CNNdroid: GPU-Accelerated Execution of Trained Deep Convolutional Neural Networks on Android

Seyyed Salar Latifi Oskouei, Hossein Golestani, Matin Hashemi[*]
Sharif University of Technology
salarlatifi@ee.sharif.edu, hossein_golestani@ee.sharif.edu, matin@sharif.edu

Soheil Ghiasi
University of California, Davis
ghiasi@ucdavis.edu

ABSTRACT

Many mobile applications running on smartphones and wearable devices would potentially benefit from the accuracy and scalability of deep CNN-based machine learning algorithms. However, performance and energy consumption limitations make the execution of such computationally intensive algorithms on mobile devices prohibitive. We present a GPU-accelerated library, dubbed CNNdroid [1], for execution of trained deep CNNs on Android-based mobile devices. Empirical evaluations show that CNNdroid achieves up to 60X speedup and 130X energy saving on current mobile devices. The CNNdroid open source library is available for download at https://github.com/ENCP/CNNdroid

Keywords

Deep Learning, Deep Convolutional Neural Network (CNN), Mobile GPU, Performance Optimization, Low Energy Consumption, Open Source Software, Android, RenderScript

1. INTRODUCTION

Mobile platforms such as smartphones, wearable devices, tiny autonomous robots and IoT devices have been increasingly finding their way into many areas (Figure 1). Numerous applications, such as speech recognition and image recognition [2], would potentially benefit from local execution of accurate machine learning algorithms on mobile devices. Local execution allows data to stay on the mobile device and hence avoids latency issues of cloud-assisted processing.

Deep CNNs can achieve state-of-the-art results in terms of both prediction accuracy and scalability. However, they are highly computationally intensive, and hence, not practical on current mobile devices without acceleration.

---

[*]Contact author: Matin Hashemi, matin@sharif.edu

Figure 1: Example applications of deep CNNs in mobile systems. Image credits: IBM Research, Guardianlv, Nixie, Android Wear.

Many hardware-based solutions have been proposed for acceleration of deep CNNs [3, 4]. IBM has also introduced a neuromorphic CMOS chip for execution of learning applications on smartphones and IoT devices [5]. While promising, such solutions are still in early stages of development and not available on current mobile devices.

As opposed to hardware-based engines, GPU already exists in many current mobile devices and can be programmed completely in software. Therefore, parallel processing capabilities of mobile GPUs can be exploited to accelerate deep CNN computations on *current* mobile devices.

On server and desktop platforms, there exists many GPU-accelerated deep CNN libraries [6, 7, 8, 9, 10, 11, 12]. However, because of architecture differences (Section 2.1), mere porting of such libraries to mobile platforms yields suboptimal performance or is impossible in some cases (Section 2.2).

On mobile platforms, to the best of our knowledge, such GPU-accelerated libraries are not available. The few existing mobile libraries for CNN computations [13, 14, 15, 16] are limited to the processing power of multi-core mobile CPUs (Section 2.3).

We present an open source GPU-accelerated library, dubbed CNNdroid, which is specifically designed and optimized for execution of trained deep CNNs on Android-based mobile devices. The main highlights of CNNdroid are as follows.

1. Support for nearly all CNN layer types (Section 3.1).

2. Compatible with CNN models trained by common desktop/server libraries, namely, Caffe [6], Torch [7] and Theano [8] (Section 3.2).

3. Easy to configure and integrate into any Android app without additional software requirements (Section 3.3).

4. User-specified maximum memory usage (Section 3.4).

5. GPU or CPU acceleration of supported CNN layers (Section 3.5).

6. Automatic tuning of performance (Section 3.6).

7. Up to 60X speedup and up to 130X energy saving on current mobile devices (Section 4).

2. BACKGROUND AND RELATED WORK

2.1 Comparing Mobile and Desktop GPUs

A modern graphics processing unit (GPU), in addition to computer graphics, can be programmed for general purpose computations as well. While desktop GPUs have long been programmable, major mobile chip manufacturers have recently made the GPU hardware available for general purpose computations. Due to strict area and power constraints, mobile GPUs have important differences with their desktop counterparts.

A modern mobile GPU is typically composed of several programmable parallel computing units called Shader Cores (SC). Every shader core is composed of several parallel ALUs. For example, Samsung Exynos 5433 chip is composed of ARM A53/A57 CPU and Mali T-760 GPU (Figure 2). Each SC in T-760 GPU has two 128-bit ALUs in VLIW format. Each 128-bit ALU is capable of performing SIMD operations, i.e., two 64-bit, four 32-bit or eight 16-bit operations in parallel [17]. In comparison with desktop GPUs, the above parallel ALU architecture relies more on software and compiler than dynamic hardware scheduler in efficient execution of parallel threads.

More importantly, fast shared memory in thread blocks which are present in desktop GPUs and widely employed in many CUDA-based desktop libraries are not available in mobile GPUs.

There are some differences on the software side as well. For example in RenderScript, Android's parallel computing platform [18], thread synchronization is not available. In addition, there must be a one-to-one correspondence between parallel threads and the data items inside one of the memory buffers that parallel threads work on.

2.2 Comparing CNNdroid with Desktop Libraries

On server and desktop platforms, there exists many libraries such as Caffe [6], Torch [7], Theano [8], TensorFlow [9], cuDNN [10], cuda-convnet [11], and Velesnet [12], which employ GPU-based parallel processing for acceleration of deep CNN computations. However, the acceleration methodologies and parallel algorithms of such libraries could not be directly utilized in mobile platforms due to the existing hardware and software differences.

For example in Caffe [6], the convolution operation is unrolled and converted to matrix multiplication, which requires
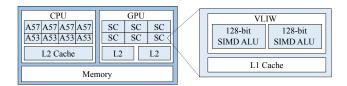

Figure 2: Example: Exynos 5433 mobile processor with ARM A53 / A57 CPU and Mali T-760 GPU (SC: Shader Core, VLIW: Very Long Instruction Word, SIMD: Single Instruction Multiple Data).

considerable amount of memory and therefore is not suitable for mobile devices with small cache and memory sizes. As another example, the parallel algorithm in Theano [8] is similar to CNNdroid but without efficient use of SIMD units in mobile GPUs. Refer to Section 3.5 for details.

More importantly, desktop libraries take advantage of thread management facilities provided by desktop GPUs and CUDA framework, such as fast shared memory and thread synchronization, which are not available in mobile GPUs and RenderScript.

2.3 Comparing CNNdroid with Mobile Libraries

On mobile platforms, to the best of our knowledge, only few deep CNN libraries exist [13, 14, 15, 16]. All such libraries, including Caffe Mobile [13] and Torch Mobile [14], are limited to the processing power of multi-core mobile CPUs, while CNNdroid efficiently employs both GPU and CPU (Section 3.5).

In addition, CNNdroid is compatible with CNN models trained by Caffe [6], Torch [7] and Theano [8], which facilitates the process of porting the trained models to mobile devices (Section 3.2).

The existing libraries require installation of Android NDK alongside Android SDK, while in CNNdroid, only Android SDK is required.

3. CNNDROID LIBRARY

3.1 CNN Layer Types

CNNdroid library supports nearly all common types of CNN layers, namely, convolution, max/mean pooling, fully connected, rectified linear unit, local response normalization and softmax. Detailed description of every layer type and its corresponding parameters are available in the library documentations [1]. Other new layers may be added as well, due to the open source nature of the library.

3.2 Model Preparation

Model Conversion Scripts: Figure 3 shows an overview of the steps involved in deploying trained CNN models on mobile devices. CNNdroid library provides a set of scripts which take the models trained by common desktop/server libraries, namely, Caffe [6], Torch [7] and Theano [8], as input and convert them into CNNdroid format. Therefore, the models which are trained by these libraries can be executed by CNNdroid library on mobile devices.

It is possible to write similar scripts for other libraries as well. CNNdroid uses MessagePack serialization format [19] for storing layer parameters in the trained model. Detailed procedure is presented in the library documentations [1].

NetFile: The developer needs to prepare a *.txt* file, called *NetFile*, similar to the *.prototxt* file in Caffe [6]. The *NetFile* specifies layer setup of the trained model, i.e., order of the CNN layers along with their configurations, e.g., padding and stride values of convolution layers. Figure 4 presents an example. Detailed instructions for preparing the *NetFile* as well as a few examples are included in the library documentations [1].

The *NetFile* also specifies three configuration parameters (Figure 4), namely, `allocated_ram` which specifies the maximum amount of memory that CNNdroid engine is allowed to allocate at runtime (Section 3.4), `execution_mode` which selects between sequential or parallel execution modes of the library (Section 3.5), and `auto_tuning` which specifies whether or not auto-tuning is turned on (Section 3.6).

3.3 Model Execution

Once the trained model and its corresponding *NetFile* are both uploaded to mobile device (Figure 3), the model can be executed in the target Android application in a few simple steps as described below (Figure 5).

The first step is to include the provided CNNdroid library files. Note that CNNdroid library is self-sufficient and does not require installation of third-party libraries. In addition, it does not require installation of Android NDK alongside Android SDK.

Next, `RenderScript` and `CNNdroid` objects are constructed (Figure 5, steps 2 and 3). The CNNdroid constructor takes the provided *NetFile* as input and automatically creates the corresponding objects for the network layers.

Finally, `compute` function of the constructed `CNNdroid` object is called, which automatically executes the trained model on either a single image or a batch of images.

3.4 Memory Allocation

The trained CNN model, which is uploaded to SD card of the mobile device, contains layer parameters in form of matrices. Inside the `compute` function (Figure 5, step 5), and before execution of every layer, the corresponding matrices are automatically loaded from SD card into RAM, which incurs an overhead.

In order to reduce this overhead, CNNdroid selects certain layers and keeps their data in RAM, while other layers are allocated and de-allocated every time. The selection procedure is automatically done in `CNNdroid` constructor (Figure 5, step 3). Starting from the largest layer, as many layers as possible are selected, till the sum of their memory sizes reaches the maximum developer-specified amount, i.e., the `allocated_ram` parameter in the *NetFile*.

Note that the `allocated_ram` parameter cannot be arbitrarily large due to practical limitations. For example, Android 5 limits the memory usage of every app to 512MB.

3.5 Acceleration Methods

Different methods are employed in acceleration of different layers in CNNdroid. Convolution and fully connected layers, which are data-parallel and normally more compute intensive, are accelerated on the mobile GPU using Render-Script framework.

A considerable portion of these two layers can be expressed as dot products. In specific, in the convolution layer, kernels get convoluted with the input frames, and in the fully connected layer, the computation can be expressed as a ma-
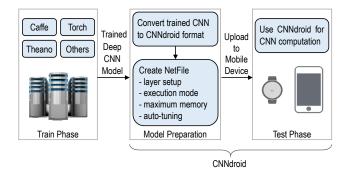

Figure 3: Overview of CNNdroid's model deployment procedure.

```
root_directory: "/sdcard/AlexNet/"
allocated_ram: 50
execution_mode: "parallel"
auto_tuning: "on"
layer {
 type: "Convolution"
 name: "conv1"
 parameters_file: "param_conv1.msg"
 pad: 0
 stride: 4
 group: 1
}
layer {
 type: "ReLU"
 name: "ReLU1"
}
layer {
 type: "LRN"
 name: "norm1"
 local_size: 5
 alpha: 0.0001
 beta: 0.75
 norm_region: "across_channels"
}
layer {
 type: "Pooling"
 name: "pool1"
 pool: "max"
 kernel_size: 3
 pad: 0
 stride: 2
}
```

Figure 4: Example NetFile showing three layers of AlexNet [20], along with `allocated_ram`, `execution_mode` and `auto_tuning` parameters.

```
// 1) Import CNNdroid library
import network.CNNdroid;

 ...

// 2) Construct Renderscript object
RenderScript myRenderScript = RenderScript.create(this);

// 3) Provide NetFile and construct CNNdroid object
String NetFile = "/sdcard/AlexNet/AlexNet_NetFile.txt";
CNNdroid myCNN = new CNNdroid(myRenderScript, NetFile);

// 4) Prepare your input, which can be
//    a single image or a batch of images
float[][][]   inputSingle = loadSingleInput();
float[][][][] inputBatch = loadBatchInput();

// 5) Call 'compute' function for CNN execution
//    and receive the result as an object
Object output = myCNN.compute(inputBatch);
```

Figure 5: Simple steps involved in using CNNdroid for execution of trained deep CNN models on Android apps. Refer to the library documentation for up-to-date details and sample projects [1].

trix to vector multiplication. The dot products are more efficiently calculated on SIMD units of the target mobile GPU. Therefore, we divide the computation in many vector operations and use the pre-defined *dot* function of the RenderScript framework. In other words, we explicitly express this level of parallelism in software, and as opposed to CUDA-based desktop libraries, do not leave it to GPU's hardware scheduler.

Comparing with convolution and fully connected layers, other layers are relatively less compute intensive and not efficient on mobile GPU. Therefore, they are accelerated on multi-core mobile CPU via multi-threading. Since ReLU layer usually appears after a convolution or fully connected layer, it is embedded into its previous layer in order to increase the performance in cases where multiple images are fed to the CNNdroid engine.

In addition to above parallel implementations, CNNdroid also includes sequential (single-thread) implementations of all layers. The execution will be sequential or parallel depending on the `execution_mode` parameter specified in the *NetFile* (Figure 4).

3.6 Auto-Tuning

In order to reach better performance across different Android based mobile devices, our GPU-accelerated parallel algorithms are developed with tuning parameters which select the amount of work assigned to parallel GPU threads and the amount of work assigned to SIMD ALUs in execution of every GPU thread. The tuning parameters basically determine the granularity of parallelism.

If turned on in the *NetFile* (Figure 4), the auto-tuner is automatically executed when the Android app is launched for the first time. It executes the CNN model for a number of predefined scenarios on the mobile device, measures their runtime and saves the optimum tuning parameters for future executions. As a result, the first launch of the application takes much longer time. For the purpose of clear and fair comparisons, auto-tuning is turned off in our experiments in Section 4.

4. EMPIRICAL EVALUATION

CNNdroid is empirically evaluated on two mobile devices, namely, Samsung Galaxy Note 4 and HTC One M9. We employ three well-known benchmark CNNs, namely, LeNet network for MNIST dataset [21], Alex Krizhevsky's network for CIFAR-10 (Alex's CIFAR-10) [22] and Alex Krizhevsky's network for ImageNet 2012 dataset (AlexNet) [20].

Layer setup of the benchmark CNNs are shown in Figure 6. We also measured the storage and memory requirement of the benchmark CNNs when ported to CNNdroid format. The results are reported in Figure 7.

We execute forward paths of the benchmark CNNs on the mobile devices and measure their accuracy, runtime and energy consumption. All benchmarks accept batches of 16 images as input in our experiments. Before running every experiment, mobile devices are fully charged and put into airplane mode and minimum screen brightness. The measurements reported below are only for CNN execution and not for loading the network parameters from SD card, because in our benchmarks network parameters are loaded only once in the beginning but CNN execution is performed on every input image.

Layer	LeNet	Alex's CIFAR-10	AlexNet
1	Conv	Conv	Conv+ReLU
2	Pooling	Pooling+ReLU	LRN
3	Conv	Conv+ReLU	Pooling
4	Pooling	Pooling	Conv+ReLU
5	FC+ReLU	Conv+ReLU	LRN
6	FC	Pooling	Pooling
7	-	FC	Conv+ReLU
8	-	FC	Conv+ReLU
9	-	-	Conv+ReLU
10	-	-	Pooling
11	-	-	FC+ReLU
12	-	-	FC+ReLU
13	-	-	FC

Figure 6: Layer setup of benchmark CNNs.

	SD Card Storage (MB)	RAM (MB)
LeNet	2	~10
Alex's CIFAR-10	0.7	~20
AlexNet	290	~300

Figure 7: Storage and memory requirements of benchmark CNNs when ported to CNNdroid format.

4.1 Accuracy

In order to measure CNNdroid accuracy, output of the last network layer in both CNNdroid and Caffe are compared for the same input. The resulting mean square error is in the order of 10^{-12}, which means there is no meaningful difference and CNNdroid is correctly implemented.

4.2 Performance

Figure 8.a shows the total measured runtime for CPU-only sequential CNN implementation as well as the speedup gained by GPU acceleration. The reported values are the average of ten executions.

Note that real-time performance is achieved on mobile devices in LeNet and Alex's CIFAR-10 benchmarks. For instance on the HTC One M9 device, the accelerated implementation achieved 60.2 and 32.2 frames per second for LeNet and Alex's CIFAR-10 benchmarks, respectively.

We also measured runtime of the heaviest convolution layer in order to observe direct impact of the GPU-based acceleration (Figure 8.b). The highest achieved speedup is 63.4X for AlexNet benchmark on Galaxy Note 4 device with Mali-T760 GPU. This GPU has 6 shader cores, each with two 128-bit ALUs. Since all elements of the matrices in our CNN model are 32-bit floating point values, a maximum of $6 \times 2 \times \frac{128}{32} = 48$ operations may run in parallel. In other words, the maximum theoretically achievable speedup is 48X. Therefore, the measured 63.4X speedup most probably comes from other factors such as software language performance of RenderScript(c99) over Java or cache effects.

Note that the benchmark CNNs in our experiments accept batches of 16 images as input, and the runtime values reported in Figure 8 are per image. It is recommended to process a batch of input images rather than a single image to get higher performance in CNNdroid.

As for the performance comparison of our mobile devices, we see that the overall speedup in AlexNet, which is a large network, is approximately 30% higher on Galaxy Note 4 compared with HTC One M9. This can be either the result of lower GPU frequency of HTC One M9 or its aggressive throttling policy in order to prevent overheating issues in long runtimes.

(a)		Sequential Runtime (ms)	Accelerated Runtime (ms)	Speedup Rate
Samsung Galaxy Note 4	LeNet	62	12.8	4.84
	Alex's CIFAR-10	313	25.3	12.37
	AlexNet	20767	481.7	43.11
HTC One M9	LeNet	81	16.6	4.88
	Alex's CIFAR-10	326	31	10.51
	AlexNet	21382	709	30.16

(b)		Sequential Runtime (ms)	Accelerated Runtime (ms)	Speedup Rate
Samsung Galaxy Note 4	LeNet	44	1.8	24.44
	Alex's CIFAR-10	162	7.5	21.6
	AlexNet	5876	92.6	63.45
HTC One M9	LeNet	62	4.3	14.42
	Alex's CIFAR-10	168	8.7	19.31
	AlexNet	5828	152	38.34

Figure 8: Average runtime of (a) the entire CNN and (b) the heaviest convolution layer, per image in a batch of 16 images, and the corresponding speedup rate.

4.3 Energy Consumption

We measured power and energy consumption per image for AlexNet benchmark on HTC One M9 by employing "Qualcomm Trepn Profiler" application [23].

The GPU accelerated execution consumes around $523\ mW$ power and $0.4\ J$ energy while the CPU-only sequential execution consumes $2338\ mW$ power and $51.6\ J$ energy. As a result, the GPU accelerated execution consumes $51.6 \div 0.4 = 129$X less battery energy.

It should be noted that we observed about 20% variability in our measurements which is expected since Trepn Profiler provides a software-only method for measuring energy consumption of a single app.

5. CONCLUSIONS

We introduced CNNdroid, an open source GPU accelerated deep CNN library for Android-based mobile devices. Empirical evaluations demonstrated up to 60X speedup and up to 130X energy saving. The source code, documentation and sample projects are published online [1].

6. REFERENCES

[1] CNNdroid open source GPU-accelerated library. https://github.com/ENCP/CNNdroid.

[2] Inchul Song, Hyun-Jun Kim, and Paul Barom Jeon. Deep learning for real-time robust facial expression recognition on a smartphone. In *IEEE International Conference on Consumer Electronics*, pages 564–567, Jan 2014.

[3] Yu-Hsin Chen, Tushar Krishna, Joel Emer, and Vivienne Sze. 14.5 eyeriss: an energy-efficient reconfigurable accelerator for deep convolutional neural networks. In *IEEE International Solid-State Circuits Conference*, pages 262–263, Jan 2016.

[4] Mohammad Motamedi, Philipp Gysel, Venkatesh Akella, and Soheil Ghiasi. Design space exploration of fpga-based deep convolutional neural networks. In *Asia and South Pacific Design Automation Conference*, pages 575–580, Jan 2016.

[5] Paul A Merolla, John V Arthur, Rodrigo Alvarez-Icaza, Andrew S Cassidy, Jun Sawada, Filipp Akopyan, Bryan L Jackson, Nabil Imam, Chen Guo, Yutaka Nakamura, Bernard Brezzo, Ivan Vo, Steven K Esser, Rathinakumar Appuswamy, Brian Taba, Arnon Amir, Myron D Flickner, William P Risk, Rajit Manohar, and Dharmendra S Modha. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, 345(6197):668–673, 2014.

[6] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[7] Torch. http://torch.ch/. Accessed 2016-08-01.

[8] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference*, 2010.

[9] TensorFlow. https://www.tensorflow.org. Accessed 2016-08-01.

[10] Nvidia cuDNN. https://developer.nvidia.com/cudnn. Accessed 2016-08-01.

[11] cuda-convent. https://code.google.com/p/cuda-convnet/. Accessed 2016-08-01.

[12] Velesnet. https://velesnet.ml/. Accessed 2016-08-01.

[13] Caffe Android Library. https://github.com/sh1r0/caffe-android-lib. Accessed 2016-08-01.

[14] Torch-7 for Android. https://github.com/soumith/torch-android. Accessed 2016-08-01.

[15] A convolutional neural network for the Android phone. https://github.com/radiodee1/awesome-cnn-android-python. Accessed 2016-08-01.

[16] Facial attractiveness prediction on Android. https://github.com/eldog/fmobile. Accessed 2016-08-01.

[17] ARM. Mali-T600 Series GPU OpenCL, Version 1.1.0, Developer Guide. Accessed 2016-08-01.

[18] Android RenderScript Developers Guide. http://developer.android.com/guide/topics/renderscript/compute.html. Accessed 2016-08-01.

[19] Messagepack. http://msgpack.org/index.html.

[20] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, 2012.

[21] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998.

[22] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.

[23] Trepn power profiler. https://developer.qualcomm.com/software/trepn-power-profiler.