# Machine Intelligence on Resource-Constrained IoT Devices: The Case of Thread Granularity Optimization for CNN Inference



MOHAMMAD MOTAMEDI, DANIEL FONG, and SOHEIL GHIASI,
University of California, Davis

Despite their remarkable performance in various machine intelligence tasks, the computational intensity of Convolutional Neural Networks (CNNs) has hindered their widespread utilization in resource-constrained embedded and IoT systems. To address this problem, we present a framework for synthesis of efficient CNN inference software targeting mobile SoC platforms. We argue that thread granularity can substantially impact the performance and energy dissipation of the synthesized inference software, and demonstrate that launching the maximum number of logical threads, often promoted as a guiding principle by GPGPU practitioners, does not result in an efficient implementation for mobile SoCs. We hypothesize that the runtime of a CNN layer on a particular SoC platform can be accurately estimated as a linear function of its computational complexity, which may seem counter-intuitive, as modern mobile SoCs utilize a plethora of heterogeneous architectural features and dynamic resource management policies. Consequently, we develop a principled approach and a data-driven analytical model to optimize granularity of threads during CNN software synthesis. Experimental results with several modern CNNs mapped to a commodity Android smartphone with a Snapdragon SoC show up to 2.37X speedup in application runtime, and up to 1.9X improvement in its energy dissipation compared to existing approaches.

CCS Concepts: • **Computing methodologies** → *Massively parallel algorithms*; • **Computer systems organization** → **Embedded systems**;

Additional Key Words and Phrases: Convolutional neural networks, mobile GPUs, thread coarsening, thread granularity

**ACM Reference format:**
Mohammad Motamedi, Daniel Fong, and Soheil Ghiasi. 2017. Machine Intelligence on Resource-Constrained IoT Devices: The Case of Thread Granularity Optimization for CNN Inference. *ACM Trans. Embed. Comput. Syst.* 16, 5s, Article 151 (September 2017), 19 pages.
https://doi.org/10.1145/3126555

## 1 INTRODUCTION

Convolutional Neural Networks (CNNs) have remarkable performance in various machine intelligence tasks [6, 8, 21]. Despite CNNs' performance in classification, they are computationally intensive. Due to this intensity, it is required to use parallelization approaches for deploying a

This article was presented in the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) 2017 and appears as part of the ESWEEK-TECS special issue.
Authors' addresses: M. Motamedi (corresponding author), D. Fong, and S. Ghiasi, Electrical and Computer Engineering Department, University of California, Davis, One Shields Avenue, Davis, CA 95616; emails: {mmotamedi, dfong, ghiasi}@ucdavis.edu.

https://doi.org/10.1145/3126555

CNN on an embedded platform. Most mobile devices are equipped with a modern System on a Chip (SoC) which offers parallelism in various forms. In order to ship a trained CNN to a mobile device, it is necessary to judiciously use all of the resources that a SoC offers to achieve the highest performance in terms of execution time and energy consumption. The research community uses tools such as Caffe [10] or Torch [4] for training a CNN to perform inference. However, deploying a trained CNN on an embedded platform is the matter of realizing the inference functionality only (i.e., the forward path). Hence, the training submodules are disregarded on these platforms.

In one of our previous works [18], we offered Cappuccino: A platform for efficient deployment of CNNs on mobile devices. Cappuccino receives a trained CNN model as an input, and synthesizes a parallel RenderScript program to be executed on a mobile platform. Cappuccino takes advantage of all the available computation hardware (CPU cores, GPU, and DSP) to accelerate the CNN.

In deploying a trained CNN on a SoC, finding the optimal number of logical threads to launch for each task is a challenging problem and plays an important role in accelerating CNNs on mobile SoCs. In most parallelization problems there is a tendency to select a higher number of logical threads. Such a choice seems reasonable since increasing the parallelism level is expected to decrease the execution time. However, a higher number of threads increases scheduling overhead and decreases data reusability. Finding a balance to achieve the highest acceleration should be investigated. In this paper, we address this question: *What is the optimal number of threads to launch per convolutional layer to achieve the fastest execution of a CNN?*

Our contributions in this paper are:

(1) We show that unlike common GPGPU recommendations, a CNN-implementation using a mobile SoC that launches the maximal number of logical threads (finest thread granularity) does not result in the highest acceleration.
(2) In coarser thread granularities, multiple tasks are assigned to a single thread. We offer a memory-aware task assignment policy which decreases the required memory bandwidth by taking advantage of inter-thread data locality.
(3) We offer a data-driven analytical model for identifying the best thread granularity for each layer of a CNN. Furthermore, we update Cappuccino with the proposed model and synthesize new parallel programs for state-of-the-art CNNs and measure their execution times. Subsequently, we compare the measured execution times with those of maximally parallel algorithms and study the impact of thread granularity on the acceleration and energy consumption.

A maximally parallel algorithm refers to a parallelization methodology that disregards the effect of thread granularity and launches the maximum possible number of threads for each task. Such an algorithm is also called parallelization with the finest thread granularity.

## 2 RELATED WORK

The research community has put forth a number of ideas for accelerating CNNs on embedded platforms. These efforts can be classified into three major categories.

*2.1.1 ASIC-based Acceleration.* The main advantage of designing an ASIC chip is to reduce overall power consumption. Chen et al. designed an ASIC chip for accelerating CNNs [3], which was able to achieve a speedup of 450X compared to a high-end GPU. In a similar work, Google has developed Google TPU to accelerate machine learning processes that use TensorFlow [11].

*2.1.2 FPGA-based Acceleration.* Researchers have proposed different approaches for FPGA-based acceleration of CNNs. Chakradhar et al. offered an accelerator that uses dynamic reconfigurability to increase the throughput [2]. Zhang et al. took a similar approach to decrease the required

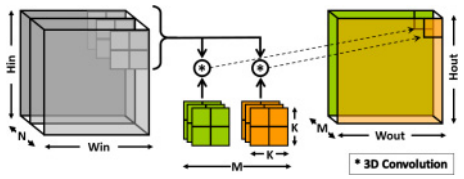


Fig. 1. A convolutional layer of a CNN. Two kernels (green and orange) get convolved with IFMs to generate two OFMs.

memory bandwidth by using the on-chip memory efficiently [22]. Motamedi et al. further improved this approach by exploiting kernel level parallelism [19]. There are many proposals of different accelerator architectures for accelerating CNNs on FPGAs. Reviewing all the research in this area is out of the scope of this paper.

*2.1.3 Mobile SoC-based Acceleration.* The last approach for using CNNs on embedded platforms is SoC-based acceleration of CNNs. Mobile SoCs are flexible and support parallelism at different levels. Moreover, this platform has been in commodity mobile devices and are extensively-used by many users daily. Oskouei et al. offered a library for parallel-execution of CNNs on Android devices [14]. We further improved that work by offering inexact computing and analyzed its impact on classification accuracy. Moreover, we offered a new approach for utilizing sub-word parallelism with zero reordering overhead [17]. Subsequently, we created Cappuccino: A platform for efficient deployment of CNNs on mobile devices [18]. In this work, we focus on the impact of thread granularity on the execution time and energy consumption. We develop a data-driven model to optimize granularity of threads for parallel execution of CNNs. Subsequently, we show that an optimal choice of thread granularity can accelerate a parallel algorithm by up to 2.37X and decrease its energy consumption by 1.9X.

The problem of finding the optimal number of threads to launch on a platform is called thread granularity or thread coarsening. This problem has been studied for server-grade GPUs. Magni et al. performed a study on the impact of thread granularity on acceleration for four GPUs [16]. They continued this work and offered an approach for optimizing thread granularity on the same GPUs [15].

The problem of selecting the best thread granularity for mobile SoCs is complicated because the target platform has a heterogeneous architecture. This allows threads to be mapped to the available CPU cores, GPUs, and DSPs. In addition, most SoCs share the same memory among all processors which leads to a more non-deterministic memory access time. The problem of selecting an optimal thread granularity for a mobile SoC has yet to be addressed.

## 3 INFERENCE USING CNNS

CNNs include millions of parameters which are obtained during the training phase. These parameters maintain the knowledge of a CNN about the classification task for which it has been trained. Using these parameters, a CNN extracts features from an input and uses those features to determine to what class the input image belongs. The classification procedure queries a CNN's forward path, which is a time-critical application. *In this research we concentrate on efficient deployments of the forward path on mobile SoCs.*

A CNN includes some convolutional layers. Each convolutional layer has three main components: Input Feature Maps (IFMs), Output Feature Maps (OFMs), and Kernels. We use $N$ to show the number of IFMs and $M$ to show the number of OFMs. Figure 1 shows one convolutional layer. The height and width of IFMs are shown by $H_{in}$ and $W_{in}$, respectively. Likewise, $H_{out}$ and $W_{out}$ show width and height of OFMs, respectively. Each convolutional layer includes $M$ kernels whose

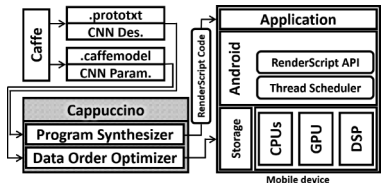


Fig. 2. Cappuccino needs the CNN description and the CNN parameter files. It uses the former to synthesize a RenderScript program and the latter to improve the efficiency of memory system.

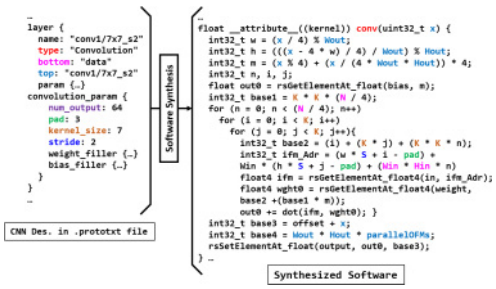


Fig. 3. Cappuccino uses a CNN description file (left side) to synthesize a RenderScript program (right side). Color coding is used to show relationships between parameters in the description file and generated program.

height and width is shown by $K$. The number of kernels is equal to the number of OFMs because each OFM is the result of a 3D convolution of one kernel with IFMs.

Each convolution kernel has to be slid over input feature maps with the stride of $S$. At every location, we perform a 3D convolution between the kernel and corresponding pixels from IFMs. the result of this convolution is a pixel in the OFM. In Figure 1, two convolution kernels get convolved with IFMs. The results are two OFMs. The color code is used to depict that each convolution kernel creates a separate OFM.

## 4 OVERVIEW OF CAPPUCCINO

While Caffe [10] is one of the dominant platforms for training a CNN, it offers little support for shipping a trained model to a resource-constrained embedded system. To address this, we developed Cappuccino [18] which takes a trained Caffe-model and automatically synthesizes a highly-parallel implementation of it for Android devices. In this section, we briefly overview Cappuccino.

To start the code synthesis process, Cappuccino needs a Caffe model file (.caffemodel) and a CNN description file (.prototxt), which contains the CNN's parameters (weights and biases) and describes the network architecture respectively. Cappuccino then gives this information to the two main submodules: the Program Synthesizer and the Data Order Optimizer. A high-level diagram of the system can be seen in Figure 2.

The Program Synthesizer reads the CNN descriptions and synthesizes a RenderScript-based, parallel program. It also optimizes the generated programs parameters to minimize the execution time. The process that Cappuccino uses to synthesize the output code is briefly summarized in Section 4.1. Figure 3 shows a sample CNN description file and the generated parallel program for it.

The second submodule of Cappuccino is the Data Order Optimizer, which is responsible for changing the order of inputs and weights from column/row major to depth major to maximize memory bandwidth utilization. It is worth mentioning that the reordering process does not increase the data size and is an offline process. Reordered parameters will be written in a

mobile device's storage (internal storage or SD card). Subsequently, the synthesized RenderScript program can be launched on the platform as an application.

Cappuccino generates a RenderScript program which fully utilizes the RenderScript APIs to maximally accelerate a given CNN. However, RenderScript runtime controls the thread scheduling process. Hence, different applications, including those generated by Cappuccino, must rely on the RenderScript thread scheduler. The RenderScript runtime utilizes all processors which are available on a device, such as CPU cores, GPU, and DSP to run a parallel program.

### 4.1 Acceleration Strategy

Cappuccino takes two general approaches for accelerating CNNs: inexact computing and parallel processing.

*4.1.1 Inexact Computing.* CNNs are robust against errors that are introduced with inexact computing. Gysel et al. has shown that implementing CNNs with dynamic fixed point, 16-bits, and 8-bits fixed point arithmetic has minimal impact on the classification accuracy [5]. On most embedded platforms, however, it is not possible to implement a custom accelerator. Therefore, on such platforms, inexact computing takes the form of imprecise computing modes that a system supports.

Android devices support two inexact computing modes which can potentially expedite the computation at the cost of decreasing the accuracy. The first inexact computing mode, which is called relaxed computing, optimizes the efficiency by inaccurate handling of de-normalized numbers. The second mode, which is called imprecise computing, further improves the efficiency by the inaccurate process of INF, NaN, and $-0.0/+0.0$. Utilizing imprecise/relaxed computing allows efficient use of hardware by enabling SIMD instructions on CPUs. Hence, the program that is synthesized by Cappuccino can be mapped to a higher number of threads which yields higher acceleration. However, before using imprecise/relaxed computing, it is necessary to consider their impact on the classification accuracy.

Cappuccino takes a layer-by-layer approach for this assessment. It changes the computation mode for one CNN layer at a time to evaluate its effect on the execution time and the classification accuracy. For example, Cappuccino can evaluate the impact of using inexact computing in a layer on the classification performance of the entire network. Therefore, it can determine which computation mode, out of the three imprecise, relaxed, or precise choices, best fits a particular layer of a given CNN model.

*4.1.2 Parallel Processing.* CNN architectures lend themselves well to parallel-processing algorithms. To explain, consider that all OFMs in a layer can be computed in parallel. Furthermore, computing different pixels in an OFM is a perfectly parallel workload, and each OFM pixel is simply the dot product of the IFM pixels and the weights, an operation that can be performed in parallel. Cappuccino uses all of these sources of parallelism to generate a RenderScript program which fully utilizes the available resources on a target SoC. This process is briefly mentioned here:

**Thread workload assignment:** For each layer, Cappuccino assigns one thread per each pixel of OFMs. Therefore, the number of logical threads for each layer will be equal to the sum of all pixels of all OFMs in that layer. The current thread assignment policy uses the finest thread granularity (i.e., it maximizes the number of logical threads and minimizes the amount of workload per thread). As we will discuss in this paper, this is not the most efficient task assignment policy.

**Sub-word Parallelism:** Cappuccino uses multithreading to exploit the Thread Level Parallelism (TLP) that a platform offers. In this context, TLP includes CPU, GPU, and DSP threads. Single Instruction, Multiple Data (SIMD) is offered in the form of sub-word parallelism on mobile devices,

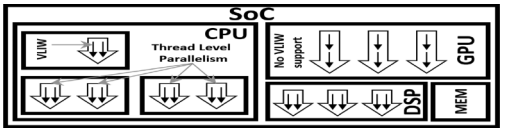


Fig. 4. Whenever possible we use sub-word parallelism inside each thread to parallelize its workload internally.

and can be used to further improve the execution time. Using SIMD, Cappuccino parallelizes the task of each thread internally as it is illustrated in Figure 4.

Each thread's task is conceptually a dot product computation. This task is accelerated using SIMD instructions to minimize inter-thread data transfer, thread synchronization, and to maximize the cache efficiency. To fully-utilize SIMD instructions, Cappuccino changes the data order to maximize the memory bandwidth utilization and optimizes the loop order to improve cache performance. However, the detailed information of the algorithms that Cappuccino uses for parallelizing CNNs on Android devices is out of the scope of this paper.

## 5 PROBLEM STATEMENT

In accelerating an algorithm on SoCs, finding the optimal number of logical threads to launch is a challenging problem. Having a finer thread granularity increases the utilization of available parallel resources on a SoC. However, this imposes extra scheduling overhead and reduces data reusability. As we will show in Section 7, choosing an optimal thread granularity improves the execution time of CNNs drastically. In this paper, we consider the following question: *What is the optimal number of threads to launch per convolutional layer to achieve the fastest execution of a CNN on a given platform?*

To answer this question, we developed a data-driven analytical model to predict the execution time for different thread granularities. We use this model to determine the best granularity to use, and update the thread assignment policy of Cappuccino. We will show that the proposed solution yields near ideal results for different CNNs.

## 6 THREAD GRANULARITY OPTIMIZATION

Thread granularity indicates how many threads are invoked for running a parallel workload. In the case of using the finest thread granularity, the number of threads are limited by the number of independent subtasks that exist in a workload. Coarsest thread granularity is the case in which only one thread is used (i.e., a sequential program). Both finest and coarsest thread granularities are two extreme cases, and there is a trade-off between the number of threads used and computational load per each of them. In this section, we show that these extreme cases do not yield optimal acceleration rates. Subsequently, we offer a performance prediction model that can be used to estimate what thread granularity results in the best acceleration for each layer of a given CNN.

### 6.1 Thread Granularity

For a convolutional layer in a CNN, assume $H_{out}$, $W_{out}$, and $M$ show height, width, and the number of Output Feature Maps (OFMs), respectively. In the naive approach of using the finest thread granularity, Equation (1) gives the number of logical threads required for parallelizing this layer. We use parameter $\alpha$ to refer to this number in this paper.

$$\alpha = M \times W_{out} \times H_{out} \tag{1}$$

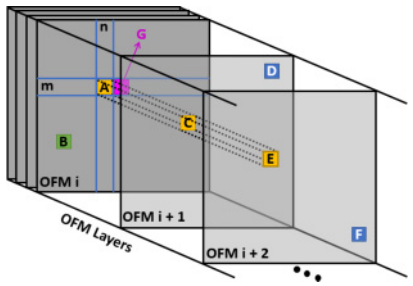


Fig. 5. Task assignment choices for thread granularities with ($g > 1$).

To have a coarser granularity, each thread should compute $g$ pixels instead of one. A higher value of $g$, gives a coarser thread granularity. For this case, the number of logical threads, $\beta$, can be found using Equation (2).

$$\beta = \alpha / g \tag{2}$$

Each 3D convolution consist of $N$ different 2D convolutions where $N$ is the number of IFMs (i.e., to compute a 3D convolution it is required to perform a series of 2D convolutions). Therefore, in the rest of this section we focus on efficient workload assignment for a 2D convolution.

*6.1.1 Memory Access Optimization.* In thread granularities with $g > 1$, it is required to select $(g - 1)$ extra pixels for each thread to compute. To illustrate this problem, consider an OFM shown in Figure 5. When $g$ equals one (finest granularity), thread $(m, n, i)$ is responsible for computing the value of pixel A in the OFM. When $g$ is greater than one, the thread granularity is coarsened and this thread becomes responsible for computing the value of additional pixels. However, this selection must be carefully considered due to potential trade-offs. Here we offer an efficient workload assignment mechanism, and consider the following three policies to guide this selection:

**Workload Assignment Policy 1:** The first policy is selecting a pixel from another location of the same OFM (e.g., pixel G or B in Figure 5). Pixels in the same OFM share the same kernel (i.e., their value is the result of convolving the same kernel with different locations of IFMs). Hence, when a thread loads parameters of that kernel from the memory, the parameter can be used $g$ times. Therefore, assuming the kernel dimension is denoted by $K$, for computing an output element, it is required to load $K^2$ pixels from IFMs, load the kernel ($K^2$ elements) and write the result back: $2K^2 + 1$ memory accesses. However, for a coarser thread granularity ($g > 1$), kernel gets loaded once and will be used $g$ times. In this case, the number of memory accesses to compute one output element, $\gamma$, can be found using Equation (3).

$$\gamma = K^2/g + K^2 + 1 \tag{3}$$

In the special case when the selected pixel is adjacent to the original pixel (e.g., pixel G), memory accesses can be optimized further. The values of pixels G and A are computed from overlapping regions of IFM pixels. Hence, part of the IFM pixels are shared and can be loaded once and be used $g$ times. The effectiveness of such an optimization depends on the values of $g$ and stride ($S$). For example, when $K \leq S$ or $K = 1$, such an optimization is ineffective. However, for most cases the required number of memory accesses in this selection policy is less than or equal to $\gamma$ (i.e., $\gamma_{Policy1} \leq \gamma$).

**Workload Assignment Policy 2:** In this policy, the selected pixel belongs to the same location of another OFM (e.g., pixel C or E in Figure 5). These pixels are the result of convolution of the same IFM with different kernels. Hence, if we use this workload assignment policy for a coarser

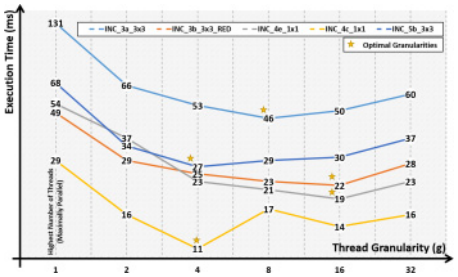


Fig. 6. Execution time of five different layers of GoogLeNet for different thread granularities on Google Nexus 5. Finest thread granularity (maximally parallel) yields sub-optimal execution time.

thread granularity ($g > 1$), IFMs will be loaded from the memory once, but will be reused $g$ times. Computing the number of memory accesses in this case is very similar to the first policy. In order to compute a pixel in an OFM, it is required to load $K^2$ pixels from IFMs and $K^2$ pixels from kernels. For coarse granularity, these pixels are used for computing all $g$ output pixels. Hence, Equation (3) yields the number of memory accesses. Notice that with this policy the kernels no longer overlap, there is no further memory reuse optimization ($\gamma_{Policy2} = \gamma$).

**Workload Assignment Policy 3:** In this policy, which is the naivest approach, there is no constraint on location or OFM from which we select an extra pixel (e.g., pixel D or F in Figure 5). Such a selection offers no memory optimization since these pixels share neither IFMs nor kernel values. In this case, the number of required memory accesses is $2K^2 + 1$. The only advantage of this policy is ease of implementation.

The first policy allows the highest memory optimization. However, implementation is sufficiently difficult and the generated program is hard to read and maintain. We use this policy when it offers an optimization over Policy 2 (i.e., when $\gamma_{P1} < \gamma$). Otherwise, we use the second policy.

*6.1.2 Case Study: Effect of Different Thread Granularities on GoogLeNet.* We implemented the aforementioned policies, in Cappuccino to create a range of thread granularities from fine to coarse with the values shown in Equation (4). We used this to synthesize a RenderScript-based implementation of GoogLeNet [21] and measured the execution time. GoogLeNet is a very deep CNN which was designed by Google. This CNN won in the 2014 ImageNet large-scale visual recognition challenge (ILSVRC) [20]. In the next subsection, we use some layers of GoogLeNet to highlight: 1) The finest thread granularity has a sub-optimal performance. 2) There is no single thread granularity that yields the optimal performance for different CNNs or even for different layers of the same CNN.

$$g \in \{1, 2, 4, 8, 16, 32\} \tag{4}$$

Figure 6 shows the execution time of five different layers of GoogLeNet for various thread granularities and Table 1 shows detailed characteristics of these layers.[1]

We performed similar experiments on different devices and different CNNs. In all of them, we observed analogous patterns, which are summarized below.[2]

[1]Execution times are measured on a Google Nexus 5 phone. Each experiment has been repeated 100 times and the average execution time is computed.

[2]Additional experiments have been performed on Google Nexus 6P and Samsung Galaxy S7. Generic patterns are discussed here which are observed on all phones for different CNNs (GoogLeNet, SqueezeNet [8], and AlexNet [13]).

Table 1. Specifications of Five Different Layers of GoogLeNet and the Optimal Thread Granularity for Each of Them for Execution on Google Nexus 5

| Layer Name | N | M | $W_{in}$ | $H_{in}$ | K | #MAC Ops | Best Granularity ($g$) |
|---|---|---|---|---|---|---|---|
| Inception_3a_3x3 | 96 | 128 | 28 | 28 | 3 | 28,901,376 | 8 |
| Inception_3b_3x3_reduce | 256 | 128 | 28 | 28 | 1 | 25,690,112 | 16 |
| inception_4c_1x1 | 512 | 128 | 14 | 14 | 1 | 12,845,056 | 4 |
| Inception_4e_1x1 | 528 | 256 | 14 | 14 | 1 | 26,492,928 | 16 |
| Inception_5b_3x3 | 192 | 384 | 7 | 7 | 3 | 10,838,016 | 4 |

The variables are as follows: N (number of IFMs), M (number of OFMs), $W_{in}$ (IFM width), $H_{in}$ (IFM height), and K (kernel dimension).

**Lessons Learned on SoC-Based Acceleration of CNNs:**

(1) Using the finest thread granularity ($g = 1$), is not always the best solution. In parallelizing a task, generating the highest number of logical threads seems favorable because it increases the parallelism. While this normally yields the best results for accelerating an algorithm on server-grade GPUs, this is not the case for embedded platforms. Since mobile SoCs are designed to operate within a restricted power budget, they have limited computing power (small number of cores). As a result, defining a high number of logical threads does not necessarily decrease the execution time because those threads will be executed as multiple batches in sequence. In other words, the limited resources forces a sequential execution; Hence, increasing the number of logical threads only imposes further scheduling overhead. In addition, the finest thread granularity prevents utilization of the memory optimization approaches that we introduced in Section 6.1. Therefore, it is predictable to have poor performance in fine thread granularities. In our experiments, we observed that the finest thread granularity always performed sub-optimally.
(2) There is no single thread granularity capable of yielding the minimal execution time for all layers in a CNN. A thread granularity that performs well for one task might have inferior performance for another. For example, consider the thread granularity with the value of ($g = 8$) for layers INC_3a_3x3 and INC_4c_1x1 shown in Figure 6.
(3) Different classes of SoCs result in different optimal thread granularities when presented with similar workloads. This is explained by the varying compute capability between different SoCs (number of cores and on-chip resources). We noticed that for the same task, high-end SoCs tended to perform better with finer granularities, and low-end SoCs performed better with coarser granularities.

## 6.2 Analysis of Characteristics of Convolution

In this subsection, we explore the characteristics of convolution as a workload. We try to find a relationship between the optimal thread granularity and the computational complexity of a convolutional layer.

*6.2.1 Computational Complexity.* To compute OFMs of a layer (dimensions: $M \times W_{out} \times H_{out}$), it is required to slide convolution kernels (M kernels, dimensions of each: $N \times K \times K$) over IFMs (dimensions: $N \times W_{in} \times W_{out}$) with the stride of $S$ and compute a 3D convolution at every location. The number of MAC (Multiply-accumulate) operations required for computing OFMs is shown in Equation (5). We use the term Computational Complexity to refer to this number.

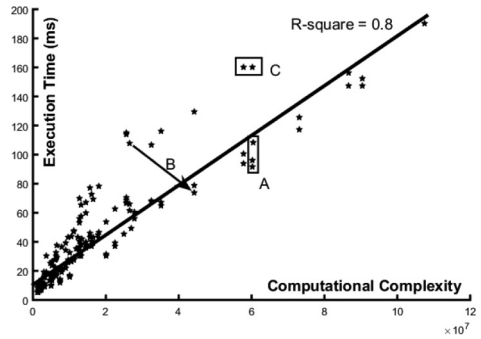


Fig. 7. Execution time of different convolutional layers based on their computational complexity. It is expected to witness a linear relationship between computational complexity and execution time. However, DVFS imposes a lot of irregular patterns.

$$Computational\ Complexity = \frac{N \times W_{in} \times H_{in} \times K^2 \times M}{S^2} \tag{5}$$

For comparison purposes, computational complexity is an acceptable metric for estimating the expected execution time of different algorithms. We use this metric to compare execution times of different convolutional layers for the same algorithm. On this ground, it is reasonable to expect a linear relationship between the execution time and the computational complexities of different convolutional layers. Figure 7 shows the execution time of different convolutional kernels (Units: ms) versus their computational complexity (Units: Number of MAC operations). Even though this data has a semi-linear pattern, it includes a large number of irregularities: 1) Convolutional layers with the same computational complexity have different execution time (an example is shown by region A in Figure 7). 2) Some layers with the same execution time have different computational complexity (an example is shown by C in Figure 7). 3) Some layers with a smaller computational complexity have a larger execution time compared to another layer with a larger computational complexity (an example is shown by B in Figure 7).
We hypothesize that such irregularities are caused by:

1) **Resource sharing and multitasking:** The platform we used for performing the experiment is a multitask platform which host many processes. Even though we killed all of the redundant tasks before every experiment, processes owned by the system remained active. Due to the nondeterministic nature of such tasks, they impose some level of noise to measurement.
2) **Dynamic Voltage and Frequency Scaling (DVFS):** DVFS is a technique which is used for adjusting the voltage and frequency of a processor based on the present computational load and the chip temperature. Load is the aggregated system demand, which can be from different processes.

These two factors explain the irregularities seen in Figure 7. We have already minimized the effect of the first factor. Hence, DVFS is responsible for most of the irregularities that we observed. In what follows, we disable the operating system's control over DVFS and perform the same experiments. Controlling DVFS in the super user mode increases the temporal determinacy of the platform and helps us in interpreting the results.

*6.2.2 Beyond Restrictions of the Operating System.* Similar to other operating systems, Android restricts user-access to some system-level settings for security and stability reasons. Therefore, in

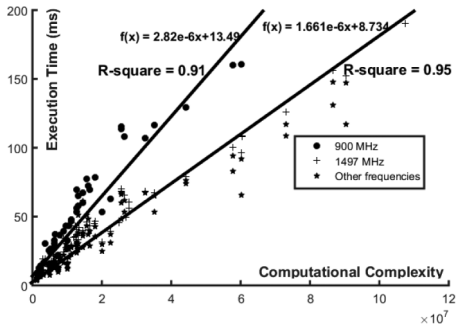


Fig. 8. Frequency based clustering of experiments. In each cluster, there is a linear relationship between the computational complexity and execution time.

order to gain control of the entire system, it is required to attain privileged access (also known as root access or superuser access). The process of modifying the default settings for obtaining the privileged access is called rooting. Controlling the power governor and DVFS is among those settings that demand superuser privilege. Hence, we rooted our device to attain control over them.[3] The platform that we use, has ten potential working frequency ranging from 300 MHz to 2.2 GHz. We froze the frequency of the platform on all ten possible values and repeated all experiments. Subsequently, we compared the achieved execution times with those shown in Figure 7 and computed the difference. In comparison, when the difference between a newly measured execution time under the frequency of $f$ with a value of Figure 7 was less then 1%, we concluded that the old value is measured when the processor was working with the frequency of $f$. The results of this comparison for $f = 900MHz$ and $f = 1497MHz$ are shown in Figure 8. As this figure illustrates, dividing the execution time to different clusters based on the frequency makes it easier to interpret the runtime measurements. In Figure 8, one cluster is shown by black circles and another by black plus signs (+). The data of each cluster is less chaotic and has a linear pattern. As the computational complexity directly impacts the execution time, it is not surprising to observe a linear relationship between these two.

We used curve fitting to determine how successful a linear fit is in explaining the variation of data in each cluster. Equations (6) and (7) show these lines. The value of $R^2$ for the first one is 0.91 and for the second one 0.95. $R^2$ is the square of the correlation between the response values and the predicted response value. This statistical parameter indicates how successfully a fit represents the data. The value of $R^2$ varies between zero to one. Values close to 1 indicate that a greater proportion of variance is accounted by the model. The value of $R^2$ in Figure 7 where the line is fitted to all data rather than small clusters is 0.8.

$$f_{900MHz}(x) = 2.82 \times 10^{-6} \times x + 13.49 \tag{6}$$

$$f_{1497MHz}(x) = 1.661 \times 10^{-6} \times x + 8.734 \tag{7}$$

Clustering the execution times based on the frequency of the processor at the time of measurement makes it possible to fit a more representative model on each cluster. We use such models to predict the execution time of a convolution layer on a given processor with a given working frequency. Subsequently, we extend this model to predict the execution time for different thread granularities.

[3]In 2010, the U.S. Copyright office permitted rooting by explicitly exempting it from Digital Millennium Copyright Act [1]. Notice that rooting is different from unlocking. All mobile devices that are used in this research were unlocked.

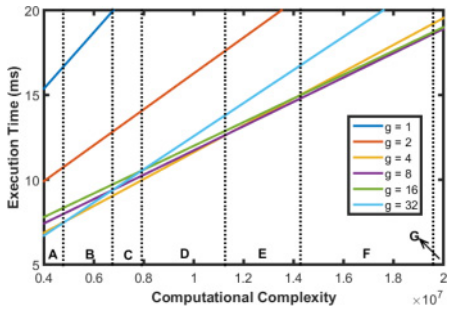


Fig. 9. Linear Regression Result for different thread granularities ($g \in \{1, 2, 4, 8, 16, 32\}$). Optimal thread granularity varies based on the value of computational complexity.

### 6.3 Analytical Model for Granularity Selection

Using a learning based system, it is possible to predict the execution time of a given convolutional layer for different thread granularities. The result of such a prediction can be used to determine which thread granularity performs the best for a given convolutional layer on a certain SoC.

*6.3.1 Execution Time Prediction Using Linear Regression.* In the previous sections we explained that it is possible to use a linear model for predicting the execution time if two conditions are satisfied: 1) The processor's frequency is fixed. 2) The performance prediction model is trained for a certain thread granularity (i.e., granularity is fixed). We ran six version of GoogLeNet with the thread granularities shown in Equation (4) and measured the execution time. Subsequently, we used the results to train six different models $f_{(freq,g)}(x)$ where in all six models the value of $freq$ equals to 1497MHz. These models are shown in Equations (8) to (13) and illustrated in Figure 9.

$$f_{(1497MHz,g=1)}(x) = 1.661 \times 10^{-6} \times x + 8.734 \tag{8}$$

$$f_{(1497MHz,g=2)}(x) = 1.059 \times 10^{-6} \times x + 5.682 \tag{9}$$

$$f_{(1497MHz,g=4)}(x) = 7.915 \times 10^{-7} \times x + 3.707 \tag{10}$$

$$f_{(1497MHz,g=8)}(x) = 7.171 \times 10^{-7} \times x + 4.557 \tag{11}$$

$$f_{(1497MHz,g=16)}(x) = 7.001 \times 10^{-7} \times x + 4.997 \tag{12}$$

$$f_{(1497MHz,g=32)}(x) = 9.775 \times 10^{-7} \times x + 2.7961 \tag{13}$$

Some thread granularities always yield an inferior performance. For example, it is easy to determine that $g = 1$ or $g = 2$ never have the best execution times. This observation for $g = 1$ is more important for two main reasons. First, unlike the default expectation, when the algorithm is maximally parallelized (finest thread granularity) it does not have the minimum execution time. Second, the optimal thread granularity performs up to 4x faster in some cases. This indicates that the impact of finding the optimal thread granularity on execution time is considerable. As Figure 9 illustrates, based on the value of computational complexity, it is possible to define seven regions where different thread granularities perform faster. For example, in the first domain (labeled A in Figure 9) the order of thread granularity performance, ranked from slowest to fastest, is $(1, 2, 16, 8, 4, 32)$. Therefore, for a convolutional layer whose computational complexity is in this region, one should use a thread granularity of $g = 32$ to achieve the fastest execution. Likewise, in the domain labeled D the order of performance is $(1, 2, 32, 16, 8, 4)$ and the highest acceleration would be with using

$g = 4$. Our model allows a user to select the best thread granularity for a convolutional layer by simply computing its computational complexity.

### 6.4 Model Constraints

The offered solutions has two main constrains: First, it is only valid for a certain frequency (in this case 1497MHz). Second, the model is platform dependent. In this subsection we address these constraints.

*6.4.1 Frequency Scaling.* We adopt a linear performance scaling model, with respect to DVFS. Such a model may not be perfectly accurate for performance prediction due to a number of reasons such as components on the SoC that do not support DVFS. However, our intuition is that it will have high fidelity [7, 9] for comparing different thread granularities, and thus, would serve the purpose of thread granularity optimization. Specifically, we use Equation (14) to estimate the execution time under granularity $g$ and frequency $f2$, where $x$ is the computational complexity. In Section 7, we evaluate the model for different frequencies.

$$f_{(f_2,g)}(x) = \frac{f_{(f_1,g)}(x) \times f_1}{f_2} \tag{14}$$

*6.4.2 Model Portability.* The model we obtained is only valid for the platform on which we created the training examples (i.e., execution times for different granularities). In other words, a trained model has knowledge about the characteristics of a particular SoC and use it to estimate how different thread granularities perform for a given workload. Since different SoCs can vary in many aspects, it is not possible to use a trained-model from one SoC on a different SoC. While this data is not portable, the training of the model can be repeated. Based on the result of this research, we modified Cappuccino to create a learning submodule in each project. In the first run, this learning toolkit will be used to run sample convolutional kernels in order to determine the computational capabilities of a SoC and finding the SoC specific regression parameters. These parameters can then be used to find the best thread granularity for each layer of any CNN. In Section 7, this approach is used for finding best thread granularities for different CNNs on a new platform. The results are shown in rows #1 to #3 of Table 3.

## 7 EXPERIMENTAL RESULTS

In this section we evaluate the effect of thread granularity on the acceleration and its energy consumption. We use our experimental results to demonstrate two points: First, we show that thread granularity plays a major role in the execution time of a parallel algorithm on a SoC. Our experimental results show that selecting an optimal thread granularity can speedup a maximally parallelized algorithm by up to 2.37X. In a maximally parallelized algorithm, the number of defined threads is equal to the number of output elements ($g = 1$). Second, we use the experimental results to validate that the offered model predicts the correct granularity precisely. We test the model for different CNNs under various frequency conditions and show that the proposed model achieves near-ideal results.

### 7.1 Platform Specification

Experiments have been performed on a Google Nexus 5 phone and a Google Nexus 6P phone. Nexus 5 is equipped with a Snapdragon 800 SoC and Nexus 6P has a Snapdragon 810 SoC. Snapdragon 800 has a Krait 400 (ARMv7) processor which has four cores. Each of these cores have a 128-bit ALU which allows them to support Very Large Instruction Words (VLIW). The SoC also

has an Adreno 330 GPU with unified shader which has 128 ALUs. The last co-processor of this SoC is a Hexagon (QDSP6) DSP which supports both Dynamic Multi Threading and VLIW.

We rooted the phone to control the SoCs DVFS. Some experiments were performed with a fixed frequency by turning off DVFS, while others used the default mode with active DVFS in which the performance governor decides the operating frequency. In all cases, we put the phone on airplane mode and killed all unnecessary processes to minimize the disturbance to our experiments from other tasks.

### 7.2 Data Acquisition

*7.2.1 Training Data.* As we explained in the previous section, we trained our granularity selection model by measuring the execution time for each convolutional layer of GoogLeNet using a fixed frequency of $f = 1497MHz$. The regression parameters for this frequency were then scaled to find the regression parameters for other frequencies.

*7.2.2 Test.* To evaluate the effectiveness of our achieved model, we used it with three CNNs, namely AlexNet, SqueezeNet, and GoogLeNet, over several working frequencies. AlexNet is widely used for performance comparison in different CNN acceleration work. Hence, it allows us to compare our results with prior art. We used two other well-known CNNs, SqueezeNet and GoogLeNet, to demonstrate our results are not CNN-dependent. SqueezeNet is tailored towards IoT applications, given its small memory footprint. GoogLeNet is an incarnation of the Inception architecture developed by Google. The Inception family has state-of-the-art performance in classification and is being used by Google [12]. For each layer of these CNNs, we measure the execution time for the following cases:

(1) **Baseline:** Cappuccino synthesizes its original maximally parallel program using the finest thread granularity for all convolutional layers. As mentioned previously, the execution time for this case is never the fastest, but serves as a control for comparing the performance enhancement using our model. In a maximally parallel algorithm, the number of logical threads is equal to the number of output elements. Hence, each thread is responsible for computing one pixel in one Output Feature Map ($g = 1$). The parameter $g$ is defined and explained in Section 6.1.
(2) **Proposed:** Cappuccino uses the model we created to predict the best thread granularity for each convolutional layer of the given CNN. The execution time is used as a performance metric to compare against the baseline and ideal thread granularities.
(3) **Ideal:** Gauging how well our model is performing requires an understanding of the ideal granularity's performance. To gain this insight, we measured the execution time of all thread granularities for every convolutional layer of the given CNN. The one that yielded the shortest execution time was labeled to be the ideal granularity and its execution time is used to quantify how well our model performed.

Each experiment was repeated 100 times for reproducibility, and the average execution time and standard deviation are reported in Table 2.

### 7.3 Training Frequency

We used the frequency on which the original model was trained ($f = 1497MHz$) to evaluate the model in this first round of experiments. Keeping the training and testing frequency invariant allowed us to investigate how well our model performs on CNNs that it was not trained on without changing too many variables.

Table 2. Impact of Thread Granularity on Execution Time. Each Experiment has been Repeated 100 Times and Results are Reported in the Form of $a \pm s$. Where $a$ is the Average and $s$ is the Standard Deviation. All Experiments are Performed on Nexus 5 and for All of Them Background Processes are Killed

| # | CNN Name | DVFS | Freq. (MHz) | Execution Time (ms) | | | Diff. with ideal | Speedup |
|---|---|---|---|---|---|---|---|---|
| | | | | Baseline | Proposed | Ideal | | |
| 1 | GoogLeNet | Off | 1497 | 3267 ± 182 | 1389 ± 116 | 1331 ± 113 | 4.36% | 2.35X |
| 2 | SqueezeNet | Off | 1497 | 1883 ± 88 | 866 ± 58 | 849 ± 49 | 2.00% | 2.17X |
| 3 | AlexNet | Off | 1497 | 1172 ± 34 | 512 ± 33 | 509 ± 33 | 0.59% | 2.29X |
| 4 | GoogLeNet | Off | 1574 | 3088 ± 187 | 1327 ± 118 | 1266 ± 109 | 4.82% | 2.33X |
| 5 | SqueezeNet | Off | 1574 | 1784 ± 75 | 819 ± 48 | 812 ± 49 | 0.86% | 2.18X |
| 6 | AlexNet | Off | 1574 | 1143 ± 35 | 488 ± 28 | 480 ± 27 | 1.67% | 2.34X |
| 7 | GoogLeNet | Off | 1728 | 2839 ± 161 | 1222 ± 107 | 1167 ± 101 | 4.71% | 2.32X |
| 8 | SqueezeNet | Off | 1728 | 1628 ± 92 | 744 ± 55 | 739 ± 54 | 0.68% | 2.19X |
| 9 | AlexNet | Off | 1728 | 1004 ± 31 | 440 ± 23 | 435 ± 21 | 2.33% | 2.28X |
| 10 | GoogLeNet | On | N/A | 2651 ± 210 | 1120 ± 152 | 1058 ± 137 | 5.86% | 2.37X |
| 11 | SqueezeNet | On | N/A | 1302 ± 105 | 598 ± 63 | 585 ± 61 | 2.22% | 2.18X |
| 12 | AlexNet | On | N/A | 847 ± 61 | 370 ± 30 | 361 ± 30 | 2.49% | 2.29X |

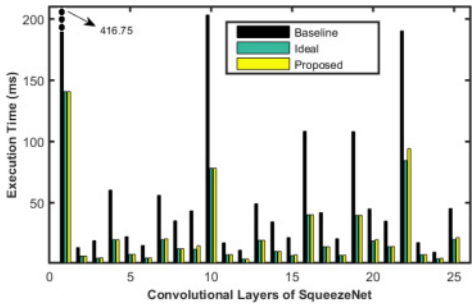


Fig. 10. Execution times of SqueezeNet under the default, proposed, and ideal granularity. In all experiments the frequency is fixed at $1497MHz$.

*7.3.1 Model Assessment.* Figure 10 shows the execution time for each convolutional layer in SqueezeNet using the baseline, ideal, and proposed granularity. The difference in execution times between using the finest and ideal granularity clearly shows how important thread granularity is in the execution time of a parallel algorithm. Under the frequency of 1497MHz, algorithms with the baseline, ideal, and proposed granularity take 1883 (ms), 849 (ms), and 866 (ms), respectively. In addition, the difference between the proposed thread granularity and the ideal granularity is 2.00%. We performed the same experiment for GoogLeNet and AlexNet and the results are shown in rows 1 and 3 of Table 2, respectively. For GoogLeNet, the case which uses the proposed thread granularity is 4.4% slower than ideal case. For AlexNet, prediction works even better and the proposed model is only 0.59% slower than the ideal case.

*7.3.2 Impact of Thread Granularity.* The fifth column of Table 2, shows the execution time of a maximally parallel algorithm (baseline). Unlike common GPGPU practices, this approach does not have the best performance on SoCs. For GoogLeNet under a fixed-frequency of $f = 1497MHz$, the execution time of the parallel algorithm with finest thread granularity is 3267 (ms). However, when we use the offered model to predict and select the optimal thread granularity, the execution time

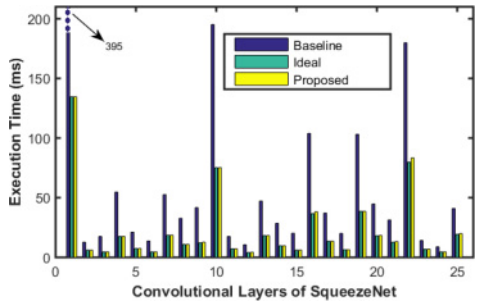


Fig. 11. Execution times for the default parallel program, parallel program with granularity prediction, and ideal parallel program. In all experiments the frequency is frozen on $1574MHz$.

is 1389 (ms). Hence, using the offered model to predict the best thread granularity accelerated the parallel algorithm for GoogLeNet by 2.35X. Similarly, SqueezeNet and AlexNet were respectively accelerated by 2.17X and 2.29X using the offered model.

### 7.4 Test Frequencies

To evaluate the expandability of the proposed model, it is required to see its performance when the frequency is not the same during the train and test for frequency scaling). For this purpose, we used the model that was trained on frequency of $1497MHz$ to predict thread granularities for layers of different CNNs under the frequency of $f = 1574MHz$ and $f = 1728MHz$.

*7.4.1 Model Assessment.* The execution times of running different layers of SqueezeNet under the frequency of 1574MHz is shown in Figure 11. For all layers of SqueezeNet except for layer #22 the execution time of the predicted thread granularity closely follows the ideal case. For SqueezeNet under the frequency of ($f = 1574MHz$) the difference between the execution time of the ideal algorithm and the algorithm with predicted thread granularity is 0.86%. This number is 4.8% for GoogLeNet under the same frequency.

*7.4.2 Impact of Thread Granularity.* In this case (fixed-frequency on $f = 1574MHz$), the baseline parallel algorithm (finest thread granularity) takes 3088 (ms) for GoogLeNet and 1784 (ms) for SqueezeNet. Using the optimized thread granularity, it is possible to speedup these algorithm by 2.33X and 2.18X, respectively.

### 7.5 Dynamic Frequency

Even though the proposed model works for fixed values of frequency, in practice the frequency of a platform is not fixed. Therefore, we evaluated the performance of the proposed model when DVFS is on. In this part, we restored DVFS to its default mode. Therefore, each layer of a CNN may be processed with different frequencies. To address this, before launching a convolution kernel we query the processor to read the current working frequency and use that to select the best thread granularity. However, the frequency may change after launching the kernel resulting in sub-optimal performance for a portion of a layer. Regardless, the execution time of each kernel is small compared to the rate of frequency changes, and thus, the effect on performance is limited.

*7.5.1 Model Assessment.* For AlexNet, the proposed granularity takes 370 (ms) for execution and the ideal case takes 361 (ms). Thus, the predicted thread granularity is 2.49% slower. The proposed thread granularity for GoogLeNet is 5.86% slower than the ideal case. Finally, the execution time of SqueezeNet in ideal case is 585 (ms), whereas the execution time for the proposed thread granularity is 598 (ms). Hence, the proposed thread granularity is 2.22% slower than the ideal case.

Table 3. Impact of Thread Granularity on Execution Time. Results are Reported in the Form of $a \pm s$. Where $a$ is the Average and $s$ is the Standard Deviation for All Experiments. DVFS has been Active for All Cases. Rows #1 to #3 Show the Effectiveness of the Model for a New Platform. Row #4 to #9 Demonstrate that the Proposed Model Continues to be Effective Even When Phones are in the Normal Mode (Typical Background Processes are Running)

| # | CNN Name | Platform | Background Processes | Execution Time (ms) | | | Diff. with ideal | Speedup |
|---|---|---|---|---|---|---|---|---|
| | | | | Baseline | Proposed | Ideal | | |
| 1 | AlexNet | Nexus 6P | Off | 512 ± 51 | 275 ± 20 | 270 ± 19 | 1.8% | 1.86X |
| 2 | SqueezeNet | Nexus 6P | Off | 690 ± 52 | 386 ± 21 | 370 ± 20 | 4.3% | 1.79X |
| 3 | GoogLeNet | Nexus 6P | Off | 1575 ± 187 | 855 ± 123 | 834 ± 103 | 2.5% | 1.84X |
| 4 | AlexNet | Nexus 5 | on | 946 ± 64 | 401 ± 19 | 371 ± 17 | 8.0% | 2.36X |
| 5 | SqueezeNet | Nexus 5 | on | 1564 ± 205 | 712 ± 101 | 683 ± 99 | 4.2% | 2.20X |
| 6 | GoogLeNet | Nexus 5 | on | 2744 ± 222 | 1151 ± 160 | 1061 ± 142 | 8.4% | 2.38X |
| 7 | AlexNet | Nexus 6P | on | 532 ± 53 | 291 ± 30 | 284 ± 31 | 2.4% | 1.83X |
| 8 | SqueezeNet | Nexus 6P | on | 765 ± 78 | 424 ± 37 | 400 ± 40 | 6.0% | 1.80X |
| 9 | GoogLeNet | Nexus 6P | on | 1640 ± 225 | 909 ± 139 | 890 ± 120 | 2.1% | 1.80X |

Table 4. Performance Comparison Between the Proposed Solution, Cappuccino [18], and CNNDroid [14] on AlexNet. Even Though the Proposed Solution has been Launched on an Older SoC (Snapdragon 800) It is 1.92X Faster Compared to CNNDroid

| | CNNDroid [14] | Cappuccino [18] | Proposed |
|---|---|---|---|
| SoC | Qualcomm Snapdragon 810 | Qualcomm Snapdragon 800 | Qualcomm Snapdragon 800 |
| Execution Time (ms) | 709 | 874 | 370 |

As Table 2 shows, for different working conditions, the proposed thread granularity follows the **ideal** case closely. The difference between execution time of the proposed case and the ideal case varies from 0.59% in the best case to 5.86% in worst case. As the achieved results demonstrate, the proposed model works effectively and yields near ideal results.

*7.5.2 Impact of Thread Granularity.* Use of a correct thread granularity when DVFS is on can further accelerate AlexNet by 2.29X compared to a maximally parallel algorithm. Likewise, an optimal choice of thread granularity using our model further accelerates GoogLeNet and SqueezeNet by 2.37X and 2.18X, respectively.

As we observed, the decision on the number of threads we want to launch for performing a task is not a trivial decision. Choosing the correct thread granularity can easily make a parallel algorithm twice faster. In our experiments, choosing the correct thread granularity accelerates the algorithm by 2.18X on average.

We repeated all experiments when phones were in the normal condition (typical background processes were running) to indicate that the proposed approach continues to be effective in this scenario. The results are reported in rows #4 to #9 of Table 3. We also compared the performance of the proposed work with Cappuccino [18] and CNNDroid [14] in Table 4.

### 7.6 Energy Consumption

In this section we evaluate the impact of thread granularity on energy consumption.

*7.6.1 Experimental Setup.* In order to measure the overall energy consumption on a mobile platform, we used the Trepn Power Profiler, a profiling application developed by Qualcomm for

Table 5. Impact of Thread Granularity on Energy

| | Baseline (J) | | | Proposed (J) | | | |
|---|---|---|---|---|---|---|---|
| CNN Name | First 1000 | Second 1000 | Average | First 1000 | Second 1000 | Average | Ratio |
| SqueezeNet | 3.388 | 3.363 | 3.376 | 1.726 | 1.788 | 1.757 | 1.92X |
| GoogLeNet | 6.123 | 6.145 | 6.134 | 3.401 | 3.446 | 3.424 | 1.79X |

investigating power usage and performance on mobile devices. It can report the power consumption of a specific application, as well as the overall system. The energy consumption was calculated using the finest temporal resolution available with Trepn, (i.e., 100 (ms)).

The energy consumption using the baseline and proposed granularities for a CNN was resolved by finding the total consumption of running a CNN 1000 times and calculating the average consumption per run.[4] To reduce potential sources of measurement noise (non-contributory energy consumption), we killed all unnecessary background processes, put the phone in airplane mode, and dimmed the screen during these experiments. However, we enabled DVFS since it plays a significant role in the overall energy consumption under a mobile devices normal working conditions. Measurements were performed twice for repeatability (2000 runs total).

*7.6.2 Impact of Thread Granularity.* For our experiments, we profiled the energy consumption of SqueezeNet and GoogLeNet on the Google Nexus 5 mobile platform using the methodology described previously. These measurements are reported in Table 5.

For SqueezeNet and GoogLeNet respectively, using the baseline granularity consumes an average of 3.4 joules and 6.1 joules. However, the proposed granularity only uses 1.8 joules and 3.4 joules, respectively. This means that an appropriate thread granularity improves the energy consumption by 1.9X for SqueezeNet and 1.8X for GoogLeNet. An optimal thread granularity improves the execution time by efficient utilization of processors. Such a utilization keeps the processors busier compared to the baseline case. This slightly increases the power consumption. A considerable reduction in the execution time and a slight increase in the power consumption leads to a decrease in energy consumption. For example, optimizing the granularity in SqueezeNet on Nexus 5 decreases the execution time by 2.18X and increases the power consumption by 1.14X. Hence, it decreases the energy consumption by 1.92X compared to the baseline.

## 8 CONCLUSION

In this paper, we studied the effect of thread granularity on parallel execution of CNNs on mobile SoCs. First, we explained that unlike common GPGPU methodologies, an algorithm which is maximally parallelized does not offer the smallest execution time. We showed that for different cases, judicious choice of thread granularity can offer up to 2.37X speedup compared to an algorithm that is maximally parallelized. Subsequently, we offered a learning based model that is able to select the best thread granularity for a given convolutional layer. Finally, we showed that algorithms written based on the thread selection policy of the offered model yield near ideal results. Our experimental results show that using the offered model can improve the execution time of a fully parallelized CNN by up to 2.37X and improves its energy consumption by up to 1.9X.

## REFERENCES

[1] 2010. Copyright office provides exemption to DMCA. (2010). https://www.copyright.gov/1201/.

[4] In a previous work [17], we used Trepn to measure the energy only when the GPU was active. In this work, we measure the total energy consumption of the application during its life cycle.

[2] Srimat Chakradhar, Murugan Sankaradas, Venkata Jakkula, and Srihari Cadambi. 2010. A dynamically configurable coprocessor for convolutional neural networks. In *ACM SIGARCH Computer Architecture News*, Vol. 38. ACM, 247–257.
[3] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and others. 2014. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 609–622.
[4] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. 2011. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*.
[5] Philipp Gysel, Mohammad Motamedi, and Soheil Ghiasi. 2016. Hardware-oriented Approximation of Convolutional Neural Networks. *arXiv preprint arXiv:1604.03168* (2016).
[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 770–778.
[7] Po-Kuan Huang, Matin Hashemi, and Soheil Ghiasi. 2008. System-level performance estimation for application-specific MPSoC interconnect synthesis. In *Application Specific Processors, 2008. SASP 2008. Symposium on*. IEEE, 95–100.
[8] 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 0.5 MB model size. *arXiv preprint arXiv:1602.07360* (2016).
[9] Haris Javaid, Aleksander Ignjatovic, and Sri Parameswaran. 2010. Fidelity metrics for estimation models. In *Proceedings of the International Conference on Computer-Aided Design*. IEEE Press, 1–8.
[10] 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 *Proceedings of the 22nd ACM International Conference on Multimedia*. ACM, 675–678.
[11] N. Jouppi. 2016. Google supercharges machine learning tasks with TPU custom chip. *Google Blog, May* 18 (2016).
[12] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. *arXiv preprint arXiv:1704.04760* (2017).
[13] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*. 1097–1105.
[14] Seyyed Salar Latifi Oskouei, Hossein Golestani, Matin Hashemi, and Soheil Ghiasi. 2016. CNNdroid: GPU-Accelerated Execution of Trained Deep Convolutional Neural Networks on Android. In *Proceedings of the 2016 ACM on Multimedia Conference*. ACM, 1201–1205.
[15] Alberto Magni, Christophe Dubach, and Michael O'Boyle. 2014. Automatic optimization of thread-coarsening for graphics processors. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*. ACM, 455–466.
[16] Alberto Magni, Christophe Dubach, and Michael F. P. O'Boyle. 2013. A large-scale cross-architecture evaluation of thread-coarsening. In *High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for*. IEEE, 1–11.
[17] Mohammad Motamedi, Daniel Fong, and Soheil Ghiasi. 2016. Fast and Energy-Efficient CNN Inference on IoT Devices. *arXiv preprint arXiv:1611.07151* (2016).
[18] Mohammad Motamedi, Daniel Fong, and Soheil Ghiasi. 2017. Cappuccino: Efficient Inference Software Synthesis for Mobile System-on-Chips. *arXiv preprint arXiv:1707.02647* (2017).
[19] Mohammad Motamedi, Philipp Gysel, Venkatesh Akella, and Soheil Ghiasi. 2016. Design space exploration of fpga-based deep convolutional neural networks. In *Design Automation Conference (ASP-DAC), 2016 21st Asia and South Pacific*. IEEE, 575–580.
[20] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)* 115, 3 (2015), 211–252. https://doi.org/10.1007/s11263-015-0816-y.
[21] 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 *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 1–9.
[22] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 161–170.