Modeling the Resource Requirements of Convolutional Neural Networks on Mobile Devices

Zongqing Lu
Peking University
zongqing.lu@pku.edu.cn

Swati Rallapalli
IBM Research
srallapalli@us.ibm.com

Kevin Chan
Army Research Laboratory
kevin.s.chan.civ@mail.mil

Thomas La Porta
Pennsylvania State University
tlp@cse.psu.edu

ABSTRACT

Convolutional Neural Networks (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 and widely deployed on mobile devices, *e.g.*, smartphones, self-driving cars, and drones. Therefore, in this paper, we aim to understand the resource requirements (time, memory) of CNNs on mobile devices. First, by deploying several popular CNNs on mobile CPUs and GPUs, we measure and analyze the performance and resource usage for every layer of the CNNs. Our findings point out the potential ways of optimizing the performance on mobile devices. Second, we model the resource requirements of the different CNN computations. Finally, based on the measurement, profiling, and modeling, we build and evaluate our modeling tool, *Augur*, which takes a CNN configuration (descriptor) as the input and estimates the compute time and resource usage of the CNN, to give insights about whether and how efficiently a CNN can be run on a given mobile platform. In doing so Augur tackles several challenges: (*i*) how to overcome profiling and measurement overhead; (*ii*) how to capture the variance in different mobile platforms with different processors, memory, and cache sizes; and (*iii*) how to account for the variance in the number, type and size of layers of the different CNN configurations.

KEYWORDS

Convolutional neural networks; modeling; mobile devices

1 INTRODUCTION

Deep learning has become the norm of state-of-the-art learning systems, especially in computer version. Convolutional Neural Networks (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 transform on the input data and outputs the data to the next layer. Different CNNs for computer vision tasks have been designed, from a few layers to a thousand layers. But, the core of these networks naturally 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 and 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 limited computing capability. CNNs for vision tasks are very complex – for example, VGGNet [21] 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 computing of CNNs on mobile devices by compressing parameters [16, 23], by cloud offload [11], and by distributing computation to heterogeneous processors on-board [18]. However, complementary to these techniques, our goal is to model the resource requirements of CNNs accurately. *Motivation for this is that our system can serve 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 resource requirements 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 performance measurements and analysis, and models the resource requirements

MM'17, October 23–27, 2017, Mountain View, CA, USA.
 ISBN 978-1-4503-4906-2/17/10…$15.00
DOI: http://dx.doi.org/10.1145/3123266.3123389

of 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 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 resource requirements of all the different types is challenging. On the other hand, since 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 resource requirements. Due to the large number of combinations of matrix sizes, this can be very challenging. However, by extracting the matrix multiplication sizes of 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 [17], VGGNet [21], GoogleNet [22], and ResNet [12] using the Caffe framework [15] on mobile platforms (*i.e.*, NVIDIA TK1 and TX1), where the inference phase is run on both CPUs and GPUs (§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 computing of CNNs on mobile devices (§4). (*iii*) We profile and model the resource requirements of CNNs. We also build a modeling tool, *Augur*, which takes a CNN model descriptor as the input and estimates the resource requirements 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 (§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 foundations for modeling the resource requirements.

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. 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. For example, the input of AlexNet $[227 \times 227 \times 3]$ (width $\times$ height $\times$ 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). 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 during training [14]. During test phase, BatchNorm normalizes the input volume on each dimension (weight $\times$ 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. Moreover, 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.

2.2 Related Work

Although CNNs have been applied to various computer vision applications on different computing platforms, only a few works

Table 1: CNN models

Layer	AlexNet	VGGNet	GoogLeNet	ResNet
CONV	5	13	57	53
POOL	3	5	14	2
NORM	2		2	53
ReLU	7	15	57	49
FC	3	3	1	1
Concat			9	
Scale				53
Eltwise				16
Total	20	36	140	227

Table 2: Timing benchmarks on AlexNet

Platform		Layerwise Pass (ms) CONV	POOL	LRN	ReLU	FC	Total (ms)	Forward Pass (ms)
TK1	CPU	318.7±0.2	6.1±0.1	103.8±0.0	4.6±0.0	186.3±0.1	619.8±0.2	619.5±0.2
		51.42%	0.99%	16.74%	0.75%	30.05%		
	GPU	24.6±3.5	2.3±0.6	2.4±0.5	5.2±1.2	35.1±5.9	73.3±10.7	54.7±2.4
		33.53%	3.15%	3.22%	7.11%	**47.95%**		
TX1	CPU	66.9±5.3	7.6±0.0	172.4±0.3	2.4±0.0	644.7±5.3	894.3±4.8	892.7±2.3
		7.48%	0.85%	19.28%	0.27%	**72.09%**		
	GPU	24.2±8.3	1.3±2.6	2.7±3.0	5.9±5.9	15.2±4.7	52.8±15.7	29.3±6.5
		45.79%	2.51%	5.12%	11.23%	28.76%		
FLOPs		666M	1M	2M	0.7M	59M	729M	
		91.36%	0.14%	0.27%	0.10%	8.09%		

Table 3: Timing benchmarks on VGGNet

Platform		Layerwise Pass (ms) CONV	POOL	ReLU	FC	Total (ms)	Forward Pass (ms)
TK1	CPU	7160.5±0.7	60.1±0.1	95.6±0.1	381.6±0.2	7697.9±0.6	7697.8±0.5
		93.02%	0.78%	1.24%	4.96%		
	GPU	263.1±19.3	7.2±0.5	17.5±1.2	57.6±0.5	347.6±20.1	326.7±2.1
		75.68%	2.06%	5.03%	16.58%		
TX1	CPU	1952.9±12.2	71.3±1.5	52.5±1.9	747.7±24.9	2824.6±23.2	2809.1±10.6
		69.14%	2.52%	1.86%	26.47%		
	GPU	136.3±5.4	3.4±1.6	9.9±4.9	32.8±1.3	184.2±7.4	175.3±2.0
		73.98%	1.84%	5.35%	17.82%		
FLOPs		15360M	6M	14M	124M	15503M	
		99.08%	0.04%	0.09%	0.79%		

Table 4: 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

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 computing of CNNs, *e.g.*, by compressing parameters [16, 23], by cloud offload [11], and by distributing computation to heterogeneous processors on-board [18]. Some consider reducing the memory usage to better fit mobile devices while maintaining high inference accuracy, *e.g.*, [10, 13]. The resource bottlenecks of running CNNs on mobile devices are preliminarily investigated in [19]. Different CNNs are benchmarked in [9], but it does not consider how to model the resource 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. It depends 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 requirement 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 and iOS, they do not support GPU acceleration on off-the-shelf mobile devices, such as smartphones or tablets. To understand the performance 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. However, 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.

Framework: There are several frameworks for deep neural networks. As mentioned before, most of the frameworks use BLAS on CPU 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). 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 resource requirements and bottlenecks of running several well known CNN models on mobile platforms.

4.1 Timing

First, we measure the timing of each 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.

Table 5: Timing benchmarks on GoogleNet

Platform		Layerwise Pass (ms)						Total (ms)	Forward Pass (ms)
		CONV	POOL	LRN	ReLU	Concat	FC		
TK1	CPU	755.3±0.2	68.8±0.1	214.3±0.2	22.8±0.0	2.0±0.0	2.7±0.0	1066.2±0.3	1065.6±0.2
		70.84%	6.45%	20.10%	2.14%	0.19%	0.26%		
	GPU	186.9±45.0	20.6±4.9	6.5±1.5	35.3±9.9	13.0±4.4	2.4±0.8	269.3 ±65.6	167.0±44.3
		69.40%	7.65%	2.40%	13.10%	4.81%	0.90%		
TX1	CPU	174.4±3.6	89.9±0.2	349.4±0.6	9.5±0.1	1.7±0.1	5.7±0.0	630.9±3.5	637.9±14.7
		27.64%	14.24%	**55.38%**	1.50%	0.36%	0.90%		
	GPU	165.9±48.8	18.5±11.2	3.3±2.3	49.5±31.2	15.4±9.8	1.2±1.1	258.1±89.8	143.9±59.2
		64.28%	7.16%	1.28%	19.16%	5.96%	0.46%		
FLOPs		1585M	13M	3M	3M		1M	1606M	
		98.80%	0.80%	0.20%	0.20%		0.06%		

Table 6: Timing benchmarks on ResNet

Platform		Layerwise Pass (ms)							Total (ms)	Forward Pass (ms)
		CONV	POOL	BatchNorm	ReLU	Scale	Eltwise	FC		
TK1	CPU	1830.4±0.4	8.8±0.0	97.1±0.1	64.0±0.1	42.0±0.1	24.8±0.1	5.4±0.0	2072.7±0.4	2072.2±0.3
		88.31%	0.42%	4.68%	3.09%	2.03%	1.20%	0.26%		
	GPU	245.8±16.3	5.5±0.6	249.5±11.6	38.7±2.0	76.0±3.3	47.0±2.7	3.9±0.1	673.0±33.4	149.4±4.9
		36.53%	0.81%	**37.08%**	5.75%	11.29%	6.98%	0.58%		
TX1	CPU	362.3±5.4	13.7±0.2	83.5±0.3	33.2±0.1	31.9±3.6	20.4±4.2	22.2±0.1	567.6±7.6	566.8±9.7
		63.83%	2.41%	14.7%	5.86%	5.62%	3.59%	3.92%		
	GPU	279.4±42.6	3.0±2.7	198.1±36.8	63.6±31.3	79.8±24.2	34.8±12.9	1.8±2.4	664.7±116.5	104.4±14.0
		42.03%	0.45%	29.80%	9.57%	12.01%	5.24%	0.27%		
FLOPs		3866M	2M	32M	9M	11M	6M	2M	3922M	
		98.59%	0.05%	0.81%	0.23%	0.27%	0.14%	0.05%		

AlexNet has the least number of layers among these models and indeed requires the least amount of computation in terms of FLOPs, *i.e.*, 729M. 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) and 900ms (on TX1). *Surprisingly, although TX1 has a more powerful CPU, the forward pass on TX1 is slower than TK1.* The CONV layers on TX1 run much faster than on TK1 (more than 4x), but the FC layers are much slower (more than 3x). Since the basic computation of both CONV and FC is matrix multiplication, the results seem contradictory at first. However, we investigate and explain the reasons for the behavior below.

First, even though the clock is slower on TX1 compared to TK1, *i.e.*, 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 bottlenecked by the memory whereas CONV layers are 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 – because the address is longer on TX1 (64 bit vs. 32 bit), more memory is used up for the addressing and we have lesser memory available to save the data itself on the cache. This means that we need 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 significantly accelerate the computation of a CNN and thus improve the performance over CPUs. More advanced TX1 GPU outperforms TK1 GPU as expected. However, we face one challenge: the summation of layerwise timing does not match the timing of the full forward pass on GPUs. The reason for the mismatch is that CUDA supports asynchronous programming. Before time measurement, an API (cudaDeviceSynchronize) has to be called to make sure that all cores have finished their tasks. This explicit synchronization is the overhead of measuring time on the GPUs. Therefore, the sum of layerwise timing on GPUs is longer than a full forward pass.

VGGNet has 2x CONV layers compared to AlexNet (Table 1). However, the number of operations is 20x that of AlexNet because VGGNet uses much larger feature maps. While other results 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. Similar to AlexNet, GoogleNet also employs LRN that significantly affects the performance on CPU for both TK1 and TX1. For example, it takes more than 55% of total time on TX1 CPU. GoogleNet has a layer, named Concat, that does not involve any computation, but concatenates the outputs from previous layers, thus involving memory operations only.

The difference between layerwise timing and full forward pass on GoogleNet is much larger than AlexNet and VGGNet as shown in Table 5. GoogleNet has many more layers than AlexNet and VGGNet and thus much more measuring overhead on GPUs. The measuring overhead may be larger than the compute time when the computation of a layer does not cost much time, *e.g.*, ReLU layers. Due to this measurement artifact, in Table 5, ReLU layers cost more time on GPUs than CPUs. This is a motivation for us to devise measurement techniques that can overcome these measurement overheads as we see later in §5.

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 terms of FLOPs as shown in Table 6. We observe that the computation of Scale and Eltwise costs more on GPUs than CPUs, which is again due to the measurement overhead on GPUs as discussed above. Interestingly, although ResNet has more FLOPs (2x) than GoogleNet, a full forward pass is faster than GoogleNet on TX1. This is because LRN of GoogleNet is very expensive: (55% of total time) on TX1 CPU. Moreover, GoogleNet has more CONV layers and the underlying matrix multiplication is smaller than that of ResNet. As GPU throughput is higher on matrix multiplication with larger size, ResNet is faster than GoogleNet on TX1 GPU.

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, the 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 speed of computing on GPUs. However, as the workspace is not the most significant part, cuDNN cannot reduce the memory usage of CNNs significantly.

Table 4 shows the memory requirement of weights and biases of CONV and FC layers, intermediate data, and workspace of CONV layers for each CNN – by parsing the model descriptor (*e.g.*, a prototxt file in Caffe). Table 4 also gives the measured memory usage of Caffe, running each CNN on these platforms. One can see that deeper CNNs (from AlexNet to ResNet) may not require more memory, especially for GoogleNet, which requires 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 computation 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 (scoring) 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 their dedicated memory, mobile platforms usually have a unified memory architecture, where GPU shares system memory with 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 system memory. As shown in Table 4, 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 each CNN (mostly weights and biases). For example, running AlexNet on TK1 GPU takes 560MB memory, which is 236MB more than TK1 CPU, while 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 overhead (*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 Analysis

FLOPs. As the throughput of both CPU and GPU is higher on the CNN with more FLOPs and a significant amount of memory operations are involved in a CNN computation, FLOPs cannot accurately reflect the compute time of a CNN. For example, ResNet is faster than GoogleNet on GPUs, though it involves more FLOPs. 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. Therefore, can one measure these layers instead of the entire network? However, this approach 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.

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 part of the layer, we will be able to accurately capture the resource requirements of these layers.

5 AUGUR

We aim to build a modeling tool that can estimate the resource requirements of any given CNN descriptor on specific mobile platforms without implementation and deployment. This way, we can take the costs into consideration during the design of a 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 used for matrix multiplications on CPUs and GPUs respectively.

[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 (up to 32GB/s) which is much slower than reading data from GPU memory (limit 200GB/s).

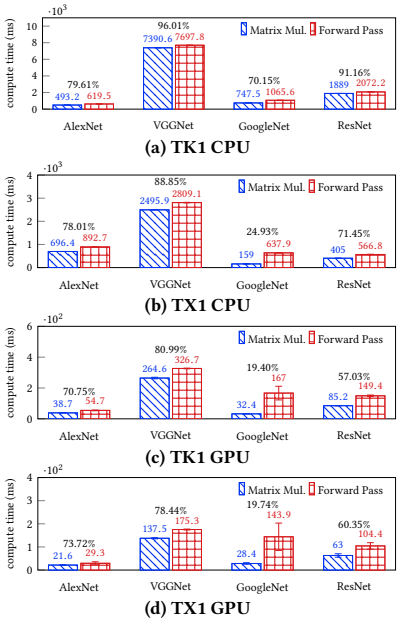
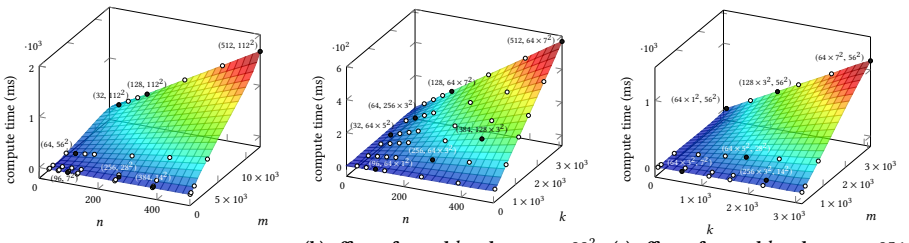
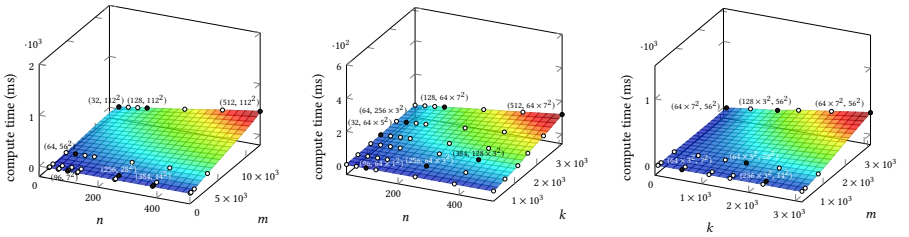


Figure 1: Matrix multiplication and forward pass of AlexNet, VGGNet, GoogleNet, and ResNet on mobile platforms.



(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 $n = 256$**

Figure 2: 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 $n = 256$**

Figure 3: Matrix multiplication on TX1 CPU with varying n, m, and k.

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 the 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 $[(O_w \cdot O_h) \times (K_w \cdot K_h \cdot K_d)][(K_w \cdot K_h \cdot K_d) \times N]$.

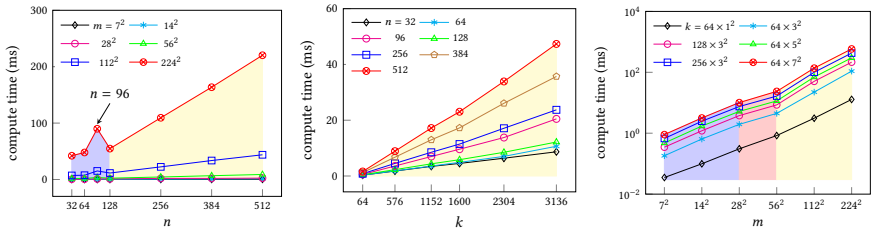
Mitigate measurement overhead: Layerwise timing measurement incurs heavy overhead on GPUs and causes a large deviation from a full forward pass. Moreover, the overhead is not fixed and varies over each measurement. As illustrated in Table 5 and 6, the measurement overhead (the difference between the sum of layerwise measurements and full forward pass) of GoogleNet (131 measurements) on TX1 GPU is 128 ms, while the overhead of ResNet (227 measurements) is 595 ms. Therefore, we need a way to mitigate the overhead for accurate timing of matrix multiplications.

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 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. 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 compute time of the matrix multiplication.

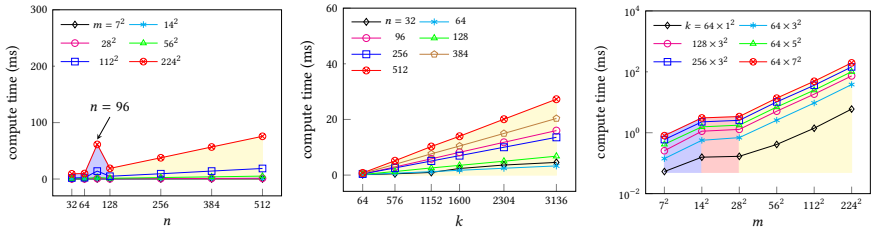
Fraction of forward pass spent by matrix multiplication: In Figure 1, we study the fraction of forward pass time spent by matrix multiplication (matmul) operations. We do so, by extracting the matmul operations, measuring them, and then comparing with the full forward pass measurement. Note that due to the above explained averaging methodology, measurement overhead for matmul operations in this section is negligible.

First, as seen in Figure 1a, matmul operations on TK1 CPU take a large portion of forward pass time – 79.61%, 96.01%, 70.15%, and 91.16% for AlexNet, VGGNet, GoogleNet, and ResNet, respectively. Note that this also approximates the time taken by CONV and FC layers from Table 2, 3, 5, and 6 (81.47%, 97.98%, 71.1%, and 88.57%). Second, the trend is similar on TX1 CPU, as depicted in Figure 1b, except GoogleNet (only about 25% time spent on matmul operations), which is caused by the particular combination of the architecture of TX1 CPU and GoogleNet as discussed in §4.1. Third,



(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 $n = 256$

Figure 4: 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 $n = 256$

Figure 5: Matrix multiplication on TX1 GPU with varying n, m, and k.

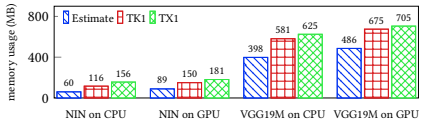


Figure 6: Memory estimate of NIN and VGG19M.

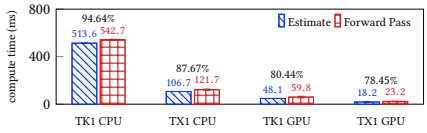


Figure 7: Timing estimate of NIN.

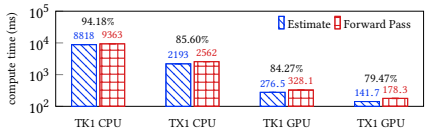


Figure 8: Timing profiling of VGG19M.

the trend on TK1 and TX1 GPUs is similar to the trend on TX1 CPU, as seen in Figure 1c and 1d. One thing to note is that while matmul operations of GoogleNet only take about 20% of the total time of forward pass, our previous measurement in Table 5, showed that CONV and FC layers take about 60% of the total forward pass time. We believe this is because the matmul operations are run 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.

In summary, for most cases, matmul operations take a large proportion (more than 60%) of the compute time of a CNN on mobile platforms. Thus, we can predict matmul time, to be able to approximately estimate the compute time of a CNN.

5.2 Modeling

So far, we have exactly measured matmul time. In this section, we aim to model this time, to be able to predict the compute time, 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.

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 commonly 1^2, 3^2, 5^2, 7^2, or 11^2. The depth of a kernel is usually the number of kernels in the 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, typically equal to three. The spatial size of output feature maps of a CNN m gradually reduces; it is common to have 224^2, 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 typical parameter settings, we carried out experiments on matmul with varying n, m, and k. 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 compared to the forward pass. Therefore, we do not consider the size of matrices for FC layers in the modeling.

Simple linearity on CPU: Figure 2 and 3 illustrate the performance of matmul on TK1 CPU and TX1 CPU, respectively. 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 labeled with coordinates to highlight the setting of varying parameters.

From Figure 2a, 2b, and 2c, it is observed that the compute time of matmul on TK1 CPU scales linearly with n, m, and k. The linearity can also be observed on TX1 CPU as depicted in Figure 3a, 3b, and 3c. Thus, we have a linear model per CPU device, which predicts the matmul time, given the matrix sizes.

Complex linearity on GPU: Figure 4 and 5 illustrate the performance of matmul with varying settings of n, m, and k on TK1 GPU and TX1 GPU, respectively. The compute time of matmul on GPUs exhibits more complex relationship with n, m, and k.

Figure 4a mainly depicts the effect of n, which is bipartite. For all the settings of m, the compute time has a monotonic relationship with n from $n = 32$ to 128, except $n = 96$ which 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 result is also found on TX1 GPU as shown in Figure 5a. Although TX1 GPU has more CUDA cores (256 compare to 192 cores in TK1 GPU) and generally computes matmuls faster than TK1 GPU, it also exhibits this pattern at $n = 96$. This artifact is related to the algorithm that determines how the CUDA cores compute matmul in parallel. Since cuBLAS is not an open-source library, it is hard to trace the exact

reason. However, it is indicated [2] that matmul works 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 TK1 GPU and TX1 GPU as depicted in Figure 4b and 5b, respectively. While the compute time increases with m on both TK1 GPU and TX1 GPU as depicted in Figure 4c and 5c, the effect of m is tripartite. The compute time has three separate linear relationships with k (different coefficients), *e.g.*, from 7^2 to 28^2, from 28^2 to 56^2, and from 56^2 to 224^2 on TK1 GPU, as highlighted by different regions in Figure 4c. In each such region, the compute time on different values of k linearly scales with m at mostly the same coefficient. Moreover, in the middle region (*i.e.*, between 28^2 and 56^2 in Figure 4c and between 14^2 and 28^2 in Figure 5c, the compute time increases with m slower than other two regions. This is especially true on TX1 GPU, where the region is much more flat and tends to plateau. This region should be the transition area, where cuBLAS adopts different schemes based on m and the number of CUDA cores to assign the workload of matmul to CUDA cores. The transition area is 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 of matmul 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, which estimates the compute time and memory 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 run the CNN. The memory includes data, parameters, and workspace. Then, Augur extracts matmuls from the computation of the CNN. Based on the models of TK1 and TX1 on matmul, *i.e.*, the linear fits obtained from Figure 2 and 4 for TK1, and Figure 3 and 5 for TX1, Augur calculates the compute time of individual matmuls and then uses their summation as the estimate of the compute time of the CNN.

To verify the accuracy of Augur, we model two CNNs (*i.e.*, NIN [20] and VGG19M[3]) and compare the estimates to the measured memory usage and compute time using Caffe. Figure 6 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 CNN computation. Note that the estimate of Augur is accurate on the memory usage of data, parameters, and workspace as discussed in §4.2.

Figure 7 and 8 evaluate the accuracy of Augur's compute time estimation of NIN and VGG19M, respectively. From Figure 7 and 8, we observe that the estimate based on only matmul can approximate the compute time of NIN and VGG19M on both CPUs and GPUs, with more than 78% accuracy for all the cases. Since matmul generally takes a larger proportion of the compute time on CPUs than on GPUs as discussed in §4.1, the estimate on CPUs (up to 94%) is closer to the actual compute time than on GPUs (up to 84%). Moreover, more powerful processing unit can perform matmul 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 (or TK1 GPU) is more accurate than TX1 CPU (TK1 GPU) for the same CNN.

In summary, Augur can estimate whether and 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 resource usage and compute time without implementation and deployment and tune the model to satisfy their specific needs.

[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.

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, 5, and 6), which commonly takes a dominant proportion of the compute time. Thus, matrix multiplication is currently exploited by Augur to estimate the compute time of a CNN. To obtain a more precise estimate, additional factors need to be taken into consideration, *e.g.*, memory operations 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.

7 CONCLUSION

In this paper, we aim to model the resource requirements of CNNs on mobile devices. 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 matrix multiplications form the core computations of a CNN, we profiled and modeled matrix multiplications on mobile platforms. Based on the measurement, profiling, and modeling, we built Augur that can estimate the compute time and memory usage 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 is a power tool that helps the design of CNNs for resource-constrained mobile devices.

ACKNOWLEDGMENTS

This work was supported in part by the Army Research Laboratory and accomplished under Cooperative Agreement Number W911NF-09-2-0053. The work was done while Zongqing Lu was at Pennsylvania State University.

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] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. 2016. An Analysis of Deep Neural Network Models for Practical Applications. *arXiv preprint arXiv:1605.07678* (2016).
[10] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. 2014. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115* (2014).
[11] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. 2016. MCDNN: An Approximation-Based Execution Framework for Deep Stream Processing Under Resource Constraints. In *International Conference on Mobile Systems, Applications, and Services (MobiSys'16).*
[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR'16).*
[13] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size. *arXiv preprint arXiv:1602.07360* (2016).
[14] Sergey Ioffe and Christian Szegedy. 2015. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *International Conference on Machine Learning (ICML'15).*
[15] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. In *ACM International Conference on Multimedia (MM'14).*
[16] Yong-Deok Kim, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin. 2016. Compression of Deep Convolutional Neural Networks for Fast and Low Power Mobile Applications. In *International Conference on Learning Representations (ICLR'16).*
[17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet Classification with Deep Convolutional Neural Networks. In *Neural Information Processing Systems Conference (NIPS'12).*
[18] Nicholas D Lane, Sourav Bhattacharya, Petko Georgiev, Claudio Forlivesi, Lei Jiao, Lorena Qendro, and Fahim Kawsar. 2016. Deepx: A Software Accelerator for Low-Power Deep Learning Inference on Mobile Devices. In *International Conference on Information Processing in Sensor Networks (IPSN'16).*
[19] Nicholas D Lane, Sourav Bhattacharya, Petko Georgiev, Claudio Forlivesi, and Fahim Kawsar. 2015. An Early Resource Characterization of Deep Learning on Wearables, Smartphones and Internet-of-Things Devices. In *International Workshop on Internet of Things towards Applications (IoT-App'15).*
[20] Min Lin, Qiang Chen, and Shuicheng Yan. 2014. Network in Network. In *International Conference on Learning Representations (ICLR'14).*
[21] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image recognition. In *International Conference on Learning Representations (ICLR'15).*
[22] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going Deeper with Convolutions. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR'15).*
[23] Jiaxiang Wu, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Cheng. 2016. Quantized Convolutional Neural Networks for Mobile Devices. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR'16).*