Machine Learning at Facebook: Understanding Inference at the Edge

Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, Tommer Leyvand, Hao Lu, Yang Lu, Lin Qiao, Brandon Reagen, Joe Spisak, Fei Sun, Andrew Tulloch, Peter Vajda, Xiaodong Wang, Yanghan Wang, Bram Wasti, Yiming Wu, Ran Xian, Sungjoo Yoo*, Peizhao Zhang

Facebook, Inc.

ABSTRACT

At Facebook, machine learning provides a wide range of capabilities that drive many aspects of user experience including ranking posts, content understanding, object detection and tracking for augmented and virtual reality, speech and text translations. While machine learning models are currently trained on customized datacenter infrastructure, Facebook is working to bring machine learning inference to the edge. By doing so, user experience is improved with reduced latency (inference time) and becomes less dependent on network connectivity. Furthermore, this also enables many more applications of deep learning with important features only made available at the edge. This paper takes a data-driven approach to present the opportunities and design challenges faced by Facebook in order to enable machine learning inference locally on smartphones and other edge platforms.

1. INTRODUCTION

Machine Learning (ML) is used by most Facebook services. Ranking posts for News Feed, content understanding, object detection and tracking for augmented and virtual reality (VR) platforms, speech recognition, and translations all use ML. These services run both in datacenters and on edge devices. All varieties of machine learning models are being used in the datacenter, from RNNs to decision trees and logistic regression [1]. While all of training runs exclusively in the datacenter, there is an increasing push to transition inference execution, especially deep learning, to the edge. Facebook makes over 90% of its advertising revenue from mobile [2] and has focused on providing its over 2 billion monthly active users the best possible experience [3]. In addition to minimizing users network bandwidth and improving response time, executing inference on the edge makes certain deep learning services possible, for example, Instagram features that involve real-time machine learning at the image capture time. Enabling edge inference requires overcoming many unique technical challenges stemming from the diversity of mobile hardware and software not found in the controlled datacenter environment.

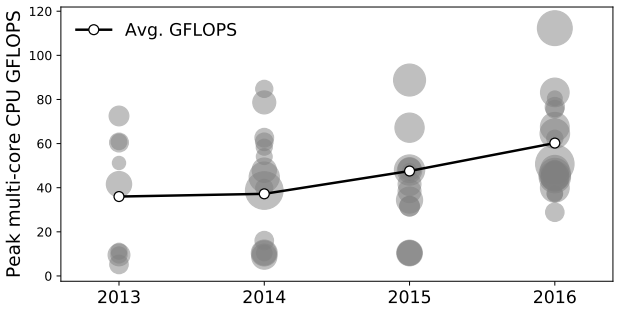


Figure 1: The distribution of peak performance of smartphone SoCs running Facebook mobile app exhibit a wide spread. The data samples represents over 85% of the entire market share and are sorted by the corresponding SoC release year. Peak performance can vary by over an order of magnitude, increasing the design challenge of performance optimization.

While inference is generally less computationally demanding than training, the compute capabilities of edge systems (both hardware and software) running the Facebook app limit what is possible. Figure 1 shows the peak performance of edge devices, representing over 85% of the entire market share, over the corresponding release year of a System on a Chip (SoC). The figure shows a wide variation in performance that must be considered to enable efficient, real-time inference across all edge devices. Trends emerge over time to tell another story: while the average theoretical performance of SoCs is improving over time, there is a consistent, widespread peak performance regardless the release year of the SoCs. To provide the best user experience despite limited performance scaling Facebook has been proactive in developing tools and optimizations to enable all models/services

*Sungjoo Yoo is a Professor at Seoul National University. A large part of this work was performed during his sabbatical leave at Facebook.

to execute across the observed performance spectrum.

Optimizations include techniques for model architecture search, weight compression, quantization, algorithmic complexity reduction, and microarchitecture specific performance tuning. These optimizations enable edge inference to run on mobile CPUs. Only a small fraction of inference currently run on mobile GPUs. This is no small feat considering the computational complexity of state-of-the-art deep learning models and that most CPUs are relatively low-performance. In our dataset, an overwhelming majority of mobile CPUs use in-order ARM Cortex-A53 and Cortex-A7 cores. While a great deal of academic work has focused on demonstrating the potential of co-processors (GPUs/DSPs) and accelerators, as we will show, in the field the potential performance benefits of mobile GPUs vs. CPUs for the Android market are not great. Considering theoretical peak FLOP performance, less than 20% of mobile SoCs have a GPU 3× more powerful than CPUs and, on a median mobile device, GPUs are only as powerful as CPUs. Inference sees limited co-processor use today as a result of close performance between CPU clusters and GPUs as well as an immature programming environment.

In this paper we detail how Facebook runs inference on the edge. We begin by reviewing the hardware and software system stack the Facebook app is run on (Section 2). This highlights the degree of device diversity and divergence of software, presenting many design and optimization challenges. Next, we review the machine learning frameworks and tool sets including PyTorch 1.0 and the execution flow for mobile inference (Section 3). To understand the optimization techniques Facebook has implemented to improve the performance and efficiency of inference we present two case studies. For horizontally integrated devices (e.g., smartphones) we show how general optimizations including quantization and compression can be used across all devices (Section 4). Vertically integrated solutions enable control over the hardware-software stack. In the case of the Oculus virtual reality (VR) platform, we show how inference can easily be ported to run on DSPs to improve energy efficiency, execution time predictability, and performance (Section 5). The degree of performance variance found in inference on the same device is presented in Section 6–this is a problem for applications with real-time constraints. Finally, we conclude by discussing the ramifications of our findings and provide our take on what it means for potential research directions in architecture and systems (Section 7).

This paper makes the following key observations:

- Nearly all mobile inference run on CPUs and most deployed mobile CPU cores are old and low-end. In 2018, only a fourth of smartphones implemented CPU cores designed in 2013 or later. In a median Android device, GPU provides only as much performance as its CPU. Only 11% of the Android smartphones have a GPU that is 3 times more performant than its CPU.

- System diversity makes porting code to co-processors, such as DSPs, challenging. We find it more effective to provide general, algorithmic level optimizations that can target all processing environments. When we have control over the system environment (e.g., Portal [4] or Oculus [5] virtual reality platforms) or when there is little diversity and a mature SW stack (e.g., iPhones), performance acceleration with co-processors becomes more viable.

- The main reason to switch to an accelerator/co-processor is power-efficiency and stability in execution time. Speedup is largely a secondary effect.

- Inference performance variability in the field is much worse than standalone benchmarking results. Variability poses a problem for user-facing applications with real-time constraints. To study these effects, there is a need for system-level performance modeling.

Facebook expects rapid growth across the system stack. This growth will lead to performance and energy efficiency challenges, particularly for ML mobile inference. While today we have focused on optimizing tools and infrastructure for existing platforms we are also exploring new design solutions to enable efficient deep learning inference at the edge.

2. THE LAY OF THE LAND: A LOOK AT SMARTPHONES FACEBOOK RUNS ON

Facebook's neural network engine is deployed on over one billion mobile devices. These devices are comprised of over two thousand unique SoCs[1] running in more than ten thousand smartphones and tablets[2]. In this section we present a survey of the devices that run Facebook services to understand mobile hardware trends.

2.1 There is no standard mobile chipset to optimize for

Figure 2 shows the cumulative distribution function (CDF) of the SoC market share. The data paints a clear picture: there is no "typical" smartphone or mobile SoC. The most commonly-used SoC accounts for less than 4% of all mobile devices. Moreover, the distribution shows an exceptionally long tail: there are only 30 SoCs with more than 1% market share and their joint coverage is only 51% of the market.

In production, smartphone hardware is extremely fragmented. This diversity comes from a combination of the multiple IP blocks in a SoC which may include CPU(s), GPU clusters, shared caches, memory controllers, image

[1]Some of the SoCs are different by connectivity modules.

[2]SoC information is widely accessible through Android system properties and Linux kernel mechanisms, such as /proc/cpuinfo file and sysfs filesystem. Android developers commonly use SoC information to optimize performance. To allow developers to optimize ML-based application performance, we developed cpuinfo library to decode SoC specification and open sourced it at `https://github.com/pytorch/cpuinfo`.

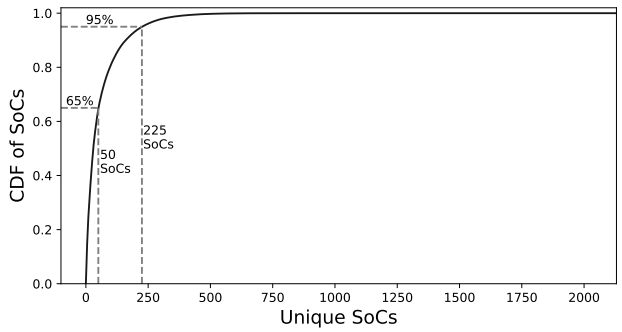


Figure 2: There is no standard mobile SoC to optimize for. The top 50 most common SoCs account for only 65% of the smartphone market.

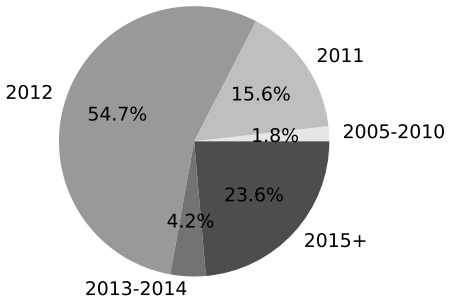


Figure 3: The most commonly-used mobile processors, Cortex A53, are at least six years old. In 2018, only a fourth of smartphones implemented CPU cores designed in 2013 or later.

processors, a digital signal processor (DSP), and even a specialized neural network accelerator (NPU). There are over 25 mobile chipset vendors which each mixes and matches its own custom-designed components with IP blocks licensed from other companies. The fragmentation of hardware is particularly acute on Android, where the Facebook app runs on over two thousand of different SoCs compared to a little more than a dozen SoCs on iOS.

2.2 Mobile CPUs show little diversity

The general availability and programmability of CPUs make them the default option for mobile inference. Thus, we pay close attention to the CPU microarchitecture differences between mobile SoCs. Figure 3 shows a breakdown of the year smartphone CPU cores were designed or released. 72% of primary CPU cores being used in mobile devices today were designed over 6 years ago. Cortex A53 represents more than 48% of the entire mobile processors whereas Cortex A7 represents more than 15% of the mobile processors. When looking at more recent CPUs, the distribution is much more diverse without dominating microarchitectures. The implication of the dominant Cortex A7 and Cortex A53 IPs for machine learning is that most of today's edge inference runs on in-order (superscalar) mobile processors with only one to four cores per cluster. Furthermore, this view of the world poses a new, real challenge for systems and computer architecture researchers – proposed mobile hardware optimizations and accelerators need to consider the long IP lifetime.

We observe a similar multi-core trend as desktop and server chips in mobile. 99.9% of Android devices have multiple cores and 98% have at least 4 cores. We find distinct design strategies between Android and iOS smartphones – iOS devices tend to use fewer, more powerful cores while Android devices tend to have more cores, which are often less powerful. A similar observation was made in 2015 [6]. *To optimize a production application for this degree of hardware diversity, we optimize for the common denominator: the cluster of most performant CPU cores.*

About half of the SoCs have two CPU clusters: a cluster of high-performance cores and another cluster

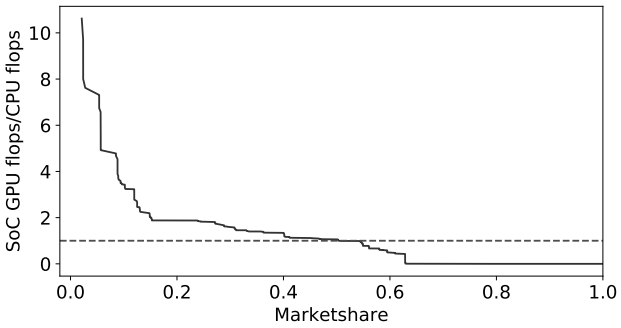


Figure 4: The theoretical peak performance difference between mobile CPUs and GPUs is narrow. In a median Android device, GPU provides only as much performance as its CPU. Only 11% of the smartphones have a GPU that is 3 times more performant than its CPU.

of energy-efficient cores. Only a small fraction include three clusters of cores. Cores in the different clusters may differ in microarchitectures, frequency settings, or cache sizes. A few SoCs even have two clusters consisting of identical cores. In nearly all SoCs, cores within the same cluster have a shared cache, but no cache level is shared between cores in the different clusters. The lack of a shared cache imposes a high synchronization cost between clusters. For this reason, Facebook apps target the high-performing cluster by, for example, matching thread and core count for neural network inference.

2.3 The performance difference between a mobile CPU and GPU/DSP is narrow

High-performance GPUs continue to play an important role in the success of deep learning. It may seem natural that mobile GPUs play a similar part for edge neural network inference. However, today nearly all Android devices run inference on mobile CPUs due to the performance limitations of mobile GPUs as well as programmability and software challenges.

Figure 4 shows the peak performance ratio between CPUs and GPUs across Android SoCs. In a median device, *the GPU provides only as much theoretical GFLOPS*

performance as its CPU. 23% of the SoCs have a GPU at least twice as performant as their CPU, and only 11% have a GPU that is 3 times as powerful than its CPU. This performance distribution is not a historical artifact but a consequence of the market segmentation: mid-end SoCs typically have CPUs that are 10-20% slower compared to their high-end counterparts. The performance distribution corresponds to a wider gap for the GPUs in SoCs targeted for different performance tiers—the performance gap for mobile GPUs is two to four times. Realizable mobile GPUs performance is further bottlenecked by limited memory bandwidth capacities. Unlike high-performance discrete GPUs, no dedicated high-bandwidth memory is available on mobile. Moreover, mobile CPUs and GPUs typically share the same memory controller, competing for the scarce memory bandwidth.

2.4 Available co-processors: DSPs and NPUs

Compute DSPs are domain-specific co-processors well-suited for fixed-point inference. The motivation at Facebook to explore co-processor performance acceleration opportunities is for the increased performance-per-watt efficiency benefit (higher performance with lower power consumption). However DSPs face the same challenge GPUs do – "compute" DSPs are available in only 5% of the Qualcomm-based SoCs the Facebook apps run on. Most DSP do not yet implement vector instructions. While all vendors are adding vector/compute DSPs, it is likely to take many years before we see a large market presence.

The regularity in the computational patterns of many DNN workloads makes NPUs exceptionally amenable to hardware acceleration. Many academic research projects, startups, and companies have proposed solutions in this space (Section 7 offers a thorough treatment). The most notable deployed NPU is the Cambricon 1A in the Kirin 970 SoC [7] and the Neural Engine in the Apple A12 Bionic SoC [8]. While relatively few NPUs exist today, and fewer programmable by third parties, we may be reaching an inflection point.

2.5 Programmability is a primary roadblock for using mobile co-processors

The major APIs used to program neural networks on mobile GPUs are OpenCL, OpenGL ES, and Vulkan on Android and Metal on iOS.

OpenCL was designed to enable general-purpose programs to run on programmable co-processors. Thus, OpenCL does not provide graphics specific functionality, e.g., 3D rendering. Focusing on general-purpose computations helps: OpenCL's API and intrinsic functions as well as support for memory address space management, and efficient thread synchronization make it easier to express computations compared to graphics-oriented APIs like OpenGL. However while most Android devices ship with OpenCL drivers, OpenCL is not officially a part of the Android system, and they do not go through the same conformance tests as OpenGL ES and Vulkan. As shown in Figure 5(a), a notable portion of Android devices ship with a broken OpenCL driver. In the worst case, 1% of the devices crash when the app tries to load the OpenCL library. The instability of OpenCL's library and driver makes it unreliable to use at scale.

OpenGL ES has proved to be a viable alternative. OpenGL ES is a trimmed variant of the OpenGL API specifically for mobile and embedded systems. Being a graphics API, OpenGL ES is not tailored to GPGPU programming. However recent versions of the API provide sufficient capabilities to program neural network computations. Different versions dictate what we can do with mobile GPUs and there are several versions of the OpenGL ES API on the market.

- OpenGL ES 2.0 is the first version of the API with a programmable graphics pipeline. All mobile devices running Facebook apps on Android support this version. With OpenGL ES 2.0 it is possible to implement neural network operators via the render-to-texture technique, but inherent limitations of the API make computations memory bound. All computations have to happen inside a fragment shader which can write only 16 bits[3] of output. Therefore, multi-channel convolution or matrix-matrix multiplication would require reading the same inputs multiple times. The computation patterns are similar to matrix-matrix multiplication on CPU using a dot product function.

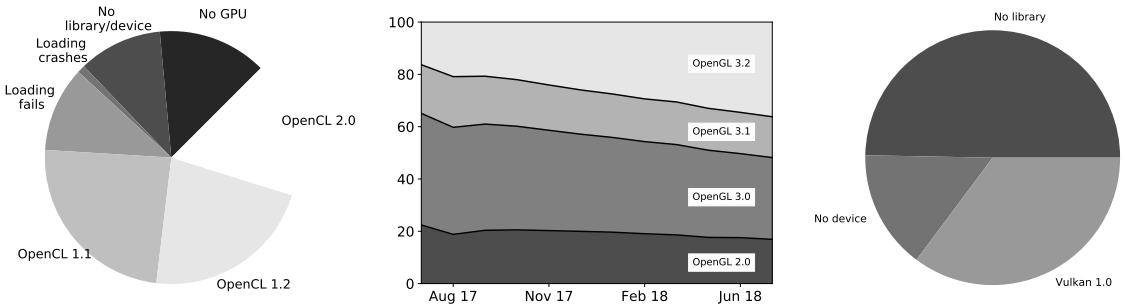
- OpenGL ES 3.0 (or newer) is supported on 83% of Android devices. It is the first version of OpenGL ES that is practical for neural network implementations. Similar to 2.0, all computations need to be implemented in fragment shaders, but OpenGL ES 3.0 supports several features for efficiency. For example, each invocation of a fragment shader can write up to 128 bits of data into each of the (up to 8) textures while also using uniform buffers to load constant data (e.g., weights).

- OpenGL ES 3.1 (or newer) is supported on 52% of the Android devices. It introduces compute shaders that provide similar functionalities available in OpenCL 1.x and early versions of CUDA. For example, important compute features such as, launching kernels on GPU with reduced overhead for the graphics pipeline, fast synchronization within a work-group, access to local memory shared by threads in a work-group, and arbitrary gather and scatter operations with random-access buffers, become available.

Figure 5(b) shows how over the past year the programmability of mobile GPUs on Android devices has steadily improved. Today, a median Android device has the support of GPGPU programming with OpenGL ES 3.1 compute shaders.

Vulkan is a successor to OpenGL and OpenGL ES. It provides similar functionality to OpenGL ES 3.1, but

[3] 32 bits with OES_rgb8_rgba8 extension



(a) OpenCL Support (b) OpenGL ES Support (c) Vulkan Support

Figure 5: Mobile GPUs have fragile usability and poor programmability.

with a new API targeted at minimizing driver overhead. Looking forward, Vulkan is a promising GPGPU API. Today, early adoption of Vulkan (see Figure 5(c)) is limited, being found on less than 36% of Android devices.

Metal is Apple's GPU programming language. Mobile GPUs on iOS devices paint a very different picture. Because Apple chipsets with the A-series mobile processors are vertically-designed, the system stack is more tightly integrated for iPhones. Since 2013 all Apple mobile processors, starting with A7, support Metal. Metal is similar to Vulkan but with much wider market share and more mature system stack support. 95% of the iOS devices support Metal. Moreover the peak performance ratio between the GPU and the CPU is approximately 3 to 4 times, making Metal on iOS devices with GPUs an attractive target for efficient neural network inference. Guided by this data and experimental performance validation, Facebook apps enable GPU-powered neural network inference on iOS for several models.

In summary, Facebook takes a data-driven design approach: the heterogeneity of SoCs makes it inordinately challenging to perform fine-grained, device/SoC-specific optimization. Diverse SoCs pose significant programmability challenge. It is difficult to deploy performance optimization techniques to SoC that are implemented with different versions of device drivers, scopes of memory granularities, and consistency models.

3. MACHINE LEARNING AT FACEBOOK

Facebook puts in significant engineering efforts into developing deep learning features for mobile platforms. Figure 6 illustrates the execution flow of machine learning, where a product leverages a series of inputs to build a parameterized model, which is then used to create a representation or a prediction. Hazelwood et al. presented the major products and services leveraging machine learning that run on Facebook customized datacenter infrastructure [1]. To ensure reliable, high-performance, and high-quality training, this phase generally happens offline in the cloud. On the other hand, an inference phase that makes real-time predictions on pre-trained models runs either in the cloud or on mobile platforms. This paper focuses on mobile inference—making real-time predictions locally at the edge.

3.1 Machine learning models and frameworks

We developed several internal platforms and frameworks to simplify the task of bringing machine learning into Facebook products. As an example, FBLearner is an ML platform that automates many tasks such as training on clusters and is the tool of choice by many teams experimenting and developing custom ML at Facebook. In addition to production tooling, ML development at Facebook has been underpinned by Caffe2 and PyTorch, as set of distinct deep learning frameworks both of which are open source. Caffe2 is optimized for production scale and broad platform support while PyTorch was conceived with flexibility and expressibility in mind allowing researchers to fully express the design space. Caffe2 provides cutting-edge mobile deep learning capabilities across a wide range of devices and is deployed broadly through the Facebook application family. In particular, it is deployed to over one billion devices, of which approximately 75% are Android based, with the remainder running iOS. Caffe2, in particular, is built with optimized mobile inference in mind to deliver the best experience possible for a broad set of mobile devices.

At the 2018 F8 Developer Conference, Facebook announced the road map for a new unified AI framework – PyTorch 1.0 [9]. Pytorch 1.0 combines the production scale of Caffe2 and the research flexibility of PyTorch. It supports the ONNX specification for ecosystem interoperability. With this, Facebook aims to accelerate AI innovation by streamlining the process of transitioning models developed through research exploration into production scale with little transition overhead.

PyTorch 1.0 adopts the ONNX specification for ecosystem interoperability. In addition to being able to export the ONNX model format, PyTorch 1.0 leverages ONNX's Interface for Framework Integration (ONNXIFI) as a stable interface for external backend integration. ONNXIFI enables PyTorch 1.0 to leverage external software libraries and hardware accelerators without requiring redundant integration work for each new backend and also supports embedded edge devices. We are also collaborating with mobile OS ecosystem partners such as Android to include a similar level of functionality, natively within the OS-specific machine learning

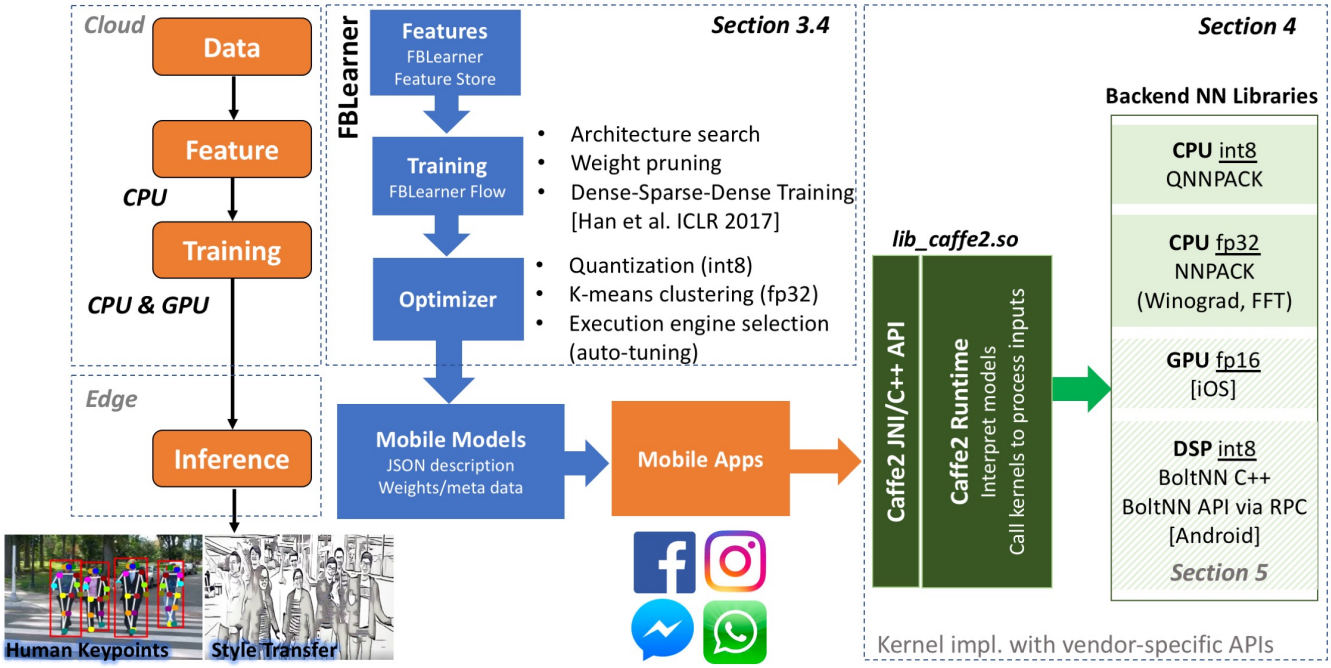


Figure 6: Execution flow of Facebook's machine learning for mobile inference.

accelerator APIs.

3.2 DNN Inference at the Edge

Smartphones are capable of realizing deep learning in real time without relying on the cloud but there are also limitations. While smartphones have improved significantly in computation performance in recent years, these mobile platforms also have various resource constraints, such as power, memory, and compute capabilities. Putting all aspects of SoC components together leads to the landscape of a widely diverse set of SoCs, as presented in Section 2. As a result, mobile presents both an opportunity and, at the same time, a challenge for machine learning. Depending on the application and product domains, Facebook takes different approaches.

Here we review how inference works today by reviewing the mobile workflow. We then provide two case studies that provide details into how we optimize for commodity devices (i.e., mobile SoCs) and vertically integrated, custom solutions (i.e., Oculus VR platforms).

3.3 Important Design Aspects for Mobile Inference and Potential Approaches

To develop mobile applications for the wild west of mobile SoCs, a simple approach is to target application development for the lowest common denominator, in this case, mobile CPUs. This, however, optimizes for ubiquity and programmability while sacrificing efficiency.

To fully harvest potential performance benefits for edge inference, there are several important design tradeoffs that we consider. Iteration speed from an idea to product deployment takes weeks–order of magnitude longer than the deployment cycle for cloud. Performance is key for edge inference; thus, performance optimization is critical for mobile. However, performance characterization and analysis is far more complicated for mobile than cloud because of fragmentation of the mobile ecosystem (Section 2).

Performance is far more limited with wider performance variance for mobile than cloud. Most cloud inference runs on server-class CPUs with theoretical computation capability of up to several TFLOPS. On the other hand, mobile SoCs are orders of magnitude less capable, and deliver between single-digit GFLOPS in the ultra low-end to few hundred of GFLOPS on the very high-end.

Furthermore, model and code sizes are imperative for mobile because of the limited memory capacity of a few GBs. Techniques, such as weight pruning, quantization, and compression, are commonly used to reduce the model size for mobile. Code size is a unique design point for mobile inference. For good deployment experience, the amount of new code pushed into the app needs to be incremental. Several methods are available for application code size management and are potentially viable. First option is to compile applications containing ML models to platform-specific object code using, for example, Glow [10], XLA [11], or TVM [12]. This often leads to larger model sizes (as the model now contains machine codes but enables a smaller interpreter.). Second option is to directly use vendor-specific APIs, such as iOS CoreML [13], from operating system vendors. Another approach is to deploy a generic interpreter, such as Caffe2 or TF/TFLite, that compiles code using optimized backend. The first approach is compiled execution which treats ML models as code whereas the later approach is interpreted execution which treats ML

models as data. Techniques are chosen depending on design tradeoff suitable in different usage scenarios.

3.4 Mobile Inference Workflow

Facebook develops a collection of internal platforms and toolkits to simplify the tasks of leveraging machine learning within its products. FBLearner offers this ecosystem of machine learning tools, enabling workflow execution and management (FBLearner Flow), registry of pointers to data sources, features, and models for training and inference (FBLearner Feature Store), optimal configurations for experiments (FB Learner AutoML), real-time prediction service (FBLearner Predictor), among many others [1].

Figure 6 depicts the execution flow for applying machine learning for edge inference. First, features are collected and selected for any ML modeling tasks from FBLearner Feature Store. The Feature Store is essentially a catalog of feature generators, that is hosted on Facebook's data centers. Then, a workflow describing architectures of a model and steps for the model training and evaluation is built with FBLearner Flow. After model training and evaluation, the next step is to export and publish the model so it can be served in one of Facebook's production inference tiers. Before models are deployed for edge inference, optimization techniques, such as quantization, can be applied in the Optimizer.

In general, to improve model accuracy, three approaches are used iteratively: increasing training data, refining feature sets, and changing model architectures, by e.g. increasing the number of layers or sharing embeddings for features. For performance and memory requirement reasons, we often quantize portions of models. One example is to reduce the precision of a large multi-GB embedding table from 32-bit single precision float to 8-bit integers. This process takes place after we verify that there is little or no measurable impact to model accuracy. Then, for edge inference, to improve computational performance while maximizing efficiency, techniques, such as quantization, k-means clustering, execution engine selection, are employed to create mobile-specific models. Once the model is deployed to a mobile platform, Caffe2 Runtime interprets models and call kernels to process inputs. Depending on the hardware architecture and the system stack support, backend neural network libraries are used by Caffe2 Runtime for additional optimization.

4. HORIZONTAL INTEGRATION: MAKING INFERENCE ON SMARTPHONES

Mobile inference is primarily used for image and video processing. Therefore, inference speed is typically measured as the number of inference runs per second. Another commonly-used metric is inference time, particularly for latency sensitive applications. To exploit performance optimization opportunities before models are deployed onto mobile platforms and to ensure fast model transmission to the edge, Caffe2 implements specific features, such as compact image representations and weight [14, 15], channel pruning [16], and quantization. In addition, tuning of spatial resolution that controls the processing time of middle layers is particularly useful for mobile. We also apply commonly-used techniques, such as pruning and quantization, to aggressively cut down the size of DNN models while maintaining reasonable quality [17].

In addition, to maintain certain performance levels for good user experience, quantization is used for edge inference. The use of quantization is a standard industry practice with support in e.g., Google's GEMMLOWP [18] and Qualcomm's neural processing SDK [19]. A floating point tensor is linearly quantized into 8 or fewer bits and all nodes in the data flow graph operate on this quantized tensor value. To efficiently quantize node outputs, we need to precompute good quantization parameters prior to inference time. There are two approaches here. One is to modify the graph at training time to learn the quantization directly–quantization-aware training [20]. The other is to add a stage after training to compute appropriate quantizers–post-training quantization. More advanced quantization techniques at the execution front-end is under investigation [21, 22, 23].

To make the best possible use of limited computing resources, Caffe2 Runtime integrates two in-house libraries, NNPACK [24] and QNNPACK [25], which provide optimized implementation of convolution and other important CNN operations, and contain platform-specific optimizations tailored for mobile CPUs.

NNPACK (Neural Networks PACKage) performs computations in 32-bit floating-point precision and NCHW layout, and targets high-intensity convolutional neural networks, which use convolutional operators with large kernels, such as 3x3 or 5x5. NNPACK implements asymptotically fast convolution algorithms, based on either Winograd transform or Fast Fourier transform, which employ algorithmic optimization to lower computational complexity of convolutions with large kernels by several times. With algorithmic advantage and low-level microarchitecture-specific optimizations, NNPACK often delivers higher performance for direct convolution implementation.

QNNPACK (Quantized NNPACK) on the other hand performs computations in 8-bit fixed-point precision and NHWC layout. It is designed to augment NNPACK for low-intensity convolutional networks, e.g. neural networks with large share of 1x1, grouped, depthwise, or dilated convolutions. These types of convolutions do not benefit from fast convolution algorithms, thus QNNPACK provides a highly efficient implementation of direct convolution algorithm. Implementation in QNNPACK eliminates the overhead of im2col transformation and other memory layout transformations typical for matrix-matrix multiplication libraries. Over a variety of smartphones, QNNPACK outperforms state-of-the-art implementations by approximately an average of two times.

The choice of two mobile CPU backends help Caffe2 Runtime deliver good performance across a variety of mobile devices and production use-cases. In the next

section we present the relative performance comparison of three state-of-the-art DNN models running on QNNPACK with quantization compared to running on NNPACK in floating-point representation.

4.1 Performance optimization versus accuracy tradeoff

The primary performance benefits with reduced precision computation come from–(1) reduced memory footprint for storage of activations, (2) higher computation efficiency, and (3) improved performance for bandwidth bounded operators, such as depthwise convolutions and relatively small convolutions. Reduced precision computation is beneficial for advanced model architectures. This inference time speedup is, however, not received equally well when the technique is applied directly onto all models.

We compare the inference time speedup of the reduced precision computation with 8-bit fixed-point over the baseline FP32 implementation (under acceptable accuracy tradeoff). First, the UNet-based Person Segmentation model [26] that relies on 3x3 convolutions with relatively small spatial extent experiences performance regression in the quantized version. This regression is caused by inability to leverage NNPACK's highly optimized Winograd-based convolution for both the low- and the high-end Android smartphones. Furthermore, for the quantized models, additional instructions are needed to extend elements from 8 to 16 bits for computation[4], leading to additional performance overhead compared to the FP32 version, which can immediately use loaded elements in multiply-add operations.

For style transfer models, a network with a relatively small number of channels and large spatial resolution is used with 3x3 convolutions. We start seeing much better performance response to QNNPACK-powered reduced precision computation. The efficiency reduction from losing Winograd is compensated by reduced memory bandwidth for these large spatial domain convolution.

Finally, when we look at a custom architecture derived from ShuffleNet [27], which leverages grouped 1x1 convolutions and depthwise 3x3 convolutions for the bulk of the model computation, we see substantial inference performance improvement from reduced memory bandwidth consumption for the depthwise convolutions. Reduced precision computation on QNNPACK improves inference performance for the depthwise-separable models that are increasingly popular in mobile and embedded computer vision applications.

However, in order to maximize performance benefit, we have to consider both algorithmic and quantization optimization. Currently, using algorithmic optimization with e.g. Winograd algorithm for CNNs can disallow quantization. Therefore, if the benefit from Winograd transformation is greater than that of quantization, we see a relative slowdown for quantized models, calling for a better use of quantization that offers more consistent performance improvement.

[4]This inefficiency is not inherent to 8-bit fixed-point convolutions, but is caused by restrictions of the NEON instruction set.

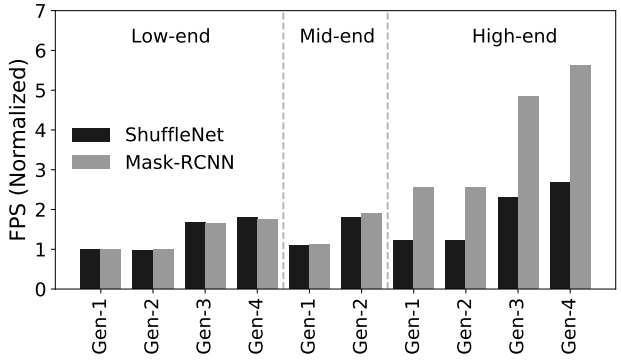


Figure 7: Performance comparison over several generations of low-end, mid-end, and high-end smartphones for two image-based DNN models, ShuffleNet [27] and Mask-RCNN [28]. The smartphone performance tier does not always correspond to inference performance. The performance of DNN models respond to hardware resources differently.

While this section represents the way designs are currently done today, as more hardware and better (co-)processors (e.g., CPUs/GPUs/DSPs) make their way onto mobile devices we will take advantage of the additional computation performance by using more accurate models. Next we present two approaches for enabling mobile inference at Facebook. First is the horizontal integration that enables Facebook Apps to run efficiently across a variety of mobile platforms. Second is the vertical integration for the VR application domain.

4.2 An augmented reality example

Smart cameras are designed to add intelligence to cameras, i.e., processing images directly on an edge device. This feature improves user experience by reducing latency and bandwidth usage. In addition to image classification models, advanced machine learning techniques are applied to estimate and improve the quality of photos and videos for various Facebook services, to understand digital contents locally, directly on smartphones.

The challenges with smart cameras are large computational demands for delivering real-time inference. To enable smart cameras on mobile platforms running the Facebook App, we train mobile specific models, compress weights for transmission, and execute quantized models. We design a collection of image classification models tailored for smartphones. To lessen the transmission cost, models can be compressed using a Deep Compression-like pipeline. As previously discussed, quantization is also considered. Finally, to improve inference performance, some of the models are processed using an 8-bit fixed point datatype for the weights. Additionally, models shipped with the k-means quantization method typically use 5 or 6 bits for the weights.

4.3 DNN model performance across a wide spectrum of smartphones

Figure 7 illustrates the performance of two important Facebook DNN models, i.e., classification and human pose estimation, across multiple generations of smartphones in the different performance tiers. The x-axis represents multiple generations of smartphones in the low-end, mid-end, and high-end performance tiers whereas the y-axis plots the normalized inference time speedup over the first smartphone generation in the low-end tier. First, we observe that the performance tier does not always directly correspond to inference performance. The newest generation of smartphones in the low-end tier offer competitive inference performance as that in the mid-end tier for both DNN models.

Furthermore, the performance of DNN models respond to different degree of hardware resources differently. For example, the DNN model used for human bounding box and keypoint detection (Mask-RCNN [28]) demands much higher compute and memory resource capacities. Thus, when comparing the inference time speedup between the latest generation of smartphones between the low-end and high-end performance tiers, we see a much higher performance speedup for smartphones equipped with more abundant resources–5.62 times speedup for Gen-4/High-End over 1.78 times speedup for Gen-4/Low-End. Although we still see higher inference time speedup in the latest generation of high-end smartphones, the speedup is less pronounced for the DNN model used for classification (ShuffleNet [27]).

5. VERTICAL INTEGRATION: PROCESSING VR INFERENCE FOR OCULUS

This model-specific inference time comparison projects the performance of realistic DNN models onto the diverse spectrum of smartphone platforms described in Section 2. In addition to the previously-shown peak performance analysis for the deployed smartphones in the wild, Figure 7 shows how different generations of smartphones across the different performance tiers react to two realistic DNN models. It is important to continue strong performance scaling for the remaining smartphones in the entire market for higher product penetration.

Oculus platforms create new forms of interactions by running multiple DNNs for tasks including hand, face, and body tracking. To provide a high-quality user experience the models must run at a steady frame rate of 30 to 60 FPS, three times greater than mobile image and video performance requirements. Moreover, headsets include multiple cameras to cover a wide field of view. This puts the performance requirements of VR platforms on the order of many hundreds of inference per second. This presents a significant challenge for embedded devices as all processing has to be performed on-device with high performance delivery. To overcome the particular design challenges in the VR design space, for mobile inference, Facebook takes a vertical design approach. In the case of Oculus, we explore and assess

DNN features	DNN models	MACs	Weights
Hand Tracking	U-Net [29]	10x	1x
Image Model-1	GoogLeNet [30]	100x	1x
Image Model-2	ShuffleNet [27]	10x	2x
Pose Estimation	Mask-RCNN [28]	100x	4x
Segmentation	TCN [31]	1x	1.5x

Table 1: DNN-powered features for Oculus.

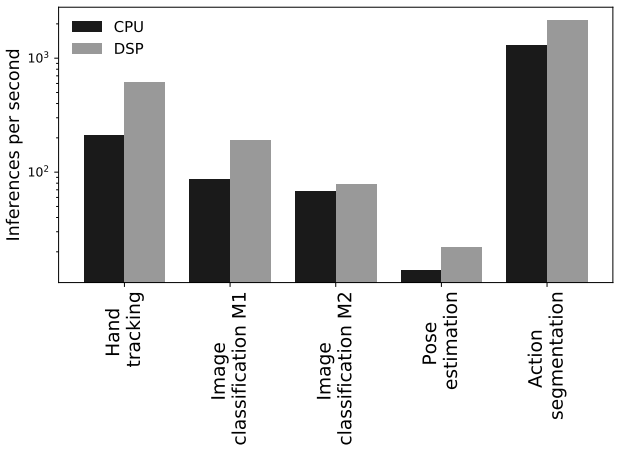


Figure 8: Inference time performance comparison between CPU and DSP.

the suitability of Qualcomm Hexagon DSPs for DNN models by offloading the most computationally demanding DNNs and compare performance across used models running on DSPs and CPUs.

5.1 DNN models and hardware used for VR platforms

The Oculus VR platform explores many state-of-the-art DNN models. Models are programmed in PyTorch 1.0 and the weights are quantized with PyTorch 1.0's int8 feature for mobile inference. Table 1 shows some of the key DNN models explored by Oculus: Hand Tracking, Image Classification Model-1, Image Classification Model-2, Pose Estimation, and Action Segmentation.

For mobile inference, the DNN models are offloaded using PyTorch 1.0's CPU and Facebook's BoltNN DSP inference backends. The CPU model utilizes a big.LITTLE core cluster with 4 Cortex-A73 and 4 Cortex-A53 and a Hexagon 620 DSP. All CPU cores are set to the maximum performance level. The four high-performance CPU cores are used by the DNN models. The DSP shares the same memory space with the mobile CPU cores and has a separate layer of caches, making it convenient to program but also isolated enough to prevent cache thrashing for other concurrent processes running on the mobile CPU. As we will later see in Section 6, dependable, stable execution is an important feature to have to guarantee user experience.

5.2 DSP evaluation results and analysis

Figure 8 compares the FPS of the DSP and CPUs for all models. DSP clearly outperforms CPU for all

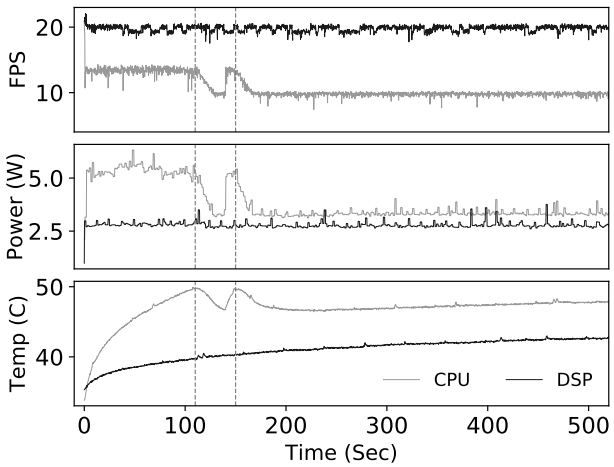


Figure 9: Inference frames-per-second performance, power, and temperature comparison for a vision model for Oculus. Thermal throttling (marked with dotted lines) prevents CPU from operating at an acceptable FPS performance level.

the models with various complexity and architectures, achieving an average speedup of 1.91x, ranging from 1.17 to 2.90 times. The highest speedup comes from models with simple convolution operations, such as in the Hand Tracking and the Image Classification Models.

When intensive memory-bound operations are involved, such as depth-wise convolutions in the image classification and pose estimation models, the speedup of DSP over CPU becomes less pronounced. This is because the memory load-store operations are at the granularity of the vector width or coarser, e.g., more than 128B in Hexagon DSPs. Thus, additional memory transformation is needed, introducing extra performance overhead. Furthermore, for memory-bound layers, such as grouped convolutions or depth-wise convolutions, extra computations are required to optimize the memory layout of activations and filters, in order to fully take advantage of the SIMD units. Finally, across all models, additional system overhead can come from remote procedure calls that flush the L2 cache on the chipset.

In addition to using the amount of performance speedup to determine where a DNN model should be executed, designs for AR/VR wearables must consume as little power consumption as possible for prolonged battery life and for ergonomic requirement of platform temperature. We compare the performance, power, and temperature of the post estimation model running on the CPU versus the DSP. Figure 9 shows that the CPU implementation consumes twice as much power as that of the DSP in the beginning. Then, thermal throttling kicks in so the power consumption of the CPU implementation drops while still using 18% more power than the DSP. The thermal throttling has a significant effect on performance, degrading the FPS performance to 10 frames-per-second. For lower platform power consumption and operating temperature, Facebook takes the vertical-designed approach to offload DNN models using the BoltNN DSP backend for its VR platforms.

Despite higher performance, lower power consumption and operating temperature, the DSP implementation comes with significantly higher programming overhead. First, because most DSP architectures support fixed-point data types and operations, DNN models need to be quantized. Depending on the application domain and models, this may cause substantial accuracy loss. It also requires developers to port model operators to fixed-point implementation; otherwise, this can easily become the performance bottleneck for light-weight operations. Furthermore, developers must pay additional attention to optimize memory layout; otherwise the memory hierarchy can become a contentious resource, leading to additional delay.

Last but not least, an important, yet less explored and understood factor to determine where models should be run at–CPU versus accelerators–is *inference time variation*. Even if one can hand optimize CPU implementation such that the inference time meets the application-specific performance target, and the power and temperature results are competitive, offloading models to accelerators may still be more desirable, despite the higher programming overhead. We next introduce the role of performance variability for mobile inference.

6. MAKING INFERENCE IN THE WILD: PRACTICAL CONSIDERATIONS FOR PROCESSING ON MOBILE DEVICES

Performance variability is a practical concern for Facebook because it is challenging to make guarantees on quality of service. Real time constraints and model accuracy are often competing objectives: higher-quality models take longer to process but provide more accuracy. For example, we might conservatively use a smaller, less computationally expensive model to meet a 95% performance target across all devices and all App instances. However, if we had a better way to model and predict performance variability we could put tighter bounds and could use different models tuned to maximize accuracy while meeting real-time performance/FPS metrics to provide the best user experience (FPS) and service (model accuracy). In this section we show how much performance can vary and suggest a simple way to model it.

6.1 Performance variability observed in the production environment

To arrive at an optimal design point, we perform rigorous evaluations for Facebook services and use the performance characterization results to drive better solutions. A key observation we derive from the performance data is that *mobile inference performance exhibits significant variability, even across the same device running the same software stack*. Figure 10 shows the inference time performance of the most time-consuming convolutional neural network layer of a key model across several generations of iPhone SoCs (x-axis). As expected, we see that the inference time (y-axis) is the

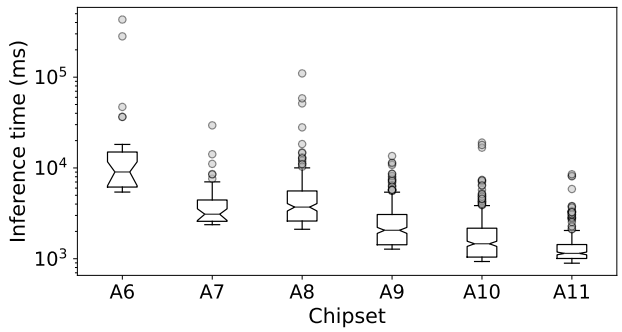


Figure 10: The inference time performance improves over generations of iPhones. However, within each generation, significant inference performance variability is observed with a large number of outliers.

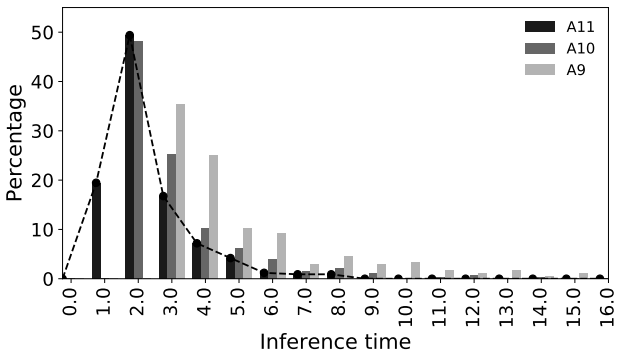


Figure 11: The inference time follows an approximate Gaussian distribution with the mean centered at 2.02ms and the standard deviation of 1.92ms.

lowest for the most recent generation of iPhones using Apple A-11 processors. Somewhat less intuitive is the observed wide performance variability of inference performance – even within the same generations of the SoCs.

We examine the inference performance results internally using our small-scale smartphone benchmarking lab. While we see a general trend of performance variability across key machine learning models, the degree of performance variability is much less pronounced, usually less than 5%. This presents a challenge as ideally we would benchmark new models under the exact conditions we expect the models to run. From our observations this undertaking seems impractical as it would require a fleet of devices.

The much higher performance variability in the production environment is likely due to higher system activities in deployed smartphones and the environment the smartphones are in (e.g., the ambient temperature or the number of Apps a user allows to run concurrently). Concurrent processes or background activities cause resource contention, leading to performance perturbation [32]. Furthermore, the performance of mobile processors is not only limited by processor junction temperature but also smartphone surface temperature for ergonomic requirements [33, 34]. This means that, depending on how and where smartphones are used, the likelihood of thermal throttling is potentially much higher in the production environment, representing more realistic usage scenarios. Finally, process variation and battery aging also contribute to performance variability. *To have representative performance results and analysis, it is important to perform in-field studies for machine learning designs and performance evaluation in the mobile space.* Furthermore, this also means that *for optimization techniques to deliver robust performance impact, the techniques must be able to endure the performance variability in the field.*

6.2 Do the performance variability characteristics follow certain trends or statistical distributions?

It is clear that inference performance on smartphones is non-deterministic and follows a wide distribution. This is highly undesirable as the non-deterministic inference time translates directly into varied quality of user experience. If we were able to model and predict variability, we could optimize designs by, for example, customizing networks to best suit different mobile platforms and users depending on situations. In-field functionality and performance evaluation is an important part of our iterative model fine-tuning process.

Figure 11 illustrates the histogram for the inference time of the key machine learning layer across three different generations of iPhone mobile SoCs. In particular, the inference time for A11 follows an approximate Gaussian distribution with the mean centered at 2.02ms and the standard deviation of 1.92ms. A recent work by Gaudette et al. [35] shares similar observations for mobile applications in general and proposes modeling techniques to predict the non-determinism in performance with general forms of Gaussian. The follow-on work [36] takes a data-driven approach with the use of arbitrary polynomial chaos expansions which approximates stochastic systems by a set of orthogonal polynomial bases, without any assumption of workload/system statistical distribution. With the ability to model performance variability, a certain level of inference performance can be guaranteed, leading to overall better quality of user experience.

In summary, the significant performance variability observed for mobile inference introduces varied user experience. If taking a classic approach to modeling and evaluating ML model performance and energy efficiency with an *average* value of experimental runs, designers risk the chance for delivering the required level of performance quality. Thus, particularly for mobile inference benchmarking, it is critical to describe how severe performance variability is for a design. One option is to represent evaluation results (for e.g., inference time performance) with the information of average, maximum, minimum, and standard deviation of experimental measurement values. Furthermore, our observation here also pinpoints the importance of in-field studies for machine learning designs.

7. DISCUSSION AND FUTURE DIRECTIONS

This section discusses the implications from the results shown in this paper which influence the important design decisions within Facebook. We also highlight the research directions for the years to come.

The majority of mobile inference run on CPUs. Given all the engineering efforts put into accelerating DNN inference with co-processors and accelerators, it is somewhat counterintuitive that inference on Android devices are processed on mobile CPUs. The reality is that it is currently too challenging to maintain code bases optimized to perform well across the wide range of Android devices (Section 2). Moreover, as illustrated in Figure 1, even if we did port all inference to run on co-processors, the performance gains would not be substantial enough to justify the implementation effort.

Most inference run on CPUs that are at least six years old. Future facing research is important, but the reality is that having large-scale, global impact on smartphones may be further off than what we think. As presented in Section 2, most inference are made on processors released in 2011 and 2012, respectively. This isn't just a case of old smartphones that are still being out there or being left on. A major portion of these smartphones are sold in the recent one to two years. To provide the same experience to all Facebook users, substantial software optimization efforts are targeted in optimizing inference for these CPUs—ones that represent the largest market share.

The performance difference between a mobile CPU and GPU/DSP is not 100×. Given the performance gap between server CPUs and GPUs is usually 60-100×, one might suspect that a similar trend is found on the mobile side. However, this is not the case. Mobile GPUs, and even DSPs, are less than 9 × faster than mobile CPUs. Similar finding is found in [6]. This is largely because mobile GPUs were not designed to process the same class of high-resolution graphics rendering that discrete GPUs are. Mobile GPUs help offload image processing in a *relatively* low-end environment. DSPs are slightly more promising–mobile inference are slowly transitioning to execute on DSPs. Furthermore, many mobile CPUs come with a decently provisioned SIMD unit, which when properly programmed provides sufficient performance for vision-based inference.

Programmability is a primary roadblock to using mobile co-processors/accelerators. As seen in Section 2, one of the main challenges with using GPUs and DSPs is programmability. For Android smartphones, OpenCL is not reliable enough for a business at the scale of Facebook . DSPs have more robust software stacks. However, porting code still takes a long time as the implementation must be signed and whitelisted by DSP vendors. The story for Apple devices is better, partially because there is so much less variety of devices and software. Metal also plays a large role as it is relatively straightforward to use. Therefore, many iPhone inference are run on mobile GPUs. With the introduction of Vulkan and DSP engineering efforts, inference are making their way into co-processors. Looking forward, more research and engineering effort put into making existing mobile GPU and DSP hardware more amenable to processing DNN inference has a high impact to ML adoption at the edge.

Co-processors and accelerators are used for power and stable performance; speedup is often secondary. The main reason mobile inference are ported to a co-processor is for improved efficiency and dependable, predictable execution time. While there are applications that require specialized hardware for performance, we suspect this finding is not a Facebook or DNN-specific phenomenon. Because our main focus is end-user usability, unless the performance gain is significant (e.g., 50x) and achieved using better tools and infrastructure, it is unlikely most of these accelerators will actually be utilized when found on mobile devices.

Accuracy is a priority, but it must come with a reasonable model size. The accuracy of a DNN model can be tied directly to user experience [1]. It is also generally true that larger models result in higher accuracy. When it comes to mobile, it is important to maximize accuracy while keeping model sizes reasonable. Facebook focuses on model architecture optimization to identify highly-accurate models while minimizing the number of parameters and MACs. Trained models are then further refined for efficiency with aggressive quantization and weight/channel pruning. Looking forward, methods to improve architecture search, including techniques, such as BayesOpt [37, 38], AutoML [39] and [40], are of important interest.

There is also a big push for generally applicable optimization techniques. Recent work on hardware for machine learning and efficient training and inference has substantially advanced the state-of-the-art [41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66]. Many of the ideas being published in the top architecture conferences are custom hardware specific and not applicable to general-purpose existing SoCs.

Performance variability is a real problem. An often overlooked issue in the architecture research community is performance variability, which results in a serious concern for real-time and user-facing systems. Optimizing designs for the average case risks user experience for a large portion of the market share while targeting designs for all mobile devices in the market implies conservative design decisions. It is important to present performance results taking into account performance distribution, particularly for mobile inference.

Researchers need to consider full-picture and system effects. Accelerating mobile inference on the edge is an important task. However, DNNs brings design challenges across the entire computing ecosystem. For example, many are looking at weight and activity sparsity to improve inference execution. However, many models are large and many users do not have access to high-speed/reliable connections. Compression can provide us with new abilities for model deployment. Generally speaking, in addition to new accelerator designs, it is important to have more holistic system solutions.

The absolute numbers make it look like everything runs on CPUs and life is fine. However, if you look at the trend—many more devices are increasingly supporting NPUs and programmable interfaces such as NN API. While it may not be intuitive to read that all mobile inference run on CPUs even when GPUs and DSPs are readily available, minimizing power consumption is still a major optimization goal which can only be addressed with specialized hardware. Having a wide spectrum of mobile chipsets means that it may be easier to get your hardware accelerator into one of the SoCs. However, the performance impact of accelerators prevails only when programmability and the entire system stack are intact.

8. CONCLUSION

The increasing importance of deep learning-based applications brings many exciting opportunities yet diverse and complicated design challenges at the edge. This paper presents the state of the industrial practices for realizing machine learning inference at the edge. We take a data-driven approach by first showing the important trends of hardware heterogeneity in the smartphone space and the maturity of the software stack. This leads to important design decisions for deep learning application development at scale. On the other hand, enabling inference for systems such as virtual reality platforms, we take a vertical integration approach. By doing so, performance acceleration with co-processors is far more realistic, leading to faster inference time, lower power consumption, and more importantly reduced performance variability. While this paper demonstrates the possibilities of deep learning inference at the edge, we pinpoint the importance of in-field studies in order to capture the performance variability effect in the production environment. We hope the observations, insights, and our design principles for different edge platforms provided in this paper can help guide our community to better design and evaluate deep learning inference at the edge.

9. REFERENCES

[1] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang, "Applied machine learning at Facebook: A datacenter infrastructure perspective," in *Proc. of the Intl. Symp. on High Performance Computer Architecture*, 2018.

[2] Facebook Investor Relations, "Facebook reports first quarter 2018 results," 2018. https://investor.fb.com/investor-news/press-release-details/2018/Facebook-Reports-First-Quarter-2018-Results/default.aspx.

[3] Forbes, "Mark Zuckerberg: 2 billion users means Facebook's 'responsibility is expanding'," 2017. https://www.forbes.com/sites/kathleenchaykowski/2017/06/27/facebook-officially-hits-2-billion-users/#e209ae937080.

[4] Portal. https://portal.facebook.com/.

[5] Oculus. https://www.oculus.com/.

[6] C. Gao, A. Gutierrez, M. Rajan, R. G. Dreslinski, T. Mudge, and C. J. Wu, "A study of mobile device utilization," in *Proc. of the IEEE Intl. Symp. on Performance Analysis of Systems and Software*, 2015.

[7] AnandTech, "Cambricon, Makers of Huawei's Kirin NPU IP, Build A Big AI Chip and PCIe Card." https://www.anandtech.com/show/12815/cambricon-makers-of-huaweis-kirin-npu-ip-build-a-big-ai-chip-and-pcie-card.

[8] Apple A12 Bionic. https://www.apple.com/iphone-xs/a12-bionic/.

[9] Facebook, "Caffe2 and pytorch join forces to create a research + production platform pytorch 1.0," 2018. https://caffe2.ai/blog/2018/05/02/Caffe2_PyTorch_1_0.html.

[10] N. Rotem, J. Fix, S. Abdulrasool, S. Deng, J. H. Roman Dzhabarov, R. Levenstein, B. Maher, S. Nadathur, J. Olesen, J. Park, A. Rakhov, and M. Smelyanskiy, "Glow: Graph lowering compiler techniques for neural networks," 2018. https://arxiv.org/abs/1805.00907.

[11] Google, "XLA is a compiler that optimizes TensorFlow computations." https://www.tensorflow.org/performance/xla/.

[12] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: An automated end-to-end optimizing compiler for deep learning," 2018. https://arxiv.org/abs/1802.04799.

[13] Apple Core ML, "Core ML: Integrate machine learning models into your app." https://developer.apple.com/documentation/coreml?changes=_8.

[14] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," *CoRR*, vol. abs/1510.00149, 2015. http://arxiv.org/abs/1510.00149.

[15] Y. LeCun, J. S. Denker, and S. A. Solla, "Optimal brain damage," in *Advances in Neural Information Processing Systems 2* (D. S. Touretzky, ed.), pp. 598–605, Morgan-Kaufmann, 1990.

[16] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks," *CoRR*, vol. abs/1707.06168, 2017.

[17] Facebook Research, "Enabling full body AR with Mask R-CNN2Go," 2018. https://research.fb.com/enabling-full-body-ar-with-mask-r-cnn2go/.

[18] Google, "gemmlowp: a small self-contained low-precision GEMM library." https://github.com/google/gemmlowp.

[19] Qualcomm, "Qualcomm Neural Processing SDK for AI." https://developer.qualcomm.com/software/qualcomm-neural-processing-sdk.

[20] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proc. of the Conference on Computer Vision and Pattern Recognition*, June 2018.

[21] E. Park, S. Yoo, and P. Vajda, "Value-aware quantization for training and inference of neural networks," *CoRR*, vol. abs/1804.07802, 2018.

[22] M. Courbariaux and Y. Bengio, "Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1," *CoRR*, vol. abs/1602.02830, 2016. http://arxiv.org/abs/1602.02830.

[23] F. Li and B. Liu, "Ternary weight networks," *CoRR*, vol. abs/1605.04711, 2016. http://arxiv.org/abs/1605.04711.

[24] NNPACK, "Acceleration package for neural networks on multi-core cpus." https://github.com/Maratyszcza/NNPACK,.

[25] M. Dukhan, Y. Wu, and H. Lu, "QNNPACK: open source library for optimized mobile deep learning." https://code.fb.com/ml-applications/qnnpack/.

[26] O. Ronneberger, P. Fischer, and T. Brox, "U-Net: Convolutional networks for biomedical image segmentation," 2015. https://arxiv.org/abs/1505.04597.

[27] X. Zhang, X. Zhou, M. Lin, and J. Sun, "ShuffleNet: an extremely efficient convolutional neural network for mobile devices," in *Proc. of the Conference on Computer Vision and Pattern Recognition*, 2018.

[28] K. He, G. Gkioxari, P. DollÃąr, and R. Girshick, "Mask R-CNN," in *Proc. of the Intl. Conference on Computer Vision*, 2017.

[29] O. Ronneberger, P. Fischer, and T. Brox, "U-Net: convolutional networks for biomedical image segmentation," in *Proc. of the Intl. Conference on Medical Image*

Computing and Computer-Assisted Intervention, pp. 234–241, 2015.

[30] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proc. of the Intl. Conference on Computer Vision and Pattern Recognition*, 2015.

[31] C. Lea, M. D. Flynn, R. Vidal, A. Reiter, and G. D. Hager, "Temporal convolutional networks for action segmentation and detection," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

[32] D. Shingari, A. Arunkumar, B. Gaudette, S. Vrudhula, and C.-J. Wu, "DORA: optimizing smartphone energy efficiency and web browser performance under interference," in *Proc. of the Intl. Symp. on Performance Analysis of Systems and Software*, 2018.

[33] Q. Xie, M. J. Dousti, and M. Pedram, "Therminator: a thermal simulator for smartphones producing accurate chip and skin temperature maps," in *Proc. of the Intl. Symp. on Low Power Electronics and Design*, 2014.

[34] Y. ju Yu and C.-J. Wu, "Designing a temperature model to understand the thermal challenges of portable computing platforms," in *Proc. of the IEEE Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems*, 2018.

[35] B. Gaudette, C.-J. Wu, and S. Vrudhula, "Improving smartphone user experience by balancing performance and energy with probabilistic QoS guarantee," in *Proc. of the Intl. Symp. on High Performance Computer Architecture*, 2016.

[36] B. Gaudette, C.-J. Wu, and S. Vrudhula, "Optimizing user satisfaction of mobile workloads subject to various sources of uncertainties," *Transactions on Mobile Computing*, 2018.

[37] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *Advances in Neural Information Processing Systems*, 2012.

[38] B. Reagen, J. M. HernÃąndez-Lobato, R. Adolf, M. Gelbart, P. Whatmough, G. Y. Wei, and D. Brooks, "A case for efficient accelerator design space exploration via bayesian optimization," in *Proc. of the Intl. Symp. on Low Power Electronics and Design*, 2017.

[39] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," 2017. `https://arxiv.org/abs/1611.01578`.

[40] Y. Zhou, S. Ebrahimi, S. Ö. Arik, H. Yu, H. Liu, and G. Diamos, "Resource-efficient neural architect," *CoRR*, vol. abs/1806.07912, 2018.

[41] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *Proc. of the Intl. Symp. on Computer Architecture*, 2016.

[42] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "ISAAC: a convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *Proc. of the Intl. Symp. on Computer Architecture*, 2016.

[43] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient inference engine on compressed deep neural network," in *Proc. of the Intl. Symp. on Computer Architecture*, 2016.

[44] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *Proc. of the Intl. Symp. on Computer Architecture*, 2016.

[45] Y. H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Proc. of the Intl. Symp. on Computer Architecture*, 2016.

[46] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "Neurocube: A programmable digital neuromorphic architecture with high-density 3D memory," in *Proc. of the Intl. Symp. on Computer Architecture*, 2016.

[47] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen, and T. Chen, "Cambricon: An instruction set architecture for neural networks," in *Proc. of the Intl. Symp. on Computer Architecture*, 2016.

[48] N. Jouppi et al., "In-datacenter performance analysis of a tensor processing unit," in *Proc. of the Intl. Symp. on Computer Architecture*, 2017.

[49] S. Venkataramani, A. Ranjan, S. Banerjee, D. Das, S. Avancha, A. Jagannathan, A. Durg, D. Nagaraj, B. Kaul, P. Dubey, and A. Raghunathan, "Scaledeep: A scalable compute architecture for learning and evaluating deep networks," in *Proc. of the Intl. Symp. on Computer Architecture*, 2017.

[50] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "SCNN: An accelerator for compressed-sparse convolutional neural networks," in *Proc. of the Intl. Symp. on Computer Architecture*, 2017.

[51] Y. Shen, M. Ferdman, and P. Milder, "Maximizing CNN accelerator efficiency through resource partitioning," in *Proc. of the Intl. Symp. on Computer Architecture*, 2017.

[52] J. Yu, A. Lukefahr, D. J. Palframan, G. S. Dasika, R. Das, and S. A. Mahlke, "Scalpel: Customizing DNN pruning to the underlying hardware parallelism," in *Proc. of the Intl. Symp. on Computer Architecture*, 2017.

[53] C. De Sa, M. Feldman, C. Ré, and K. Olukotun, "Understanding and optimizing asynchronous low-precision stochastic gradient descent," in *Proc. of the Intl. Symp. on Computer Architecture*, 2017.

[54] B. Feinberg, S. Wang, and E. Ipek, "Making memristive neural network accelerators reliable," in *Proc. of the Intl. Symp. on High Performance Computer Architecture*, 2018.

[55] M. Song, J. Zhang, H. Chen, and T. Li, "Towards efficient microarchitectural design for accelerating unsupervised GAN-based deep learning," in *Proc. of the Intl. Symp. on High Performance Computer Architecture*, 2018.

[56] M. Rhu, M. O'Connor, N. Chatterjee, J. Pool, Y. Kwon, and S. W. Keckler, "Compressing DMA engine: Leveraging activation sparsity for training deep neural networks," in *Proc. of the Intl. Symp. on High Performance Computer Architecture*, 2018.

[57] M. Song, K. Zhong, J. Zhang, Y. Hu, D. Liu, W. Zhang, J. Wang, and T. Li, "In-situ AI: towards autonomous and incremental deep learning for IoT systems," in *Proc. of the Intl. Symp. on High Performance Computer Architecture*, 2018.

[58] A. Yazdanbakhsh, K. Samadi, H. Esmaeilzadeh, and N. S. Kim, "GANAX: a unified SIMD-MIMD acceleration for generative adversarial network," in *Proc. of the Intl. Symp. on Computer Architecture*, 2018.

[59] V. Aklaghi, A. Yazdanbakhsh, K. Samadi, H. Esmaeilzadeh, and R. K. Gupta, "SnaPEA: Predictive early activation for reducing computation in deep convolutional neural networks," in *Proc. of the Intl. Symp. on Computer Architecture*, 2018.

[60] K. Hegde, J. Yu, R. Agrawal, M. Yan, M. Pellauer, and C. W. Fletcher, "UCNN: Exploiting computational reuse in deep neural networks via weight repetition," *Proc. of the Intl. Symp. on Computer Architecture*, 2018.

[61] E. Park, D. Kim, and S. Yoo, "Energy-efficient neural network accelerator based on outlier-aware low-precision computation," *Proc. of the Intl. Symp. on Computer Architecture*, 2018.

[62] M. Song, J. Zhao, Y. Hu, J. Zhang, and T. Li, "Prediction based execution on deep neural networks," *Proc. of the Intl. Symp. on Computer Architecture*, 2018.

[63] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, V. Chandra, and H. Esmaeilzadeh, "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network," *Proc. of the Intl. Symp. on Computer Architecture*, 2018.

[64] A. Jain, A. Phanishayee, J. Mars, L. Tang, and G. Pekhimenko, "Gist: Efficient data encoding for deep neural network training," *Proc. of the Intl. Symp. on Computer Architecture*, 2018.

[65] R. Yazdani, M. Riera, J.-M. Arnau, and A. Gonzalez, "The dark side of DNN pruning," *Proc. of the Intl. Symp. on Computer Architecture*, 2018.

[66] M. Riera, J. M. Arnau, and A. Gonzalez, "Computation reuse in DNNs by exploiting input similarity," *Proc. of the Intl. Symp. on Computer Architecture*, 2018.