Measuring Inference Performance of Machine-Learning Frameworks on Edge-class Devices with the MLMark™ Benchmark

Peter Torelli
President, EEMBC
peter.j.torelli@eembc.org

Mohit Bangale
Senior Engineer - ML, Ignitarium
mohit.bengale@ignitarium.com

This paper explains how EEMBC—a non-profit consortium of embedded technology companies—developed the MLMark benchmark for characterizing machine-learning inference on edge devices, and discusses results obtained running the benchmark on multiple accelerators.

1 Introduction

In the past five years, machine learning (ML) as a practical application has consumed the vast majority of research in computer engineering. Despite 50 years of research on this topic, only recently has the technological environment exposed researchers to affordable, high-performance computers designed for ML acceleration. The *cloud-edge* paradigm is entirely suited to ML development: training done in the *cloud*, with its bottomless-well of compute resources, and inference performed on the *edge*, at a billion locations in a space once referred to as "ubiquitous computing." In this paper we are interested in the characteristics of the billions of tiny devices rapidly emerging at the edge. These devices have significant constraints on energy use, size and cost; constraints which point back to a need for effective performance analysis, which in turn requires an effective benchmark.

In this paper, we will first provide a short overview of benchmarking from our perspective, followed by the motivation for exploring the machine learning domain. Next the paper covers how we applied our traditional methodology to the problem by developing the MLMark™ benchmark[1] with a team comprised of engineers from Intel, NVIDIA, Arm, TI, Ignitarium, Flex, and several other member companies. Lastly, we will review data that has been collected to date, with some preliminary conclusions.

2 EEMBC history and motivation

EEMBC was founded by EDN in 1997 with the goal of addressing the lack of benchmarks in the embedded microprocessor industry. While desktop CPUs had plenty of options due to popularity of the Windows operating system, embedded platform benchmarks were limited to a few choices, and the industry had no standards body to represent its needs. After accomplishing its initial goal, EEMBC went on to address the changing needs of the industry by expanded into areas such as multicore processing; energy consumption of the MCU core and its peripherals (e.g., SPI, BLE); mobile phones; automated driving; and most recently, machine learning.

In the past, machine learning research depended on large and esoteric custom architectures, or in the last decade, huge arrays of GPUs or symmetric-core CPUs. Recently, an extensive R&D investment and academic research in this space has optimized two key variables: *software efficiency*, which has produced smaller, more effective neural nets requiring fewer resources (plus better-realized APIs to make research more accessible); and dedicated *hardware acceleration* that requires less power and has a lower cost. The combined effect of these optimizations has pushed these machine-learning devices down into EEMBC's benchmarking domain of embedded computing.

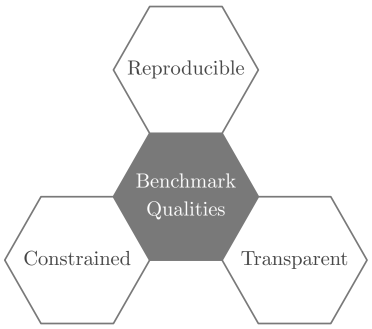
[1] https://www.mlmark.org

3 Benchmarks

A benchmark is a tool for distilling the behavior of a complex system into a single number, preferably one that increases to indicate the improvement of said system in response to an input change. This simple measurement capability enables the entire foundation of system-performance analysis because every model, every decision, every engineering tradeoff requires empirical grounding. Here are a few observations EEMBC has fostered over the years.

3.1 Benchmark qualities

For a benchmark to be effective, it must be *reproducible*, *transparent*, and *constrained*.



Reproducible means that the same inputs must generate the same output, within a small degree of tolerance for systemic or stochastic variation. No random components exist unintentionally in the benchmarks themselves, and tolerances are recommended when comparing scores or certifying them. MLMark has no random variation, the same images are used for each iteration, and the amount of runtime must exceed a certain minimum value to avoid timing resolution and operating system asynchronous behavior from interfering. MLMark also reports values to three significant digits, helping to further filter out these variations.

Transparent means anyone may view the means by which a score is generated, including code, algorithms, and run-rules. Some EEMBC benchmarks are licensed to help fund support and research, while others are available at no cost under an augmented Apache 2.0 license. MLMark is available on GitHub, and in order for someone to upload scores to the website database, the target source-code implementation must be first included in the repository for all to see.

Constrained, the most nuanced of the three requirements, refers to the how the benchmark enables flexibility while at the same time facilitating meaningful comparisons between very different systems.

One classic example of trying to maintain benchmarking constraints can be seen in the MCU CoreMark$^{®}$ benchmark. Consider the execution loops within parts of the benchmark, which are always the same length. If a compiler were to unroll the loops entirely, an MCU might perform better due to lack of branch misprediction. This optimization would illustrate a best-case scenario for that architecture under those fixed-loop conditions. However, if the number of iterations were to change by one, the performance would drastically change. If an outside observer was not privy to the compiler optimization, this discontinuity would seem jarring.

A good benchmark responds to small changes at the input with commensurate changes at the output. By constraining what is permissible in terms of optimizations, this kind of non-intuitive discontinuity vanishes. This is generally done with *run-rules*, which are procedural do's and don'ts, but can also be done by in many other ways such as limiting what portions of the benchmark may be altered and how, or by selecting specific input stimuli used in all measurements.

Going back to the CoreMark example: this doesn't mean that a developer or manufacturer cannot perform loop unrolling to show benefit, but it needs to be clearly stated the score was obtained out-of-spec (and thus is not valid for upload to the EEMBC database). MLMark addresses constraints by putting limiting the input datasets and models to specific versions. However, what the underlying framework actually does to optimize the model is unrestricted. We will discuss this more later under *accuracy measurement.*

The MLMark benchmark meets all three criteria: *reproducibility* by pre-selecting the exact stimuli and models that must be used and accounting for measurement precision; *transparency* by providing the source code and conversion scripts used to generate published scores, and *constrained* by only allowing the use of certain datasets and rigid run-rules policies while still allowing the framework to optimize at its discretion.

3.2 Benchmark components

Embedded benchmarks generally consist of a *test harness* and a *workload* (see Figure 1). The test harness has two jobs: interface the user to the benchmark, and interface the benchmark to the hardware. The workload is the actual operation(s) of interest to the benchmark.

3.2.1 Test harness

Historically, embedded devices have lacked the capacity to run a highly-advanced operating system. Many run "bare metal", where the workload is the only code present in the firmware, with no operating system. In these cases, the test harness usually runs on a remote operating system and communicates to the device via a serial debug port or a USB port. If the hardware has the capability, the test harness may also run on the target device.

MLMark is a set of interpreted Python scripts intended to run on Linux in version 1.0. MLMark provides an API abstraction layer to multiple targets. With this abstraction layer, the target may be the host CPU, a USB device, an FPGA, or any other type of accelerator: it is not relevant to the test harness if the DUT is local or remote. Some of the targets require interface layers to their frameworks. These are provided as libraries which have been pre-compiled for Ubuntu 16.04, but source code is provided for recompilation on different target platforms.

3.2.2 Workload

A workload consists of a *behavioral model* defining what the device under test actually does during the benchmark. This region of execution is fenced by timestamps to ensure the measurement happens as close to the execution as possible. In earlier benchmarks, the workload consisted of C-code that computed a specific task, like an FFT kernel, or an emulation of a real-world application, like the combination of a state-machine, XML parser, and a compression algorithm (see CoreMark-PRO). More recent EEMBC benchmarks include wireless transfers, security handshakes, etc.

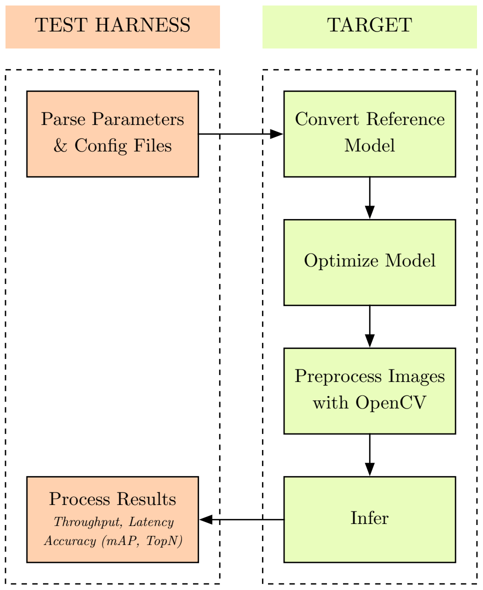


Figure 1. The MLMark software architecture API between test harness and target falls largely on the framework boundary.

In MLMark, the workloads consist of neural-net graphs, also known as *models*. In a similar analogy to older C-based benchmarks where the compiler optimized the C-code, models are optimized by the framework before execution on the hardware. For example, in MLMark the model for ResNet-50 is provided as a TensorFlow graph. When this is run on NVIDIA hardware, the TensorFlow graph is converted to a UFF format for NVIDIA hardware. During the conversion process, the NVIDIA framework, TensorRT, looks for hardware specific optimizations which may include a wide variety of operations such as node pruning or scheduling to various heterogeneous compute elements in the accelerator. The same thing happens with Intel hardware through OpenVINO and Google TPUs through their TPU compiler.

In the first version of MLMark, three vision models were chosen based on their popularity among academics and support from the industry: MobileNet V1.0, a small image classification network; ResNet-50 V1.5, a larger, more accurate image classifier; and SSDMobileNet v1.0, a single-shot detector using MobileNet for classification. Refer to the MLMark GitHub repository[2] for specific copyright information about the authors who graciously made these models available to the public.

[2] https://github.com/eembc/mlmark

The workloads also require input datasets of images. For the classification models, we used the ILSVRC2012 dataset which contains over 6.4 GB of images. For the object detection model, we used the COCO2017 dataset, which is slightly smaller, about 1.3 GB. Due to copyright licensing limitations, the images are not part of the repository, but instructions are provided to obtain them.

3.2.3 Scoring and accuracy

Performance is generally measured in units of work per second, where units of work may be iterations of a loop, CPU instructions retired, or in the case of MLMark V1.0, image inferences. The timing points inside MLMark occur on the boundaries of the inference operation, after dataset loading, and after retrieving inference results. Dataset loading was excluded from the timing due the many ways an inference engine may load data. Retrieving results was included in the timing loop because in order to assess the inference, the data must be read back to the host system. However, given the amount of time spent on inference, reading back results is a very fast operation compared to loading the dataset. See Figure 2.

MLMark produces two different performance metrics: *throughput* and *latency*. *Throughput* is simply the average inference performance computed by the total number of inferences performed within the sum of all timing windows. Since version 1.0 of MLMark contains image-based inferences, the throughput is the same as frames-per-second.

Latency is measured as the 95th percentile of each inference's timing window, or how long it takes for one inference to complete. This statistical designation means 95% of the time the actual latency will be equal to or better than this reported value. This value often differs from datasheets, which may report the best-case latency.

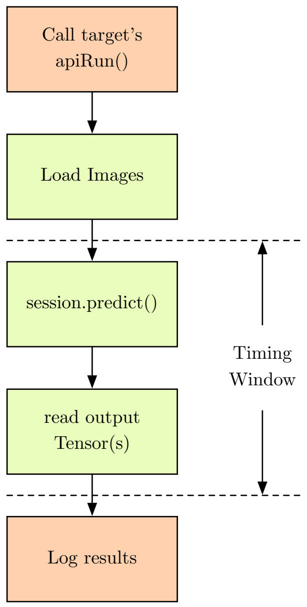


Figure 2. Performance is measured at the start of inference and the end of result retrieval (TensorFlow example).

For the workloads used in MLMark, accuracy is expressed in reference to *ground-truths*, which are a set of human-annotated results for each input datum. For example, during object classification the ground truth for an image would be one (or more) categories associated with that image (dog, tree, car). For object detection, a collection of regions of interest (or ROIs) and their associated categories would make up the ground truth for that image (e.g., the sub-image at rectangle [(10,10),(100,100)] is a dog, with 50% confidence).

The accuracy score depends on the type of inference. For classification, it is defined as Top-1 and Top-5, which indicate if the ground-truth category matched the most confident prediction of the system, or if it was one of the Top-5 predictions. For a large number of inferences, a perfect score would be a Top-1 of 100%, meaning for every inference, the system's top prediction was always the ground-truth.

For object detection, the accuracy measurement is much more sophisticated, since not only must the system classify an object, it must also detect the boundaries of the object in an image. There are many degrees of error here: false object detections, incomplete bounding boxes, wrong classification for a correct bounding box. The method for computing this is known as "mean average-precision", or mAP for short. The details are beyond the scope of this document, but are documented in many papers, see: (Hui, 2018), (Wikipedia, n.d.), and (Shah, 2018). Since there is also a prediction threshold for each detection, MLMark limits the threshold to 30% confidence.

MLMark does not mandate an accuracy threshold. Instead, the run rules state that performance numbers must always be reported *simultaneously* with the accuracy of the inference. In this way, the benchmark explicitly exposes the performance vs. accuracy tradeoff.

4 ML frameworks

Each accelerator manufacturer provides a framework that enables the user to perform an inference on a model. The framework is usually one or more libraries. Every accelerator company provides their own frameworks, with some integrated into Caffe or TensorFlow. All frameworks consist of roughly the same API functions: load a graph, optimize it, perform an inference, and fetch results.

Here are a few examples of frameworks found in the MLMark target area:

- TensorFlow: a self-contained framework created by Google which natively runs on a CPU (and can run on a GPU with extensions)
- TensorFlow Lite (TFLite): a somewhat optimized version of TensorFlow targeting edge devices
- Intel OpenVINO: an expansive API that interfaces with Intel CPUs and other accelerators
- NVIDIA TensorRT: a library that builds on CUDA and cuDNN
- Arm NN: a library built on the Arm Compute Library that provides acceleration for Arm CPUs with NEON, and Arm GPUs with Mali.
- Google TPU: the TPU compiler which compiles a model into a native binary for their TPU accelerator

All of these components in the framework work together to make sure the model's graph is converted to an optimal format for the underlying hardware.

5 Configuration variables that impact performance

Simply loading the model with the framework and providing an input stimulus for inference is half the challenge. There are also a number of variables that may impact performance. MLMark exposes three of these: *precision*, *concurrency*, and *batching*.

5.1 Precision

The baseline models provided by MLMark encode their weights in IEEE 32-bit floating-point format, also known as FP32. Since the majority of the math in CNN inference involves matrix operations, it is theoretically possible to double the efficiency of the network by using 16-bit floating-point, FP16. This may be IEEE FP16 or *bfloat16* (a format invented by Google. Optimizing even further) or an 8-bit integer format, INT8, which has been gaining popularity because some models show little decrease in accuracy when reducing the precision of the weights. Typically, converting an FP32 to an INT8 requires quantization, which means adjusting the weights of the model to accommodate the reduced fidelity. As a result, MLMark provides pre-quantized versions of the models in INT8 format using post-training integer quantization, or PTIQ (as well as the scripts required to do this for oneself).

The tradeoff here seems straightforward: reducing the number of bits in the weight may boost performance but also reduce precision of the predictions. However, as we will see this is not always the case. Further complicating matters, not all frameworks and hardware combinations support all precisions. MLMark allows configuration to support all available formats on the hardware.

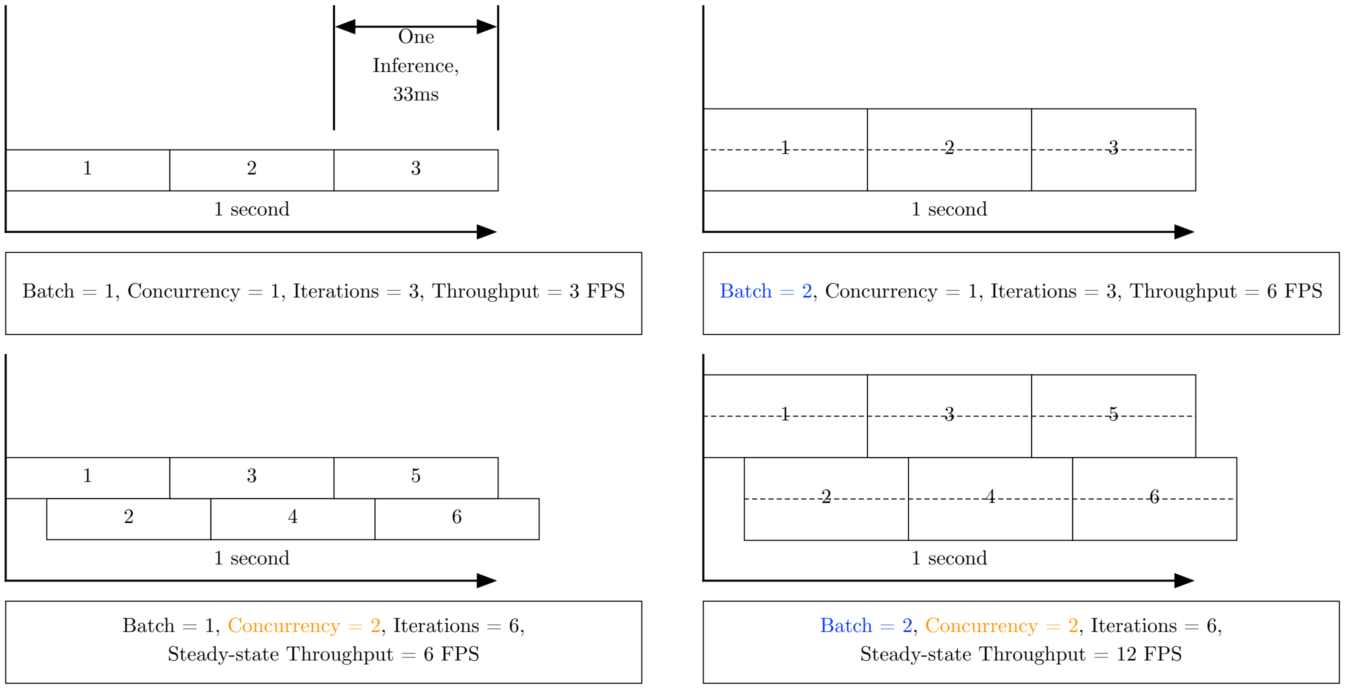

One Inference, 33ms
1
2
3
1 second
Batch = 1, Concurrency = 1, Iterations = 3, Throughput = 3 FPS
1
2
3
1 second
Batch = 2, Concurrency = 1, Iterations = 3, Throughput = 6 FPS
1
3
5
2
4
6
1 second
Batch = 1, Concurrency = 2, Iterations = 6, Steady-state Throughput = 6 FPS
1
3
5
2
4
6
1 second
Batch = 2, Concurrency = 2, Iterations = 6, Steady-state Throughput = 12 FPS

Figure 3. An illustration of the subtle differences between batching and concurrency. When batching, all inputs must be presented to the input of the model at the same time; when running concurrent inferences, each inference may begin once the first stage in the pipeline is available.

5.2 Batching and concurrency

Multiple images may be inferred at the same time either through *batching* or *concurrency*, or in some cases, both. Not all hardware supports these capabilities. Figure 3 illustrates the subtle difference between the two from an execution timing perspective.

Batching refers to performing multiple inferences simultaneous by increasing the dimensionality of the layers' tensors to accommodate more images. For example, an input tensor may be of dimension [1, 32, 32, 24], which would mean one image, 32x32 with 8-bits per-pixel. By expanding this input tensor to [4, 32, 32, 24], each successive operation in the graph performs four simultaneous inferences since the tensors will reshape to accommodate the new data; the mathematic operations remain the same.

Concurrency refers to exploiting the parallel nature of the graph as a pipeline. Unlike batching, concurrency feeds successive inputs to the model after each layer has completed calculation. Concurrency may also refer to having multiple instances of the same model active at one time, as many as the available resources allow. Concurrency is very platform dependent because it requires both APIs in the framework and hardware features to facilitate scheduling.

The framework and hardware determine which method of parallelization are available. In either case, significant gains may be realized by utilizing idle resources. This is where smaller models have an advantage, as more resources may be engaged to increase performance.

6 Methodology

This section describes how the decisions we made affect the constraints on data collection, input format, and measurement.

6.1 Input models file formats

Here we are dealing with a number of different frameworks. Some frameworks are designed to read a particular file format. For example, TensorRT framework has parsers for neural network models in Caffe, UFF and ONNX file formats while Google TPU can read only INT8 TFLite, and only after being compiled for the Edge TPU. We will go through details as follows:

6.1.1 Native TensorFlow PB format

TensorFlow PB refers to "protobuf" or "protocol buffers." This format is from Google which is language and platform-neutral. (Google, n.d.) It is a mechanism for serializing structured data. Both the graph definition and weights are stored together in this file format. Thus, this is a convenient way to store a neural network model.

TensorFlow's model zoo hosts many standard neural network models in this format. We are considering models in this file format as "golden" models. In case a framework cannot read the model in this format, a converter is used to convert the model in PB format into the appropriate format.

6.1.2 UFF file format

UFF stands for "Universal Framework Format," Not to be confused with Universal File Format which is widely used in CAD domain. This is an NVIDIA format "designed to encapsulate trained neural networks so that they can be parsed by TensorRT. It's also designed in a way of storing the information about a neural network that is needed to create an inference engine based on that neural network." (NVIDIA, n.d.)

The V1.0 MLMark release TensorRT target uses an UFF parser. However, per NVIDIA's documentation, this format will be deprecated in the future. (NVIDIA, 2019)

6.1.3 Conversion process : TensorFlow PB to UFF

TensorFlow PB to NVIDIA UFF conversion needs to be done on an x86 machine (laptop or a desktop with NVIDIA GPU). This limitation exists because TensorRT's Python API was not supported on their Jetson platform and the converter script - "convert_to_uff" - needs this Python API. Prerequisites are CUDA, cuDNN and TensorRT. The "convert_to_uff" utility is part of the TensorRT installation.

The following command is sufficient for simpler models like ResNet-50 and MobileNet which have one input layer and one output layer. The user can find the name of last layer using the command:

```
$ convert_to_uff input_file.pb -l
```

This command is also used to convert the UFF file:

```
$ convert_to_uff input_file.pb -o
output_file.uff -O name_of_output_node
```

6.1.4 TFLite file format

TFLite is specially designed for inference on embedded devices. The serialization format used in TFLite is different from TensorFlow. TensorFlow uses "Protocol Buffers" while TFLite makes use of "FlatBuffers."

As stated by Google CodeLabs: "The primary benefit of FlatBuffers comes from the fact that they can be memory-mapped, and used directly from disk without being loaded and parsed. This gives much faster startup times, and gives the operating system the option of loading and unloading the required pages from the model file, instead of killing the app when it is low on memory." (CodeLabs, n.d.)

6.1.5 TensorFlow to TFLite conversion without quantization

TFLite converter is part of the TensorFlow installation. The converter takes the output file name, input model in PB format, and the names of the input and output layers, e.g.:

```
% tflite_convert \
--output_file=foo.tflite \
--graph_def_file=Mobilenetfrozen_graph.pb \
--input_arrays=input \
--output_arrays=\
MobilenetV1/Predictions/Reshape_1
```

The TFLite converter applies many optimizations apart from quantization which improves performance of the model, such as pruning unused graph-nodes, and joining operations into more efficient composite operations.

In practice, it is observed that just the format conversion from PB to TFLite gives significant performance improvement without even quantization. See the results section on TensorFlow versus TFLite.

6.2 Eight-bit integer (INT8) datatypes

6.2.1 Why quantize?

Typically, training happens in FP32 where all of the weights and biases are 32-bit IEEE floats. It is possible to tradeoff some accuracy and reduce model size to one-fourth of its original by quantizing the model.

INT8 quantization is trending in the embedded inference world. The methodology for generating, and the usage of INT8 quantified models vary from framework to framework. For example, TensorRT contains the code to create an optimized INT8 inference engine on the fly, while the Google Edge TPU needs a pre-quantized INT8 model as an input.

6.2.2 Model optimization by framework: TensorRT

TensorRT needs a small image dataset of around 1000 images for INT8 calibration. This calibration image dataset is usually a subset of the validation dataset. Calibration is a one-time process, meaning once the engine has been created it can be reused for subsequent inference engine generation. The output of this process is called a “Calibration Table File” and is generally quite small (few KB).

6.2.3 Post-training full-integer quantization: PTIQ

TFLite now supports converting all model values (weights and activations) to 8-bit integers when converting from TensorFlow to TFLite’s flat-buffer format. This results in four times reduction in model size. It also boosts performance 3-4x on a CPU. Furthermore, this fully quantized network model can be deployed on integer-only hardware accelerators.

The post-training quantization method stores only the weights as 8-bit integer, but this full-integer quantization method statically quantizes all weights and activations. (TensorFlow, n.d.)

6.2.4 TFLite compiled for Edge TPU

Google’s Edge TPU is able to run only INT8 TFLite models. A model also needs to be compiled with edge TPU compiler. The compiler creates a single custom operation (binary) for all Edge TPU compatible ops, until it reaches an unsupported op. The remaining layers stay in their non-binary format and are run on the CPU or host.

The Edge TPU compiler is a command-line tool which requires very little setup, e.g.:

```
$ edgetpu_compiler [options] model.tflite
```

7 Results and observations

After the initial release of MLMark, several dozen scores were collected on available edge hardware. During the coming months we will continue to add new scores and targets to the repository. Note that many new accelerators are not represented due to MLMark's first rule of transparency: the implementation must be published along with the score. The following sections discuss the first round of observations.

Figure 4 combines several results on different hardware, workloads and precision formats into one chart. For each device, all three workloads are presented, color coded by the weight precision. (Note that some of the Arm devices do not have SSDMobileNet scores, this is due to an API limitation at the time of collection.) Due to the extreme range in performance, the data is presented in log format. The units can interchange frames and inferences, since a single inference call to the API requires one image frame as an input, and produces one inference summary (classifications, detections, etc.). For devices that reported scores at more than one batch or concurrency setting, the highest score from the setting was chosen per workload. Raw data can be found at the MLMark website, www.mlmark.org.

It should come as no surprise that there are orders of magnitude difference in this chart. After all, we are comparing quad-core Cortex-A5X/A7X devices to hundred-core dedicated neural accelerators. The purpose of this paper is not to make marketing claims, but to take a snapshot in time of the state of the industry to serve as the leftmost point on a trend graph that will span decades.

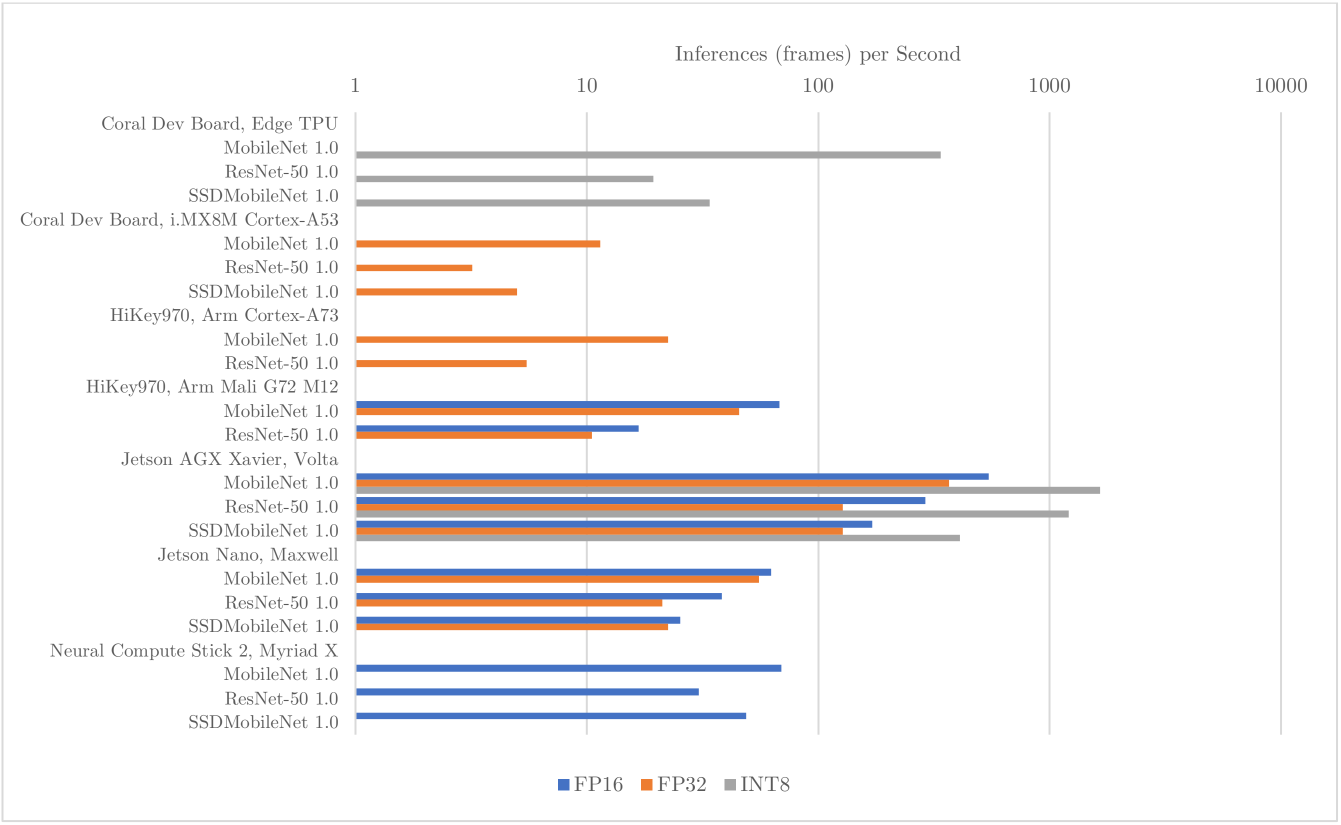


Figure 4. MLMark overall results (as of 2019-11-05); Each cluster of scores was produced by a specific device; three workloads are illustrated, at three different precisions at a single batch & concurrency setting.

7.1 Performance by class

Figure 5 below plots all performance scores against three basic technology classes: CPU, GPU and accelerator. When grouped by technology class, the results show CPU as being the lowest performing, GPU the highest, and accelerators spanning a middle range. Note that the only CPUs used in the results were Arm Cortex A5X/A7X-class.

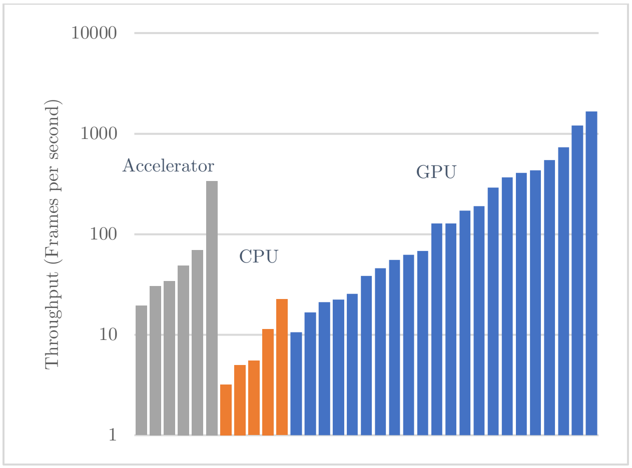


Figure 5. Performance grouped by technology class.

7.2 Performance gains from decreasing precision

One would expect moving from FP32 to FP16 would yield a doubling of performance. This was not observed (Table 1) in the three devices that supported both formats.

	FP16	FP32	% Increase
Jetson Xavier AGX	(fps)	(fps)	
MobileNet 1.0	547	367	33%
ResNet-50 1.0	291	128	56%
SSDMobileNet 1.0	171	128	25%
Jetson Nano			
MobileNet 1.0	62.7	55.3	12%
ResNet-50 1.0	38.4	21.2	45%
SSDMobileNet 1.0	25.3	22.4	11%
HiKey970, Mali G72			
MobileNet 1.0	68.0	45.6	33%
ResNet-50 1.0	16.7	10.5	37%

Table 1. Comparison of FP32 vs. FP16 performance across targets that support both.

7.3 Batching and concurrency

In Figure 6, the batch size was increased on the NVIDIA Jetson Xavier from 1 (a single image) to 32 simultaneous images per call to the inference engine for the MobileNet V1.0 workload. TensorRT also supports concurrent streams, which are plotted against the batch data for similar input sizes. Latency at the 95th percentile is plotted on the secondary vertical axis. While batching shows a significant increase in performance at the start of the curve, ROI begins to decrease rapidly after 16 images. Latency for batching is relatively linear. Concurrency shows no increase in throughput and even worse latency.

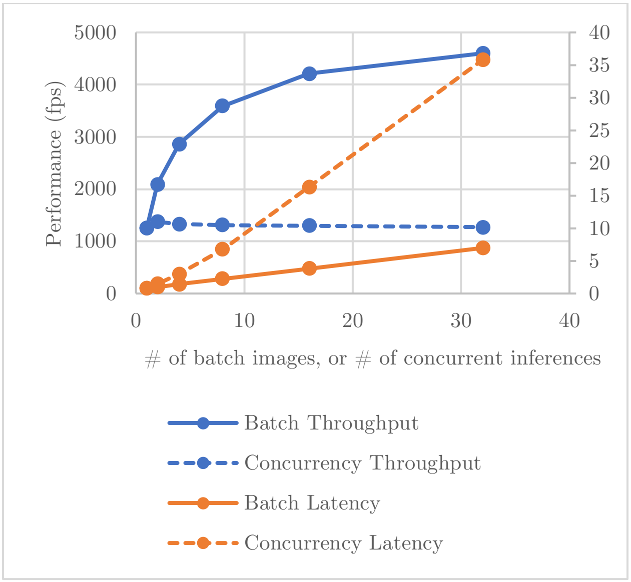


Figure 6. Batching and concurrency using TensorRT on the NVIDIA Xavier AGX platform.

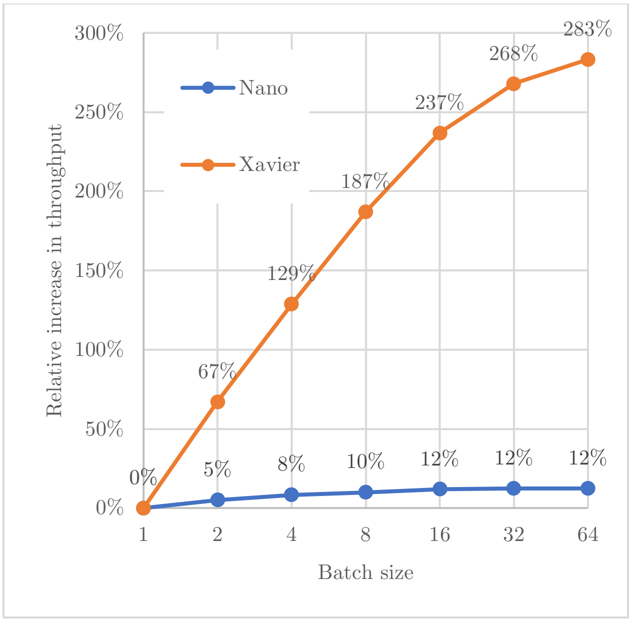


Figure 7. Batch performance depends on available compute resources; Xavier AGX has more available resources than the Nano, hence it obtains better batch performance.

7.4 Precision type impact to accuracy

When using lower precision datatypes for weights, conventional wisdom expects accuracy to decrease by some amount. Figure 8 plots all accuracy values (mAP and Top-1%) against all throughputs for different datatype precision; hardware, model and target are intentionally excluded simply to see larger trends. The results indicate that there is approximately 5% or less variation in accuracy across 32-, 16- and 8-bit datatypes when viewed as precision clusters, and the inverse relationship isn't an inherent property of the models used.

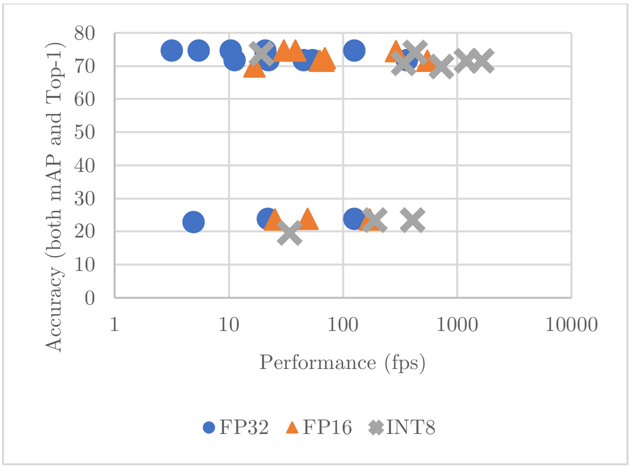


Figure 8. Change in precision versus performance for all hardware, targets, models and configurations.

7.5 Model format optimizations

The Arm NN API provides a single-line change in the code to switch between TensorFlow (protobuf) and TensorFlow Light (FlatBuffer) models. This enabled us to look at the same hardware, same precision, and the same graph, but with a different architectural representation of that graph. A 5-10% improvement can be seen.

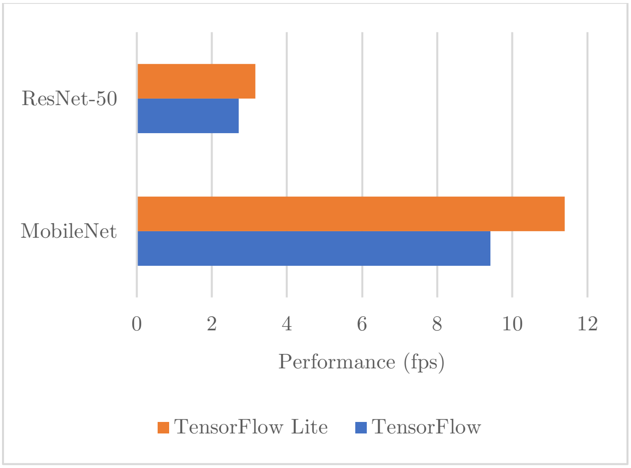


Figure 9. Observing the difference in performance between model formats.

7.6 Economical value

In an attempt to assess cost and performance, we compare the lowest selling price of the hardware platform to their best score on MobileNet (since this generally is the best performance model on all hardware). If we consider a metric of frames-per-second, per dollar spent, the accelerators exceed the only CPU in the study by a large margin. There are three distinct strata in Table 2, low [0,0.1], medium (0.1,0.7], and high [0.7,2.4]. Additional data will be very insightful over time.

Platform & Accelerator	**fps/US$**
Jetson Xavier AGX, Volta	2.4
Coral Dev Board, Edge TPU	2.3
Neural Compute Stick 2, Myriad X	0.7
Jetson Nano, Maxwell	0.6
HiKey970, Arm Mali G72 M12	0.2
HiKey970, Arm Cortex-A73	0.1
Coral Dev Board, i.MX8M Cortex-A53	0.1

Table 2. Performance-per-dollar comparison, where cost is the price of the entire platform, not simply the processor.

7.7 Other unusual observations

Every component in the benchmark chain—except the test-harness—is experiencing a rapid state of revision: some frameworks have monthly releases, and others like TensorFlow have nightly patches. As a result, we often found ourselves re-running experiments with multiple versions of frameworks in an attempt to be as fair as possible. Here are some of the more puzzling instances we observed.

7.7.1 TensorFlow Light lags TensorFlow on x86_64 for the same precision

It is observed that TFLite runs slower than TensorFlow on a x86-64 machine. The following comparison was made on an Intel i3 laptop running Ubuntu 16.04 and TensorFlow 1.15.0rc2:

Workload, Precision	Target Framework	FPS on x86_64
MobileNet, FP32	TensorFlow	31.7
MobileNet, FP32	TFLite	30.2

Following comparison was made on a i3 laptop running ubuntu 16. TF version is 1.15.0rc2. The actual reason is explained in this stack overflow answer: "Existing TensorFlow Lite op kernels are optimized for ARM processor by using NEON instruction set. If SSE is available, it will try to adapt NEON calls to SSE, so it should be still running with some sort of SIMD. Still this code path remains un-optimized." (tehtea, 2019)

7.7.2 FP32 TFLite outperforms INT8 on x86_64 architecture

There is a huge performance gap between FP32 and INT8, in the opposite direction of expectations, when running on x86_64 compared to other architectures. The following comparison was made on an Intel i3 laptop running Ubuntu 16.04 and TensorFlow 1.15.0rc2:

Workload, Precision	**FPS on x86_64**
MobileNet, FP32	30.2
MobileNet, INT8	0.331

This could be related to TFLite optimizations for Arm but not x86, as found in this TensorFlow issue: "[This is likely because quantized INT8 requires an ARM NEON to be faster than float. On a PC float runs better. This is because quantized int relies on special instructions that have not been emphasized on intel x86_64.]" (abhi-rf, 2018)

8 Conclusions

First and foremost, the wide range of performance values measured and the fragile nature of the software during the process is indicative of a nascent industry on the left-hand side of the "innovation-optimization" pendulum. In one case, simply updating software to a minor version release of a mature framework exposed a 40% performance increase. It is not clear that the results in this paper will even be the same in six months *on the very same hardware.*

However, we can conclude that some early assumptions are correct, such as performance gains made from decreasing the precision of the neural-net weights have had only minimal impact on accuracy. The accuracy decrease is not as severe as one would expect, a 5% decrease in accuracy compared to a 75% decrease in data size. Sensitivity to precision still remains a function of other variables besides the hardware, it remains to be seen if these other formats will continue to persist.

We've also seen that neural nets pose an interesting scalability artifact: parallelism is inherent in tensor math and can be accessed through batching simultaneous operations on the same model, however there needs to be enough resources available to see this gain.

In terms of performance per dollar, the small number of CPU and GPU scores in the results don't make a clear case for accelerator dominance, but we can see trends emerging in dollars that favors the accelerators.

9 Future

We will continue to add scores, targets, and models. Already we have observed huge performance increases in the latest versions of drivers which have not been captured in this document. Future studies we would like to pursue include performance mapping to historical data and power analysis. Converting the model performance numbers into TOPS (trillions-of-operations per second) would enable us to map performance on to historical trends. Power consumption was excluded since most of the boards did not have power-plane isolation, meaning overall platform scores would include NIC, USB and other ancillary power unrelated to the accelerator.

10 Thanks

The authors would like to thank the members of EEMBC who helped make the hard decisions required to complete this benchmark, as well as the countless people staffing the GitHub accounts at Tensorflow and ARM for answering our questions. Further, this research would not have been possible without the academics publishing the actual neural-net models.

11 Bibliography

abhi-rf. (2018, August 18). *INT TFLITE very much slower than FLOAT TFLITE #21698*. Retrieved from GitHub: https://github.com/tensorflow/tensorflow/issues/21698

CodeLabs. (n.d.). *TensorFlow for Poets*. Retrieved from codelabs.developers.google.com: https://codelabs.developers.google.com/codelabs/tensorflow-for-poets-2-ios/#2

Google. (n.d.). *Protocol buffers are a language-neutral, platform-neutral extensible mechanism for serializing structured data.* Retrieved from developers.google.com: https://developers.google.com/protocol-buffers

Hui, J. (2018, March 7). *mAP (mean Average Precision) for Object Detection*. Retrieved from medium.com: https://medium.com/@jonathan_hui/map-mean-average-precision-for-object-detection-45c121a31173

NVIDIA. (2019, October). *TensorRT Release Notes v6.01*. Retrieved from docs.nvidia.com: https://docs.nvidia.com/deeplearning/sdk/pdf/TensorRT-Release-Notes.pdf

NVIDIA. (n.d.). *Deep Learning SDK Documentation*. Retrieved from docs.nvidia.com: https://docs.nvidia.com/deeplearning/sdk/tensorrt-archived/tensorrt_301/tensorrt-release-notes/rel_3.html

Shah, T. (2018, January 26). *Measuring Object Detection models — mAP — What is Mean Average Precision?* Retrieved from medium.com: https://towardsdatascience.com/what-is-map-understanding-the-statistic-of-choice-for-comparing-object-detection-models-1ea4f67a9dbd

tehtea. (2019, January 8). *Why is TensorFlow Lite slower than TensorFlow on desktop?* Retrieved from stackoverflow: https://stackoverflow.com/questions/54093424/why-is-tensorflow-lite-slower-than-tensorflow-on-desktop

TensorFlow. (n.d.). *Post-training Quantization*. Retrieved from www.tensorflow.org: https://www.tensorflow.org/lite/performance/post_training_quantization

Wikipedia. (n.d.). *Evaluation measures for information retrieval*. Retrieved from wikipedi.org: https://en.wikipedia.org/wiki/Evaluation_measures_(information_retrieval)