Integer ConvNets on embedded CPUs: Tools and performance assessment on the cortex-A cores

Integer ConvNets on Embedded CPUs: Tools and Performance Assessment on the Cortex-A Cores

Valentino Peluso, Antonio Cipolletta, Francesco Vaiana, Andrea Calimera
Politecnico di Torino, 10129 Torino, Italy

***Abstract*—Quantization via fixed-point representation is commonly used to reduce the complexity of Convolutional Neural Networks (ConvNets). It is particularly suited for accelerating edge-inference on embedded devices as it enables to reduce resource requirements with no loss of prediction quality. However, porting integer ConvNets on low-end CPUs is not straightforward: it calls for proper software design and organization with a high degree of hardware awareness. Today there are plenty of fixed-point libraries integrated into different inference engines which provide design support. The aim of this work is to review the most stable tools and analyze their performance on different use-cases processed on embedded boards powered by Arm Cortex-A cores. The collected results provide an interesting analysis with useful guidelines for developers and hardware designers.**

I. Introduction

There is a very active research branch, at the intersection between deep learning and embedded systems, that is facing a new challenge: move the inference stage from the cloud to the edge, namely, on board of end-nodes like mobile phones, drones, vehicles, etc. The availability of smart systems capable of locally inferring highly informative data would bring about the predictability of the service response time, more user privacy, and the cut of continuous internet connectivity.

In order to meet the stringent energy constraints of mobile applications, a typical end-node has too few computational resources to guarantee the processing of ConvNets with an acceptable latency. While new custom hardware accelerators for the mobile segment are expected to reach massive production soon, there are orthogonal approaches which rely on software acceleration [1], [2]. These methods include compression techniques that shrink the complexity of the ConvNets thus to fit low-cost embedded CPUs already available on the market, e.g. the Cortex-A cores developed by Arm. In this context, quantization is one of the most effective technique. It translates a pre-trained floating-point ConvNet into an integer model without loss of accuracy. Recent studies demonstrated that 8-bit fixed-point integers ensure the same accuracy of 32-bit floating-point. The advantages brought by fixed-point quantization are many. First, it allows to linearly reduce the memory size of the model. Second, it is orthogonal to other compression techniques like pruning and neural architecture search. Third, it can reduce latency due to higher cache utilization and lower memory bandwidth.

The focus of this work is on the third point. The fact that integer ConvNets get faster just because they use scaled bit-widths cannot be taken as granted. To achieve speed-up requires a careful code optimization without which the potential savings turn into overhead. Let's consider the Cortex-A architecture. It comes with the NEON unit, an advanced single-instruction multiple-data (SIMD) architecture that supports integer vector operations. A proper use of such module allows to maximize the parallelism, reduce the memory accesses and hence achieve substantial performance boost. However, a correct execution of multiply-and-accumulate operations in fixed-point asks for additional instructions not needed in floating-point, such as data extension, arithmetic shifts, etc. An accurate schedule is crucial to avoid under-utilization of the local memories (register file and caches) and the execution lanes. It is clear that custom code optimization and a proper compiling strategy are paramount.

Being-aware of this issue, both industry and academia are investing in new tools and libraries that can support the deployment of integer ConvNets. The most representative examples are Arm NN[1] by Arm, ncnn by Tencent[2], and TensorFlow Lite by Google[3]. These tools provide an abstract user interface coupled with specialized deep learning libraries that collect hand-written kernels built with NEON intrinsics and/or assembly code. Given the custom nature of such tools and kernels, it is natural to ask whether they can generalize over multiple tasks and use-cases. Different ConvNets make use of several layers that may require alternative kind of optimization (e.g. traditional convolution vs depth-wise convolution). Commonly adopted embedded CPUs may have a different instruction set (e.g. ARMv7 vs. ARMv8), cache memories differently sized, and multiple clock frequencies. Moreover, there might be cases where multi-thread execution can be adopted to boost performances.

The complexity of the problem recalls the synthesis of digital circuits, where different descriptions of the same module may lead to different quality-of-design, also depending on the target technology node. The final solution is the result of an iterative design flow tuned against the specific needs. Against this background, the objective of this work is to try answering the following questions: *Are existing tools ready to handle the heterogeneity of ConvNets on embedded systems? Are they scalable? What's the exact level of performance they achieve?*

The paper first reviews existing tools and libraries for the deployment of integer ConvNets on the Cortex-A CPUs. Then, it introduces a performance assessment using different

[1]https://mlplatform.org/
[2]https://github.com/Tencent/ncnn
[3]https://www.tensorflow.org/lite

architectures and benchmarks, with emphasis on the aforementioned aspects.

II. Inference Engines For Integer ConvNets

The success of deep learning is driven by the availability of open-source frameworks that accelerate training and inference of deep neural networks. The most representative examples are TensorFlow by Google and PyTorch by Facebook. Deep learning frameworks support automatic differentiation (needed for back-propagation) and integrate optimized computing libraries for the acceleration on GPUs and high-end CPUs. On top of that, they offer a high-level interface that allows easy use of the most common neural network layers. A designer just needs to provide a model description, typically through a Python class, and specify the training options (learning rates, epochs, etc.) with few lines of code. These tools are not devised for deploying and serving trained models on embedded devices, hence the performance they achieve on low-end CPUs shows substantial margins for optimization. Other solutions tailored for edge-inference emerged in the last years. Referred as *inference engines*, they serve as a bridge between deep learning frameworks and embedded platforms.

Similarly to deep learning frameworks, inference engines are structured on two layers: *back-end* and *front-end*. The back-end is at the lowest level and its function is to collect highly optimized neural kernels, generally written using intrinsics or assembly code. Since the mobile market offers diverse hardware solutions (CPUs, GPUs, and DSPs), state-of-the-art inference engines provide different back-ends. The front-end operates at the highest level providing users with an interface to import a pre-trained ConvNet and to run optimization for the low-level implementation. The selection of the back-end is done at compile time depending on the target architecture. It is worth emphasizing that inference engines not only improve the intrinsic performance of a ConvNet model, but also significantly reduce the code-size by dropping those functions that are not needed during the feed-forward pass of the network.

The earliest versions of the inference engines were mainly focused on the acceleration of floating-point ConvNets, but with the recent advancements of compression methods and the large demand for embedded applications, the new releases integrated execution in fixed-point. Even though quantization methods that leverage extreme bit-widths, e.g. 3 or 4-bit [3], are currently available in literature, 8-bit represents the most suited option. The Arm instruction set supports SIMD integer instructions down to 8-bit, and playing with lower precision would require extra operations to correctly feed the execution units incurring latency penalty. That is why most of the available tools give support for 8-bit only. As it will be detailed in the next section, the implementation of 8-bit integer ConvNets poses several challenges. Different methods do exist which reflect the availability of several integer libraries and back-ends. The study reported in this work considers those integrated into three production-ready inference engines, namely, *Arm NN*, *ncnn*, *TensorFlow Lite*, considered the most advanced and stable solutions for Cortex-A CPUs.

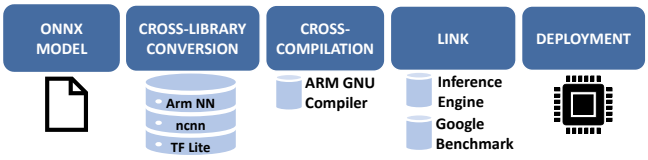

Figure 1. Abstract view of the benchmarking framework.

III. Fixed-Point Libraries

The belief that quantization alone leads to simpler arithmetic and then to higher performance is not exactly true. For such specific reason, the back-end side of any inference engine is built upon dedicated low-level kernel libraries that collect customized integer routines for the most common deep learning layers, e.g. convolution, pooling, fully-connected, activation, etc. Much effort is focused on the optimization of convolutional kernels, well-known to be the most expensive and critical layers in state-of-the-art neural networks. The need for custom kernel optimization comes from several issues which are synthetically reviewed in this section. Indeed, the savings brought by quantization originate from a higher degree of freedom in orchestrating the hardware resources. This is at the same time an opportunity for optimization as well as a source of variability.

First, the shift towards fixed-point numbers requires a quantization scheme, either asymmetric (Arm NN and TensorFlow Lite) or symmetric (ncnn). Regardless the adopted strategy, both schemes need extra arithmetic operations which may introduce overhead. The latter can be reduced by an efficient design of input and output processing stages as dicussed in [1], achieving variable gains depending on the underlying architecture.

Second, the most common implementation of convolution is based on general matrix-multiplication (GEMM) [4]. In its general formulation, GEMM encompasses the transformation of the multi-dimensional input tensors into two-dimensional matrices. The dimension of these matrices is too large to fit into the typical caches of embedded cores. Therefore, the workload is split in smaller bunches of data processed sequentially. The optimal size of these bunches may vary depending on the ConvNet model and the target architecture.

Third, due to lower bit-width, 8-bit quantization enables to store more data in the same memory, hence it reaches higher cache hits. Furthermore, it alleviates the memory bandwidth, as a single load/store operation can move multiple data within the same instruction. However, identifying a unique strategy for dispatching the data and feeding the arithmetic units is not straightforward. The optimal solution depends on several factors, such as the convolution parameters (filter size, strides, etc.) which determines the dimension of matrices, the size of caches and register file, the instruction set (32-bit vs 64-bit).

Fourth, a proper implementation of multiply-and-accumulate (MAC) operations for integers needs particular attention to avoid overflow and fast saturation. For instance, MACs between 8-bit operands need 32-bit registers for accumulation; in general-purpose CPUs, this calls for custom

Table I
OVERVIEW OF THE SELECTED BENCHMARKS.

Model	**# Weights** [M]	**# MACs** [M]	**Top-1** (%)	**Top-5** (%)
VGG19	143.7	19632	73.83	91.79
ResNet50 v1	25.5	3864	74.93	92.38
MobileNet v2	3.5	429	70.94	89.99

variable allocations into the register file, as well as different scheduling of extension, multiplication, and sum reduction, that can be implemented in many ways.

Last but not least, multi-thread acceleration can be managed through different policies. Each inference engine adopts its own. For instance, there exist custom libraries based on *pthread* (like those in Arm NN and Tensorflow Lite) that statically distribute the workload over the available cores. Another solution is to adopt the OpenMP pragmas (like in ncnn), where the optimization is left to the compiler.

From this qualitative analysis, it is clear that the performances of integer ConvNets depend on a large number of design variables. Hence, the orthogonal use of inference engines is questionable.

IV. BENCHMARKING FRAMEWORK

To evaluate the performances of integer ConvNets, a cross-library benchmarking framework has been designed. As a key feature, it guarantees the inter-operation of the same model among the different inference engines. A pictorial description is given in Fig. 1. The input is a pre-trained network in Open Neural Exchange (ONNX) format. The ONNX model is then translated in a format that is compliant with the specification of each target inference engine. This ensures the ConvNet under analysis has exactly the same architecture and parameters (e.g. strides, paddings, etc.) over all the experiments. The final output is a C++ application that loads and configures the model on the target core. The framework relies on the google benchmark suite [5] to enable the collection of performance statistics at run-time. The application is cross-compiled using the GNU Arm Embedded Toolchain and linked with the respective pre-compiled inference engine and the google benchmark suite. Finally, the executable is downloaded to the target device and the performance metrics are collected.

V. EXPERIMENTAL SET-UP AND RESULTS

A. Benchmarks

The selected benchmarks are three state-of-the-art ConvNets whose characteristics are reported in Table I. VGG and ResNet won the ImageNet competition in 2014 and 2015, respectively. MobileNet is a network designed for embedded applications. The table collects the number of trainable parameters (# Weights) and multiply-and-accumulate operations (# MACs) as well as the top-1 classification accuracy over the ImageNet validation set. The models are taken from the ONNX model zoo [6]. To notice that these networks are employed as features extractor in several computer vision tasks, including object detection, localization, and segmentation. Some examples are Single-Shot Detectors [7] and Faster R-CNN models [8].

The three benchmarks are representative of different kinds of workload: VGG is dominated by the fully-connected layers, hence is memory-bounded; ResNet is thinner and deeper and contains one single fully-connected layer as the last stage for classification; MobileNet is a compact model (lower number of parameters and operations) built upon depth-wise convolutions. Such diversification is paramount to assess the flexibility of the inference engines.

Table II
HARDWARE SPECIFICATIONS OF THE TARGET BOARDS.

Board	**ISA**	**CPU**	**Freq.** MHz	**RAM** GB	**L1** kB	**L2** kB
C2	ARMv8	A53	1536	2	32	512
XU4	ARMv7	A7	1400	2	32	512
		A15	2000	2	32	2048

B. Hardware platforms

Two different commercial off-the-shelf boards were used as test-benches: the ODROID-C2 and the ODROID-XU4. They are powered by different versions of the Arm Cortex-A CPU, an architecture largely employed on SoCs for embedded applications. The technical specifications are summarized in Table II. The C2 board hosts the Amlogic S905 SoC based on a quad-core Cortex-A53 CPU, which belongs to the ARMv8 family. The XU4 board is powered by the Samsung Exynos 5422 SoC, a 32-bit big.LITTLE (ARMv7) integrating a high-performance A15 CPU and a low-power A7 CPU, for a total of 8 cores (4 core each); in our experiments, the two clusters are used separately. Both the target boards run the Ubuntu 16.04 Mate distro released by Hardkernel. Multiple inference runs have been iterated enabling 1, 2, 3, and 4 threads. All the experiments are run with maximum clock frequency (column *Freq.* in Table II).

C. Inference Engines and Toolchains

To ensure reproducibility of the experiments, we report the versions of each tool integrated in the benchmarking framework: Arm NN, ver. 19.05; ncnn, commit number `bcfe9f4`; TensorFlow Lite, ver. 1.14; google benchmark suite, ver. 1.5.0; Arm GNU Cross-Toolchain released by Linaro, ver. 6.5.

D. Experimental Results

Results are collected in Table III. Each column refers to a different engine: Arm NN (A), ncnn (N), and TensorFlow Lite (T), respectively. Reported values indicate the inference time averaged over 100 runs, interleaved by a two-second pause to avoid thermal throttling; numbers in bold identify the best performance for each hardware-ConvNet configuration.

Arm NN reveals slower than its competitors. However, given the high variability of results across different configurations, it is fair to assume that there may be other use-cases with different trends, e.g. other ConvNets or target cores. This can be understood observing single-thread executions with the Cortex-A53, where the performance gap highly depends on the benchmark: for ResNet, Arm NN is only 33% slower than the other engines, while for the other networks we observed a much higher execution times ($> \times 2$). To notice that for the XU4 (rows A15 and A7), the results for VGG are not reported as the benchmark crashed due to out-of-memory execution. As general trend we observed Arm NN gets different memory

Table III
AVERAGE LATENCY [S] OVER 100 RUNS. THE DASH (–) INDICATES EXECUTION FAILED FOR OUT-OF-MEMORY ERROR. BEST RESULTS ARE HIGHLIGHTED IN BOLD (LOWER IS BETTER).

CPU	ConvNet	1 Thread			2 Threads			3 Threads			4 Threads		
		A	N	T	A	N	T	A	N	T	A	N	T
A15 (XU4)	VGG19	–	**2.12**	2.79	–	**1.34**	1.53	–	**0.95**	1.14	–	**0.92**	0.95
	ResNet50	3.40	0.75	**0.58**	1.77	0.42	**0.33**	1.23	0.30	**0.25**	0.96	0.26	**0.21**
	MobileNet v2	1.60	0.17	**0.15**	0.83	**0.09**	**0.09**	0.57	**0.06**	0.07	0.45	**0.05**	0.06
A53 (C2)	VGG19	13.79	**3.77**	5.63	10.25	**2.25**	2.78	8.98	**1.67**	1.95	8.40	1.63	**1.56**
	ResNet50	1.52	1.15	**1.13**	0.79	0.64	**0.62**	0.55	0.45	**0.45**	0.42	0.42	**0.37**
	MobileNet v2	0.43	**0.22**	0.25	0.23	**0.11**	0.15	0.16	**0.08**	0.12	0.13	**0.07**	0.11
A7 (XU4)	VGG19	–	**9.02**	12.99	–	**5.22**	6.55	–	**3.36**	4.67	–	**3.28**	3.72
	ResNet50	11.28	**2.30**	2.61	5.73	**1.21**	1.44	3.92	**0.81**	1.06	3.01	**0.70**	0.84
	MobileNet v2	4.86	**0.40**	0.61	2.46	**0.20**	0.34	1.68	**0.14**	0.25	1.30	**0.10**	0.20

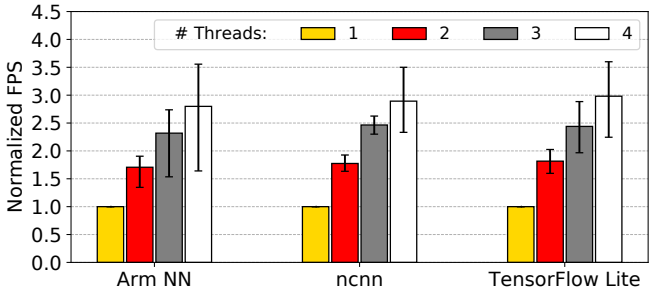

Figure 2. Impact of multi-thread execution on the Cortex-A53 CPU.

usage depending on the hardware parallelism, 32- vs. 64-bit These findings suggest that Arm NN makes use of different memory allocation strategies on the two back-ends.

For a more intelligible analysis, the next paragraphs comments on the achieved results addressing the three most influencing factors: topology of the ConvNet (benchmark), core architecture (hardware), parallel execution (multi-thread). Unless specified, the analysis refers to single-thread execution.

Benchmark. ncnn achieves the best performances on VGG, while TensorFlow Lite shows the lowest latency on ResNet. Looking at the network architectures, this observation suggests that ncnn kernels are more efficient for fully-connected layers, the most expensive in the former ConvNet; instead, TensorFlow Lite is faster in traditional convolutions. For the two engines, we measured similar execution times in MobileNet, which is dominated by depth-wise convolutions. The performance of the target engine is affected by the network topology. An exception can be found for the A7 core, where ncnn gets better in all cases. These findings demonstrate the same optimization brings to different advantages depending on the underlying hardware.

Hardware. A more in-depth analysis concerning the dependence from the target CPU shows interesting insights. Specifically, the availability of more resources affects the performance of the inference engines differently. First, Arm NN poorly exploits the higher computational power of the A15; surprisingly, we observed a faster execution on the A53 ($\times 2.9$ on average over the three networks). This gives additional evidence that Arm NN is poorly optimized for the ARMv7 instruction set. On the low-power A7, ncnn is the most efficient solution in all benchmarks, processing up to 2.5 frame/s with MobileNet. On the high-performance A15, TensorFlow Lite gets better resource utilization, with an average speed-up of $4.4\times$ compared to the A7; instead, for ncnn the throughput improvement is limited to $3.2\times$.

Multi-thread. Fig. 2 gives a synthetic overview of the multi-thread analysis. For each engine, the bars show the frame/s (FPS) normalized w.r.t. single-thread execution averaged over the three benchmarks. Results refer to the A53 CPU. The variance lines quantify the min-max range over the three ConvNets. For all the inference engines the performance linearly improves with the number of threads, yet with different slope and variance. Moreover, the benefits of multi-thread largely varies across the networks: for all the selected engines, VGG showed the lowest scalability (lowest tail of the error bars). Since VGG represents a memory-intensive workload, this trend might reveal an under-utilization of caches, which overwhelms the benefits of more execution units.

VI. CONCLUSIONS

In this work we assessed the performance of existing design tools for the deployment of integer ConvNets on embedded CPUs. A unified benchmarking framework was used to minimize the variability introduced by different front-ends, still the back ends introduce many sources of performance variability. Empirical results revealed that the implementation of integer ConvNets is still an open problem, since existing solutions are optimal only in specific corner cases. The reported analysis is a first guideline that may help designers and developers in deciding the engine that gets the best from hardware.

REFERENCES

[1] B. Jacob *et al.*, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 2704–2713.

[2] V. Peluso *et al.*, "Weak-mac: Arithmetic relaxation for dynamic energy-accuracy scaling in convnets," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2018, pp. 1–5.

[3] F. Tung *et al.*, "Clip-q: Deep network compression learning by in-parallel pruning-quantization," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 7873–7882.

[4] K. Goto *et al.*, "Anatomy of high-performance matrix multiplication," *ACM Transactions on Mathematical Software (TOMS)*, vol. 34, no. 3, p. 12, 2008.

[5] A microbenchmark support library. [Online]. Available: https://github.com/google/benchmark

[6] A collection of pre-trained, state-of-the-art models in the onnx format. [Online]. Available: https://github.com/onnx/models

[7] W. Liu *et al.*, "Ssd: Single shot multibox detector," in *European conference on computer vision*. Springer, 2016, pp. 21–37.

[8] S. Ren *et al.*, "Faster r-cnn: Towards real-time object detection with region proposal networks," in *Advances in neural information processing systems*, 2015, pp. 91–99.