Fast and Energy-Efficient CNN Inference on IoT Devices

Mohammad Motamedi  
ECE Department  
University of California, Davis  
Email: mmotamedi@ucdavis.edu

Daniel Fong  
ECE Department  
University of California, Davis  
Email: dfong@ucdavis.edu

Soheil Ghiasi  
ECE Department  
University of California, Davis  
Email: ghiasi@ucdavis.edu

Abstract—Convolutional Neural Networks (CNNs) exhibit remarkable performance in various machine learning tasks. As sensor-equipped internet of things (IoT) devices permeate into every aspect of modern life, it is increasingly important to run CNN inference, a computationally intensive application, on resource constrained devices. We present a technique for fast and energy-efficient CNN inference on mobile SoC platforms, which are projected to be a major player in the IoT space. We propose techniques for efficient parallelization of CNN inference targeting mobile GPUs, and explore the underlying tradeoffs. Experiments with running SqueezeNet on three different mobile devices confirm the effectiveness of our approach. For further study, please refer to the project repository available on our GitHub page: https://github.com/mtmd/MobileConvNet.

I. INTRODUCTION
Convolutional Neural Networks (CNNs) have been extensively used in different image and video processing applications [1], [3], [7]. Even though CNNs have remarkable performance, they are computationally intensive. There have been different proposals for accelerating CNNs using GPUs ([2], [5]), FPGAs ([8], [4]) and ASICs. Among these platforms, GPUs are the main platform for accelerating Convolutional Neural Network due to their high performance and ease of use.

There are two classes of GPUs: server-class and mobile-class. Server-class GPUs have more computational capabilities and consume a considerable amount of energy. These GPUs are used in cloud-based computing for training CNNs and processing computations which are offloaded to a cloud from mobile devices. Accelerating CNNs on server-class GPUs is very well studied. Nowadays, many applications rely on cloud computing to take advantage of machine learning techniques with CNNs. However, cloud computing is not always the best solution for several reasons. First, using a wireless media for transferring data to a cloud requires a considerable amount of energy. Second, sometimes it is not possible to offload computations to a cloud in locations with weak signal reception. Finally, due to privacy concerns, using remote processing infrastructures is not always a solution.

Unlike server GPUs, mobile GPUs have limited hardware resources and they are designed to work within restricted power budgets. In this paper, we have used RenderScript for accelerating CNNs on mobile GPUs. RenderScript is a framework for utilizing heterogeneous computing on Android phones. During execution, the runtime engine distributes the computation on available processing elements such as CPU cores and GPU.

In this paper, we propose a solution for utilizing RenderScript features to implement CNNs efficiently on Android devices. Moreover, we will explain that the design parameters which yield the minimum execution time vary between different platforms and layers of a CNN. Subsequently, we implement and optimize SqueezeNet [1] on three different mobile devices and find the optimal design parameters for each.

The approach we propose in this paper can be used for accelerating any CNN efficiently. Having an execution time of less than a quarter of a second and an energy consumption of half a joule, the proposed algorithm makes it feasible to locally use CNNs on mobile devices.

II. CONVOLUTIONAL NEURAL NETWORKS
Convolutional Neural Networks (CNNs) have millions of parameters, which are obtained during a training procedure. In this work, we focus on the acceleration of the forward path using mobile GPUs.

Each CNN has multiple convolutional layers. Each layer uses 3D kernels for feature extraction. The result of multiple feature extractions are added together to create Output Feature Maps. The convolution operation for two filter banks that each of them has three kernels is shown in Figure (1). The number of input feature maps, output feature maps, and the output size are numInputLayers, numOutputLayers, and outputWidth \times outputHeight, respectively.

One kernel has dimension $K \times K$, and a layer has numInputLayers \times numOutputLayers of these kernels. Each pixel in an Output Feature Map is the sum of a convolutions between inputs and the respective kernels. To generate adjacent pixels in an Output Feature Map, the kernel
SqueezeNet has eight fire layers and two convolutional layers. The input to the very first layer is a 224 x 224 RGB image. SqueezeNet has four different types of layers: convolutional, pooling, softmax, and fire. The first three layers are commonly used in many CNN architectures. Fire layers are unique to SqueezeNet and consist of three convolutional layers: one squeeze layer and two expand layers. More than 90% of the execution time in CNNs is spent in convolutional layers. In this section, we propose an approach for accelerating the convolution using RenderScript on mobile devices.

### Parallel Algorithm Design

A sequential implementation of the convolution operation is shown in Figure (2). Loops #2 to #6 perform a 3D convolution (both input and kernel are 3D data structures). Loop #1, repeats the 3D convolution for different kernels in order to compute different layers of the output. Recall that each layer of the output is the result of a 3D convolution of the input with a distinct filter bank. In Figure (2), S (stride) is the number of pixels that a kernal slides during each step of the convolution. The convolution operation takes a major part of the execution time in CNNs. In this section, we propose an approach for accelerating the convolution using RenderScript on mobile devices.

A prototype of vectorized dot function which is offered in RenderScript. The RenderScript type float4 is used to represent a vector of four elements of type float. The input of this function are two vectors of type float and the output is the dot product of these two vectors.

#### A. Parallel Computation of Output Elements

Output of each convolution layer is a 3D matrix which includes `numOutputElements` elements (Equation (1)). Computing these elements is a perfectly parallel workload. In theory, it is possible to assign one thread per element and execute all these threads concurrently. To implement this in RenderScript, it is required to define an allocation with dimensions matching the size of the output matrix. Successively, for each element of that allocation, it is required to compute the result of a dot product between the corresponding window of the input and a specific kernel. This dot product is performed using the pseudo code shown in Figure (3). Notice that this code has to be executed for every element of the output allocation. In each allocation, every thread has an index `x`. For the aforementioned output allocation `x ∈ [0, numOutputElements - 1]`. In each thread, index `x` is used to determine values of `w`, `h`, and `m` as it is shown in Equations (2), (3), and (4) respectively. The value of `(m, w, h)` in each thread is unique. These parameters help a thread to work on a distinct part of inputs and generate a specific element in the output matrix.

\[
\begin{align*}
    w &= x \mod outputWidth \quad (2) \\
    h &= \left\lfloor \frac{x}{outputWidth} \right\rfloor \mod outputHeight \quad (3) \\
    m &= \left\lfloor \frac{x}{outputWidth \times outputHeight} \right\rfloor \quad (4)
\end{align*}
\]

#### B. Utilizing Vectorized Operations

RenderScript offers various vector math functions that interpret their input arguments as a representation of vectors. Currently, the maximum supported length for these vectors is four. These built-in functions can be efficiently mapped on supported GPUs. A prototype of the vectorized dot function is shown in Figure (4). In this case, inputs are two vectors, each of which includes four floating point numbers. The function performs a dot operation between the two vectors and returns the result as a floating point number.

We use the vectorized dot function for further acceleration of the convolution. To do so, it is required to change the implementation offered in the pseudo code of Figure (3). As previously discussed, this code computes the dot product between a certain window of the input and a kernel. This process is performed in parallel for all elements of the output allocation. However, the process itself is sequential (three nested loops in Figure (3)). In the following subsections, we

---

1. Allocation is a class in RenderScript which is designed for passing data to a RenderScript kernel. More information is available on Android website: https://developer.android.com/reference/android/renderscript/Allocation.html
The function perform a vectorized dot product, and accumulate the result. It is sufficient to read them as vectors with a length of four, respectively. These parameters determine the location of the output element that thread \( x \) generates. Therefore, in order to generate the output of the convolution in a reordered format, it is required to change these parameters. For example, the result of computations in the second thread \((x = 1)\) is stored in the second location of the output memory. After reordering, however, the second element of the output array should be \((m = 1, w = 0, h = 0)\). This is illustrated in Figure (7). To create the output in the reordered format, it is required to generate indexes for stacks of four layers, instead of a single layer (Figure (7)). Hence, we start indexing the second row when we have indexed all first rows of all four layers. Therefore, \( w \) and \( h \) can be computed using Equations (7) and (8), respectively. For computing the value of \( m \) (layer index), it is required to see which stack and layer a particular output belongs to (Figures (5) and Figure (7)). Equation (9) computes the value of \( m \).

\[
\begin{align*}
    w &= \lfloor x / 4 \rfloor \% \text{outputWidth} \quad (7) \\
    h &= \lfloor x / 4 \times \text{outputWidth} \rfloor \% \text{outputHeight} \quad (8) \\
    m &= (x \% 4) + \lfloor x / 4 \times \text{outputWidth} \times \text{outputHeight} \rfloor \times 4 \quad (9)
\end{align*}
\]

Using Equations (7), (8), and (9), it is possible to generate the output in the reordered format. Such an output can be directly used as the input to the next layer without additional processing overhead. The pseudo code in Figure (8) shows the final implementation.

D. Optimizing Thread Granularity

There is a trade-off between the number of threads and the amount of code that should be executed in each thread. Decreasing the number of threads increases the amount of code per thread (i.e., each thread has to compute multiple elements of output instead of one). Increasing the number of threads beyond some hardware limit will not improve the execution time since there is not sufficient resources for parallel execution of all of these threads. In addition, when a thread is responsible for generating more than one output value, it is possible to decrease memory access by data reusability. Therefore, in this problem, defining the highest possible number of threads is...
E. Pooling and Softmax

There are two types of pooling functions in SqueezeNet: max pooling and average pooling. As their names explain, the former finds the maximum and the latter computes the average of numbers in a given window. We have used vectorized function \( \text{fmax} \) and \( \text{sum} \) for RenderScript based implementation of these functions. Implementations are analogous to convolution layers. The execution time of the \text{softmax} function is negligible; hence, a GPU based implementation of this function is not required.

In this section we explained the approach used for RenderScript based implementation of SqueezeNet. For simplicity, we skipped a considerable amount of details. For further study, please refer to the project repository available on our GitHub page\(^2\).

IV. Experimental Results

We have used three different mobile devices for evaluating the proposed RenderScript based parallel algorithm. The specifications of these phones are shown in Table (II). These phones are equipped with three major Qualcomm Snapdragon SoCs. In order to increase the precision, all experiments of this section have been repeated 10 times, the average is computed and reported. During the experiments the phone’s background processes were stopped, placed in airplane mode, and fully-dimmed their screen brightness.

In this section, we present the results of executing the proposed algorithm on these phones and perform some analysis. We have accelerated SqueezeNet\(^1\) using the proposed approach. SqueezeNet has two normal convolutional layers and nine fire layers. Each fire layer has three convolutional layers: two squeeze layers and one expand layer. We use \( \text{Fn SQn} \) and \( \text{Fn EXn} \) notations to refer to squeeze and expand layers, respectively.

A. The Effect of Thread Granularity

As we explained in section (III-D), for each convolutional layer in SqueezeNet, there is a finite set of valid values for \( g \). The result of implementation of SqueezeNet for different values of this set is shown in Figure (10).

As Figure (10) illustrates, having the finest thread granularity (\( g = 1 \)) is not the optimal solution for any layer. By increasing the value of \( g \), data reusability increases, thereby reducing the execution time. However, after some threshold, any further increase of the value of \( g \) exacerbates the execution time. Notice that implementations based on very large values of \( g \) do not use the available parallel resources efficiently.

Thread granularity plays an important role in determining the execution time. In Table (III) we show the execution time of SqueezeNet for three different smart phones. For all of these phones, using optimal thread granularity drastically decreases the execution time. Even on modern platforms, such as Galaxy S7, optimal thread granularity can yield a speedup of 2.52X compared to a pessimal thread granularity. Finally, the optimal solution for thread granularity for some layers of SqueezeNet are shown in Table (I). For some layers, such as F2 EX1, F3 EX1, and F4 EX1, the optimal solution is identical for different platforms. However, the optimal thread granularity varies based on the convolution layer specifications and the target hardware.

| Hardware Specifications of Mobile Devices which are Used in Evaluations. |
|-----------------------------|---|------------|-------------|
| SoC                        | Tech | CPU Frequency | GPU         |
| Samsung Galaxy S7          | Snapdragon 820 | 14 nm     | 0.307 - 2.15 GHz | Adreno 330 @624 MHz |
| Huawei Nexus 6P            | Snapdragon 810 | 20 nm     | 0.384 - 1.96 GHz | Adreno 430 @650 MHz |
| LG Nexus 5                 | Snapdragon 800 | 28 nm     | 0.3 - 2.27 GHz  | Adreno 330 @450 MHz |

| Effect of Thread Granularity, Choosing Optimal Granularity Over Pessimal Improves the Execution Time by at Least 2X. |
|-----------------------------|-----------------|-----------------|
| Fire Layers (ms)             | Convolutional Layers (ms) | Speedup |
| Optimal                     | Pessimal        | Speedup         |
| Galaxy S7                   | 268.94          | 852.36          | 3.17X         |
| Nexus 6P                    | 234.72          | 543.01          | 2.31X         |
| Nexus 5                     | 325.6           | 828.35          | 2.56X         |
TABLE I
OPTIMAL THREAD GRANULARITIES FOR SQUEEZE NET ON DIFFERENT PLATFORMS.

|                | Conv1 | F2EX1 | F2EX3 | F3EX1 | F3EX3 | F4EX1 | F4EX3 | F5EX1 | F5EX3 | F6EX1 | F6EX3 | F7EX1 | F7EX3 |
|----------------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| Galaxy S7      | G6    | G8    | G4    | G8    | G8    | G4    | G12   | G12   | G6    | G4    |
| Nexus 6P       | G6    | G8    | G4    | G8    | G4    | G8    | G4    | G16   | G6    | G6    | G6    |
| Nexus 5        | G12   | G8    | G16   | G8    | G16   | G8    | G32   | G8    | G12   | G12   | G12   | G12   |

Fig. 10. Execution time of layers of SqueezeNet for different thread granularities. Numbers are measured on Nexus 5. Highest number of threads (g = 1) has the worst execution time.

TABLE IV
EXECUTION TIME (MILLISECOND) OF LAYERS OF SQUEEZE NET USING SEQUENTIAL, PARALLEL AND IMPRECISE PARALLEL ALGORITHMS.

| Algorithm       | Conv 1 | Fire 2 | Fire 3 | Fire 4 | Fire 5 | Fire 6 | Fire 7 | Fire 8 | Fire 9 | Conv 10 |
|-----------------|--------|--------|--------|--------|--------|--------|--------|--------|--------|---------|
| Galaxy S7       |        |        |        |        |        |        |        |        |        |         |
| Sequential      | 2523.64| 526.36 | 561.73 | 2029.00| 490.09 | 1044.27| 1138.00| 1929.91| 428.91 | 1567.91 |
| Precise Parallel| 55.82  | 25.55  | 26.00  | 46.37  | 20.82  | 33.91  | 33.92  | 58.55  | 23.82  | 103.73  |
| Imprecise Parallel| 7.10 | 18.10  | 15.60  | 21.50  | 20.70  | 20.10  | 23.10  | 22.70  | 17.80  | 19.00   |
| Nexus 6P        |        |        |        |        |        |        |        |        |        |         |
| Sequential      | 3275.55| 711.82 | 743.82 | 2652.18| 678.45 | 1422.82| 1530.27| 2606.73| 617.00 | 2954.09 |
| Precise Parallel| 52.36  | 18.54  | 19.45  | 44.82  | 17.82  | 29.00  | 27.91  | 55.73  | 21.45  | 82.55   |
| Imprecise Parallel| 12.40 | 9.87   | 8.89   | 12.07  | 7.55   | 9.11   | 10.48  | 16.04  | 13.78  | 15.40   |
| Nexus 5         |        |        |        |        |        |        |        |        |        |         |
| Sequential      | 8873.45| 1835.91| 2015.73| 7232.36| 1889.55| 3983.09| 4285.73| 7220.91| 1571.18| 4742.73 |
| Precise Parallel| 86.17  | 24.33  | 25.59  | 69.00  | 22.76  | 37.92  | 39.83  | 80.67  | 23.50  | 161.42  |
| Imprecise Parallel| 16.70 | 13.30  | 9.20   | 17.50  | 8.50   | 9.10   | 9.20   | 12.60  | 8.40   | 26.40   |

TABLE V
POWER AND ENERGY CONSUMPTION OF SQUEEZE NET USING SEQUENTIAL AND PARALLEL ALGORITHMS.

|                | Total Power (mW) | Differential Power (mW) | Energy/Image (J) | Energy Ratio |
|----------------|------------------|-------------------------|------------------|--------------|
|                | Baseline Sequential Imprecise Parallel | Sequential Imprecise Parallel | Sequential Imprecise Parallel |              |
| Galaxy S7      | 173.18           | 1552.51                 | 2921.79          | 1379.33      | 2748.61     | 17            | 0.569          | 29.88X       |
| Nexus 6P       | 1480.97          | 1999.12                 | 5461.89          | 518.15       | 3980.92     | 8.96          | 0.514          | 17.43X       |
| Nexus 5        | 422.71           | 1023.3                  | 1170.45          | 600.29       | 747.74      | 26.37         | 0.106          | 249.47X      |
B. Imprecise Computing

RenderScript offers two modes for imprecise computing for applications that do not need a strict implementation of IEEE 754. These modes are called relaxed and imprecise floating point computation. The relaxed floating point mode enables flush to zero for denormalized numbers and round toward zero. The imprecise computing mode enables everything in the relaxed computing. In addition, in this mode operations resulting in 0.0 can return +0.0 and operations on INF and NAN are undefined.

Using imprecise computing accelerates the execution time since some optimizations are only available for computation with relax/imprecise precisions. Notice that relaxed and imprecise are GPU modes and using these modes does not have any effect on the CPU side. We have implemented SqueezeNet with both relaxed and imprecise precisions. Subsequently, we have tested the implementation on the first 10000 samples of ILSVRC 2012 validation dataset [6]. For all these samples, the prediction results are identical to the original predictions of SqueezeNet. Therefore, using either of these modes does not change the classification accuracy.

Table (IV) includes the execution time of different convolutional layers of SqueezeNet. Execution time was measured on three different phones. For each platform, the first row shows execution times for sequential execution. Second row includes times for parallel execution, and the third row shows execution times for imprecise parallel implementation. The units of all execution times is in milliseconds. Table (VI) shows total execution time (all layers including pooling and softmax) of sequential and parallel implementation of SqueezeNet. In precise computing mode, processing each image takes between 388.36 (ms) on Nexus 6P to 588.29 (ms) on Nexus 5. As Table (VI) shows, the parallel algorithm is drastically faster than the sequential implementation of SqueezeNet on each platform. In precise processing mode, the proposed parallel algorithm is at least 28.24X (Galaxy S7) and at most 74.68X (Nexus 5) faster than the sequential equivalent.

As we explained, using imprecise computing does not have any impact on the accuracy of SqueezeNet. However, using imprecise computing decreases the execution time drastically by using SIMD optimization of GPUs. On Galaxy S7, the imprecise parallel algorithm is 2.11X faster than the precise parallel implementation. On Nexus 5 and Nexus 6P, imprecise parallel algorithms are 4.16X and 3X faster than precise parallel implementation, respectively. Using imprecise computing, the total execution time of SqueezeNet varies between 129.21 millisecond (Nexus 6P) to 207.1 millisecond (Galaxy S7). The speedup of imprecise implementation of proposed parallel algorithm is 59.54X (Galaxy S7), 133.89X (Nexus 6P), and 310.74X (Nexus 5) compared to the basic sequential implementation. Speedup factors which are shown in Table (VI) demonstrate that efficient use of mobile GPUs can decrease the execution time of neural networks drastically and make it possible to locally utilize CNNs on mobile devices.

C. Power Consumption

Trepn profiler is an application made by Qualcomm for performance monitoring of applications on mobile devices. Trepn especially performs well on mobile devices with Snapdragon processors. We have used Android Intents to call Trepn for automated performance monitoring of parallel and sequential implementation of SqueezeNet.

In all experiments, mobile devices have been in the airplane mode and screen brightness is set to minimum. Moreover, in all experiments background processes of different applications have been stopped. Table (V) shows the result of performance monitoring using Trepn profiler. Baseline indicates the power consumption of the system in the idle state. For example, for Nexus 6P the base power consumption is 1480.97 milliwatts. Differential power shows the required power for running an algorithm. On Galaxy S7 and Nexus 6P the required power for parallel algorithm is more than a sequential algorithm, since the parallel implementation requires more active cores simultaneously. However, in Nexus 5, the required power in parallel is less than sequential. There are two reasons for this low power consumption. First, maximum clock frequency of the GPU in Nexus 5 is 200 MHz less than the two other platforms. Second, Nexus 5 is equipped with an older Snapdragon chipset which has a lower performance. For a task such as running SqueezeNet the difference in performance is not considerable. However, in applications with intense process requirement such as games, the two other platforms outperform Nexus 5.

The main performance indicator is energy consumption. Sequential algorithms might have a better power consumption, but since the execution time is longer than parallel algorithms, the total energy consumption is much larger. In the third column of Table (V) total energy consumptions for both sequential and parallel algorithms are computed. Using the parallel algorithm, the required energy for processing a single image on different platforms varies between 0.106 to 0.569 joules.

V. Conclusion

In this paper we proposed a solution for accelerating CNNs on mobile GPUs. Using this approach we have accelerated SqueezeNet and ran it on three different mobile devices with different chipsets. Experimental results on different platforms show that the proposed algorithm achieves a speedup of at least 59.54X and at most 310.74X. In addition, the energy consumption of the proposed parallel algorithm is at least 29.88X and at most 249.47X less than the energy consumption of the sequential algorithm. Offering an execution time of less than a quarter of a second and around half a joule energy consumption, the proposed algorithm makes it feasible to use CNNs on mobile devices.

ACKNOWLEDGMENT

The authors would like to thank NVIDIA corporation for donating GPUs which were used in this research.

REFERENCES

[1] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer. SqueezeNet: Alexnet-level accuracy with 50x fewer parameters and¡ 1mb model size. arXiv preprint arXiv:1602.07360, 2016.
[2] 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 the 22nd ACM international conference on Multimedia, pages 675–678. ACM, 2014.
[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems, pages 1097–1105, 2012.
[4] M. Motamedi, P. Gysel, V. Akella, and S. Ghiassi. Design space exploration of fpga-based deep convolutional neural networks. In 2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC), pages 575–580, Jan 2016.
[5] S. S. L. Oskouei, H. Golestanli, M. Kachuee, M. Hashemi, H. Mohammadzade, and S. Ghiassi. Gpu-based acceleration of deep convolutional neural networks on mobile platforms. arXiv preprint arXiv:1511.07376, 2015.
[6] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. International Journal of Computer Vision (IJCV), 115(3):211–252, 2015.

---

**TABLE VI**

| Platform   | Sequential | Precise Parallel | Speedup | Imprecise Parallel | Speedup |
|------------|------------|------------------|---------|-------------------|---------|
| Galaxy S7  | 12331.82   | 436.71           | 28.24X  | 207.1             | 59.54X  |
| Nexus 6P   | 17299.55   | 388.36           | 44.55X  | 129.21            | 133.89X |
| Nexus 5    | 43932.73   | 588.29           | 74.68X  | 141.38            | 310.74X |

**TOTAL EXECUTION TIME (MILLISECOND) OF SQUEEZENET ON DIFFERENT PLATFORMS.**
[7] 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 the IEEE Conference on Computer Vision and Pattern Recognition, pages 1–9, 2015.

[8] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pages 161–170. ACM, 2015.