a, 7b, and 7c, it is observed that the compute time of matmuls on TK1 CPU scales linearly with n, m, and k. The linearity can also observed on TX1 CPU for the compute time as depicted in Fig. 8a, 8b, and 8c, and for energy as illustrated in Fig. 9a, 9b, and 9c. Thus, we can easily obtain linear models per CPU device, which predict the compute time and energy of matmuls, given the matrix sizes.

Complex linearity on GPU: Fig. 10 and 11 illustrate the compute time of matmuls with varying setting of n, m, and k on TK1 GPU and TX1 GPU, respectively, and Fig. 12 depicts the energy of matmuls on TX1 GPU. The

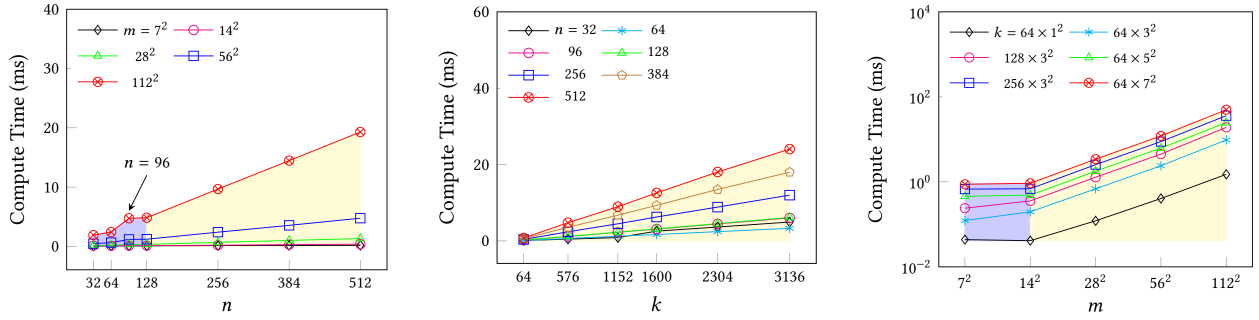


(a) effect of n and m, where $k = 576$

(b) effect of k and n, where $m = 56^2$

(c) effect of m and k, where $n = 256$

Fig. 10. Compute time of matrix multiplication on TK1 GPU with varying n, m, and k.

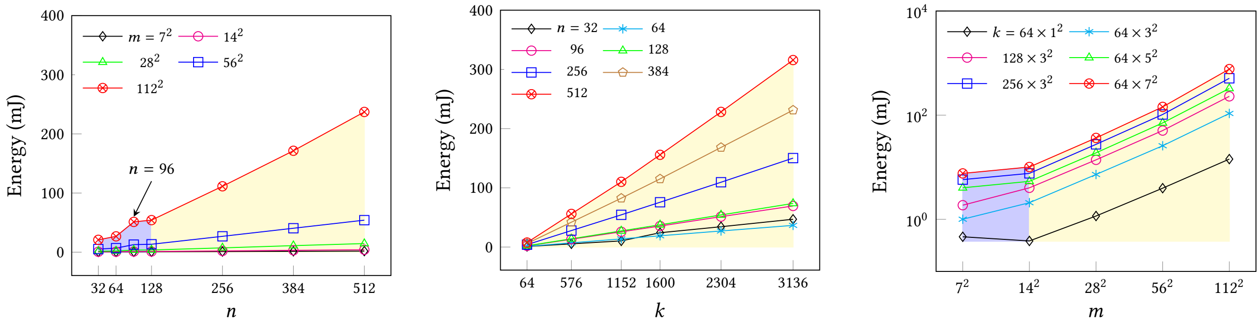


(a) effect of n and m, where $k = 576$

(b) effect of k and n, where $m = 56^2$

(c) effect of m and k, where $n = 256$

Fig. 11. Compute time of matrix multiplication on TX1 GPU with varying n, m, and k.



(a) effect of n and m, where $k = 576$

(b) effect of k and n, where $m = 56^2$

(c) effect of m and k, where $n = 256$

Fig. 12. Energy of matrix multiplication on TX1 GPU with varying n, m, and k.

compute time and energy of matmuls on GPUs exhibits more complex relations with n, m, and k.

Fig. 10a mainly depicts the effect of n, which is bipartite. For all settings of m, the compute time has a monotonic relationship with n from $n = 32$ to 128, except $n = 96$ that incurs even longer compute time than $n = 128$, while, from $n = 128$ to 512, the compute time exhibits a perfect linear relationship with n. Similar trend is also found on TX1 GPU for both the compute time and energy as shown in Fig. 11a and 12a, where the compute time and energy of $n = 96$ is equivalent to that of $n = 128$. Although TX1 GPU has more CUDA cores (256 compare to 192 cores in TK1 GPU) and generally computes faster than TK1 GPU on matmuls, it also has this special value of n. This characteristic of GPUs should be related to the scheme that determines how the CUDA cores compute a matmul in parallel, not just the number of CUDA cores in a GPU. Since cuBLAS is not an open-source library, it is hard to verify these explanations or trace the exact reason behind that. However, it is indicated [2] that matmuls work best if n and m are multiples of 128 on Maxwell architecture (TX1 GPU) and if n is multiple of 256 and m multiple of 192 on Kepler architecture (TK1 GPU). This may explain why it behaves differently when n is small.

For given values of n and m, the compute time linearly increases with k on both TK1 GPU and TX1 GPU as depicted in Fig. 10b and 11b, same with energy on TX1 GPU as illustrated in Fig. 12b. On the other hand, they reveal again that $n = 96$ incurs longer compute time than $n = 128$ for all the values of k on TK1 GPU and has equivalent compute time and energy with $n = 128$ on TX1 GPU.

While the compute time increases with m on TK1 GPU as depicted in Fig. 10c, the effect of m is also bipartite. The compute time has two separate linear relationships with k (different coefficients), *e.g.*, from 7^2 to 56^2 and from 56^2 to 112^2, as highlighted by different regions in Fig. 10c. In each such region, the compute time on different values of k linearly scales with m at mostly the same coefficient.

The effect of m on the compute time and energy on TX1 GPU is also bipartite as illustrated in Fig. 11c and 12c, separated by $m = 14^2$. The compute time and energy in region between $m = 7^2$ and 14^2 may have different linear relations with m for different k. However, beyond $m = 14^2$, both the compute time and energy linearly scale with m at mostly the same coefficient for a given k.

The dichotomy of the performance on both TK1 GPU and TX1 GPU should be related to cuBLAS. cuBLAS may adopt different schemes based on the matrix sizes and the number of CUDA cores to assign the workload to the cores. The transition from one to another can be different on TK1 GPU and TX1 GPU, mainly because they have different number of CUDA cores.

Based on the characteristics discussed above, we are able to model the compute time and energy of matmuls on a specific GPU, though we need more data points than that on a CPU.

5.3 Accuracy

Based on the measurement, profiling, and modeling of CNNs on mobile devices, we built the modeling tool, Augur, that estimates the compute time, memory, and energy usage for any given CNN. Augur first parses the descriptor of a CNN. Based the type and setting of each layer, it calculates the minimal memory needed to perform the computation of the CNN. The memory includes data, parameters, and workspace. Then, Augur extracts the matmul from the computation of the CNN. Based on the profiling of matrix multiplication on TK1 and TX1, *i.e.*, the linear fits obtained from Fig. 7 and 10 for TK1, and Fig. 8, Fig. 9, 11, and 12 for TX1, Augur calculates the compute time (also energy for TX1) for individual matmuls and then uses their summation as the estimate of the compute time (also energy for TX1) of the CNN.

To verify the accuracy of Augur, we model two CNNs that Augur has not profiled before (NIN [28] and VGG19M[3]) and compare the estimates to the measured memory usage, compute time and energy using Caffe.

Fig. 13 depicts the memory usage of NIN and VGG19M on different processing units. The estimate of memory usage is always less than the actual usage, because the estimate does not take into account the memory usage of Caffe itself, which is framework-dependent. However, it is easy to incorporate that if a specific framework is targeted to perform the computation of CNNs. Note that the estimate of Augur is accurate on the memory usage of data, parameters, and workspace as discussed in Section 4.2.

Fig. 14 and 15 evaluate the accuracy of Augur's compute time estimation of NIN and VGG19M, respectively. From Fig. 14 and 15, we observe that the estimate based on only matmuls can approximate the compute time of NIN and VGG19M on both CPUs and GPUs, with more than 70% accuracy for all the cases. Since matmuls generally take a larger proportion of the compute time on CPUs than on GPUs, the estimate on CPUs (up to 95%) is closer to the actual compute time than on GPUs (up to 78%). Moreover, a more powerful processing unit can perform matmuls faster, but the speed-up is not the same across all operations. Therefore, the matmul of a CNN takes a smaller proportion of the compute time on a more powerful processing unit. This explains why the estimate on TK1 CPU (TK1 GPU) is more accurate than TX1 CPU (TK1 GPU) for the same CNN.

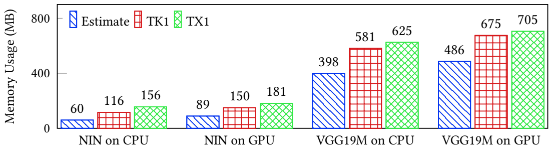


Fig. 13. Memory estimate of NIN and VGG19M.

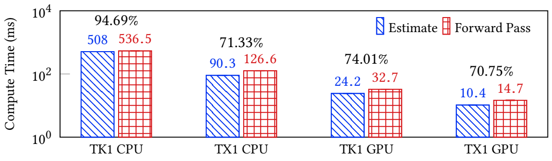


Fig. 14. Timing estimate of NIN.

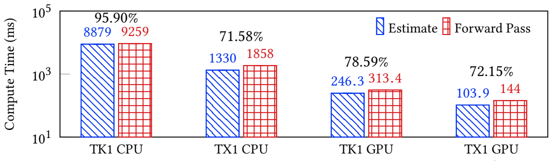


Fig. 15. Timing estimate of VGG19M.

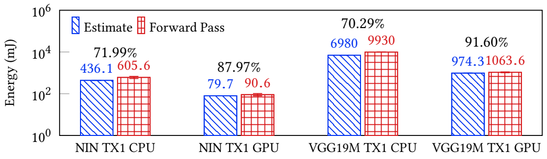


Fig. 16. Energy estimate of VGG19M and NIN on TX1.

Fig. 16 demonstrates the accuracy of Augur on the energy consumption of NIN and VGG19M on TX1. On TX1 CPU, the extent that the estimated energy is close to the measured energy matches that of the compute time; the estimate of compute time and energy is all about 70% for both NIN and VGG19M on TX1 CPU. However, on TX1 GPU, the estimate of energy is closer to the actual energy usage than that of compute time. That is, 87.97% and 91.60%, respectively, for NIN and VGG19M on energy, and 70.75% and 72.15% on compute time. As discussed in Section 5.1, the throughput and power of GPU is usually higher on the matmuls of a CNN than during a forward pass. Therefore, the estimate of energy tends to be more accurate than that of compute time.

Additionally, we investigate the accuracy of Augur on SqueezeNet and MobileNet on TX1. Fig. 17 and Fig. 18, respectively, depict Augur's estimate on compute time and energy of SqueezeNet and MobileNet. Augur's estimation of SqueezeNet is less accurate than NIN and VGG19M on TX1 CPU and GPU. The main reason is that POOL and ReLU layers (non-matmul operations) of SqueezeNet take about 1/3 of compute time on TX1 as we found layerwise forward pass, which makes the matmul-based estimation less accurate. The estimate of MobileNet on TX1 CPU is similar with SqueezeNet, because MobileNet contains many Scale layers which run slow on TX1 CPU. However, these

[3] VGG19M is a modified version of VGGNet with more CONV layers. The FC layers in the original VGGNet are replaced by a CONV layer and a POOL layer to reduce memory usage.

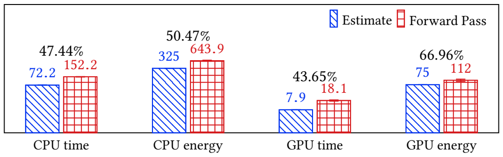


Fig. 17. Estimate of SqueezeNet on TX1.

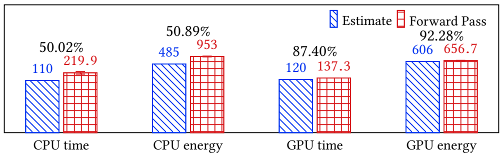


Fig. 18. Estimate of MobileNet on TX1.

layers run very fast on TX1 GPU and thus the estimation on TX1 GPU is much more accurate, about 90%.

In summary, Augur can estimate how efficiently a CNN can be run on mobile devices before any deployment. It can also help the design of CNNs for resource-constrained mobile devices. When designing a CNN model using Augur, designers can estimate the memory usage, compute time, and energy without implementation and deployment and tune the model to satisfy their specific needs.

6 Discussion

Augur can be extended to support additional mobile platforms by simply profiling matrix multiplication operations on them. Matrix multiplications of a CNN take most computation (more than 90% of FLOPs from Table 2, 3, 4, and 5), which commonly takes a dominant proportion of the compute time and energy. Thus, matrix multiplication is currently exploited by Augur to estimate the compute time and energy of a CNN. To obtain a more precise estimate, additional factors need to be taken into consideration, *e.g.*, non-matmul operations (scale, pooling, etc), memory operations (memcpy, im2col, col2im, etc.), convolution types (depthwise or channel shuffle), and CNN architectures (stacked or branched). Augur will be enhanced with these features and this will be our future work.

Moreover, we observe that a framework customized for running CNNs on mobile platforms is highly desired. The framework should be optimized for performing the test phase of CNNs and tailored for the characteristics of mobile platforms, *e.g.*, the unified memory architecture.

On CPUs and GPUs with/without frequency scaling, matrix multiplication can be used to estimate the performance and resource usage of the computation of CNNs on mobile devices (refer to the early version of this work [29] for the case with frequency scaling), and matrix multiplication can be modeled based on the matrix sizes. However, on TK1 and TX1 GPUs, the compute time can be greatly accelerated when the frequency is fixed at the peak. For example, the compute time of a forward pass of GoogleNet on TX1 GPU is 32ms, while it is 143ms on TX1 GPU with frequency scaling as in [29].

Although fixing the frequency of mobile GPUs at the peak consumes more energy, it may be an option to improve the performance (compute time) of CNNs when the resource usage (energy) is not a major concern. For example, the performance of GoogleNet can be improved for real-time processing on 30fps live videos. On the other hand, the frequency can also be adjusted to reduce the energy cost by sacrificing the performance when energy is a major concern. How to tradeoff between performance and resource usage of CNNs on mobile devices will also be our future work.

7 Conclusion

In this paper, we aim to model the resource requirements of CNNs on mobile devices in terms of memory usage, compute time, and energy. By deploying several popular CNNs on mobile CPUs and GPUs, we measured and analyzed the performance and resource usage at a layerwise granularity. Our findings pointed out the potential ways of optimizing the performance of CNNs on mobile devices. As the computation of a CNN is mainly governed by matrix multiplications, we profiled and modeled matrix multiplications on mobile platforms. Based on the measurement, profiling, and modeling, we built Augur that can estimates the compute time, memory, and energy of the CNN so as to give insights on whether and how efficiently the CNN can be run on a mobile platform without implementation and deployment. Therefore, it can help the design of CNNs for resource-constrained mobile devices. When designing a CNN using Augur, designers can acknowledge the performance and resource usage and tune the model to satisfy their specific needs.

Acknowledgments

This work was supported in part by the Army Research Laboratory under Cooperative Agreement Number W911NF-09-2-0053, NSFC under grant 61872009, and Hikvison. An early version of this work appeared in the Proceedings of ACM Multimedia 2017 [29].

References

[1] *Caffe*. http://caffe.berkeleyvision.org/.
[2] *cuBLAS*. https://developer.nvidia.com/cublas.
[3] *CUDA C Programming Guide*. https://docs.nvidia.com/cuda/.
[4] *cuDNN*. https://developer.nvidia.com/cudnn/.
[5] *OpenBLAS*. http://www.openblas.net/.
[6] *TensorFlow*. http://www.tensorflow.org/.
[7] *Theano*. http://deeplearning.net/software/theano/.
[8] *Torch*. http://torch.ch/.
[9] A. Canziani, A. Paszke, and E. Culurciello. An analysis of deep neural network models for practical applications. *arXiv preprint arXiv:1605.07678*, 2016.
[10] X. Cui, Y. Chen, and H. Mei. Improving performance of matrix multiplication and fft on gpu. In *International Conference on Parallel and Distributed Systems (ICPDS)*, pages 42–48, 2009.
[11] K. Gauen, R. Rangan, A. Mohan, Y.-H. Lu, W. Liu, and A. C. Berg. Low-power image recognition challenge. In *Design Automation Conference (ASP-DAC), 22nd Asia and South Pacific*, pages 99–104, 2017.
[12] P. Georgiev, N. D. Lane, K. K. Rachuri, and C. Mascolo. Leo: Scheduling sensor inference algorithms across heterogeneous mobile processors and network resources. In *Proceedings of International Conference on Mobile Computing and Networking (MobiCom'16)*, pages 320–333, 2016.
[13] Y. Gong, L. Liu, M. Yang, and L. Bourdev. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014.
[14] P. Guo and L. Wang. Accurate cuda performance modeling for sparse matrix-vector multiplication. In *2012 International Conference on High Performance Computing & Simulation (HPCS)*, pages 496–502, 2012.

[15] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In *International Conference on Learning Representations (ICLR'16)*, 2016.

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

[17] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR'16)*, pages 770–778, 2016.

[18] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[19] L. N. Huynh, Y. Lee, and R. K. Balan. Deepmon: Mobile gpu-based deep learning framework for continuous vision applications. In *MobiSys*, pages 82–95, 2017.

[20] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5MB model size. *arXiv preprint arXiv:1602.07360*, 2016.

[21] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning (ICML'15)*, 2015.

[22] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of ACM International Conference on Multimedia (MM'14)*, pages 675–678, 2014.

[23] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. In *ASPLOS*, 2017.

[24] Y.-D. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin. Compression of deep convolutional neural networks for fast and low power mobile applications. In *International Conference on Learning Representations (ICLR'16)*, 2016.

[25] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems (NIPS'12)*, pages 1097–1105, 2012.

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

[27] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, and F. Kawsar. An early resource characterization of deep learning on wearables, smartphones and internet-of-things devices. In *Proceedings of the 2015 International Workshop on Internet of Things towards Applications*, pages 7–12, 2015.

[28] M. Lin, Q. Chen, and S. Yan. Network in network. In *International Conference on Learning Representations (ICLR'14)*, 2014.

[29] Z. Lu, S. Rallapalli, K. Chan, and T. La Porta. Modeling the resource requirements of convolutional neural networks on mobile devices. In *Proceedings of ACM International Conference on Multimedia (MM'17)*, pages 1–9, 2017.

[30] C. Reale, H. Lee, H. Kwon, and R. Chellappa. Deep network shrinkage applied to cross-spectrum face recognition. In *Proceedings of IEEE International Conference on Automatic Face & Gesture Recognition (FG'17)*, pages 897–903, 2017.

[31] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations (ICLR'15)*, 2015.

[32] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR'15)*, pages 1–9, 2015.

[33] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of IEEE conference on Computer Vision and Pattern Recognition (CVPR'16)*, pages 2818–2826, 2016.

[34] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng. Quantized convolutional neural networks for mobile devices. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR'16)*, pages 4820–4828, 2016.

[35] X. Zhang, X. Zhou, M. Lin, and J. Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of IEEE conference on Computer Vision and Pattern Recognition (CVPR'18)*, 2018.

Zongqing Lu is an Assistant Professor in the Department of Computer Science, Peking University. He received the B.S. and M.S. degrees from Southeast University, China, and the Ph.D. degree from Nanyang Technological University, Singapore, 2014. Prior to joining Peking University in 2017, he worked as a postdoc in the Department of Computer Science and Engineering, Pennsylvania State University. His current research interests include mobile/edge computing and learning to cooperate.

Swati Rallapalli is a research staff member at the IBM T.J. Watson Research Center. Her work is focused on distributed video analytics (using CNNs/RNNs) on mobile systems and user analytics. In particular, her research focuses on inferring user activities from multiple mobile sensors under stringent resource constraints. She also serves as a principal investigator on the Quality Aware Semantic Video Analytics under US Army Research Lab funded Network Science Collaborative Technology Alliance. She received her PhD from University of Texas, Austin in 2014.

Kevin Chan is research scientist with the Computational and Information Sciences Directorate at the U.S. Army Research Laboratory. Previously, he was an ORAU postdoctoral research fellow at ARL. His research interests are in network science and dynamic distributed computing, with past work in dynamic networks, trust and distributed decision making and quality of information. He has been an active researcher in ARL's collaborative programs, the Network Science Collaborative Technology Alliance and Network and Information Sciences International Technology Alliance. Prior to ARL, he received a PhD in Electrical and Computer Engineering (ECE) and MSECE from Georgia Institute of Technology. He also received a BS in ECE/EPP from Carnegie Mellon University.

Shiliang Pu received the Ph.D. degree in applied optics from the University of Rouen, Mont-Saint-Aignan, France, in 2005. He is currently the Executive Vice Director of the Research Institute with Hikvision, Hangzhou, China. He is also responsible for the company's technology research and development work on video intelligent analysis, image processing, coding, and decoding. His current research interests include image processing and pattern recognition.

Thomas La Porta is the Director of the School of Electrical Engineering and Computer Science at Penn State University. He is an Evan Pugh Professor and the William E. Leonhard Chair Professor in the Computer Science and Engineering Department. He received his B.S.E.E. and M.S.E.E. degrees from The Cooper Union, New York, NY, and his Ph.D. degree in Electrical Engineering from Columbia University, New York, NY. He joined Penn State in 2002. He was the founding Director of the Institute of Networking and Security Research at Penn State. Prior to joining Penn State, Dr. La Porta was with Bell Laboratories where was the Director of the Mobile Networking Research Department. He is an IEEE Fellow and Bell Labs Fellow. He also won two Thomas Alva Edison Patent Awards. Dr. La Porta was the founding Editor-in-Chief of the IEEE Transactions on Mobile Computing. He has published numerous papers and holds 39 patents.