μLayer: Low Latency On-Device Inference Using Cooperative Single-Layer Acceleration and Processor-Friendly Quantization

Youngsok Kim[1], Joonsung Kim[1], Dongju Chae[2], Daehyun Kim[3], and Jangwoo Kim[1,*]
[1]Department of Electrical and Computer Engineering, Seoul National University
[2]Department of Computer Science and Engineering, POSTECH
[3]Samsung Research

Abstract

Emerging mobile services heavily utilize Neural Networks (NNs) to improve user experiences. Such NN-assisted services depend on fast NN execution for high responsiveness, demanding mobile devices to minimize the NN execution latency by efficiently utilizing their underlying hardware resources. To better utilize the resources, existing mobile NN frameworks either employ various CPU-friendly optimizations (e.g., vectorization, quantization) or exploit data parallelism using heterogeneous processors such as GPUs and DSPs. However, their performance is still bounded by the performance of the single target processor, so that real-time services such as voice-driven search often fail to react to user requests in time. It is obvious that this problem will become more serious with the introduction of more demanding NN-assisted services.

In this paper, we propose μLayer, a low latency on-device inference runtime which significantly improves the latency of NN-assisted services. μLayer accelerates each NN layer by simultaneously utilizing diverse heterogeneous processors on a mobile device and by performing computations using processor-friendly quantization. Two key findings motivate our work: 1) the existing frameworks are limited by single-processor performance as they execute an NN layer using only a single processor, and 2) the CPU and the GPU on the same mobile device achieve comparable computational throughput, making cooperative acceleration highly promising. First, to accelerate an NN layer using both the CPU and the GPU at the same time, μLayer employs a layer distribution mechanism which completely removes redundant computations between the processors. Next, μLayer optimizes the per-processor performance by making the processors utilize different data types that maximize their utilization. In addition, to minimize potential latency increases due to overly aggressive workload distribution, μLayer selectively increases the distribution granularity to divergent layer paths. Our experiments using representative NNs and mobile devices show that μLayer significantly improves the speed and the energy efficiency of on-device inference by up to 69.6% and 58.1%, respectively, over the state-of-the-art NN execution mechanism.

ACM Reference Format:
Youngsok Kim, Joonsung Kim, Dongju Chae, Daehyun Kim, and Jangwoo Kim. 2019. μLayer: Low Latency On-Device Inference Using Cooperative Single-Layer Acceleration and Processor-Friendly Quantization. In *Fourteenth EuroSys Conference 2019 (EuroSys '19), March 25–28, 2019, Dresden, Germany.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3302424.3303950

1 Introduction

Recent advances in Neural Networks (NNs) have made NNs achieve human-level accuracy on various tasks useful for mobile services (e.g., digit/image recognition [42, 47], language translation [38]). Using NNs, mobile service providers are actively developing new services; Google Handwriting Input to let users handwrite text on their Android devices [19], Google Translate to perform real-time visual translations [24], and YouTube to perform video segmentation [13]. Virtual assistants, such as Google Assistant and Apple Siri, use NNs to enable voice-triggered actions [64, 68]. Such diverse use cases make NNs the key workload of mobile devices toward richer user experiences.

Traditionally, mobile devices relied on the abundant cloud-side resources to satisfy the high responsiveness requirement of the mobile services; however, with the advent of high-performance mobile System-on-a-Chips (SoCs), it is now feasible to employ *on-device inference* which executes the entire NN using only the underlying hardware resources. Since the high responsiveness is a key requirement of the mobile services, mobile NN frameworks should efficiently exploit the underlying hardware resources to minimize inference

*Corresponding author.

EuroSys '19, March 25–28, 2019, Dresden, Germany

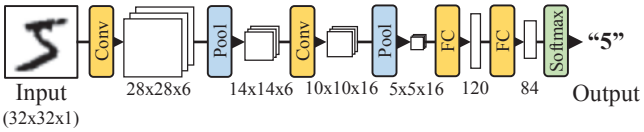
ACM ISBN 978-1-4503-6281-8/19/03…$15.00
https://doi.org/10.1145/3302424.3303950

latency. To reduce the latency, existing frameworks employ CPU-friendly optimizations (e.g., 8-bit quantized integers, vectorization) or utilize heterogeneous computing resources such as Graphics Processing Units (GPUs) [32, 43, 46]. Some recent studies [43, 55, 56] further reduce the latency by executing each NN layer on different computing resources.

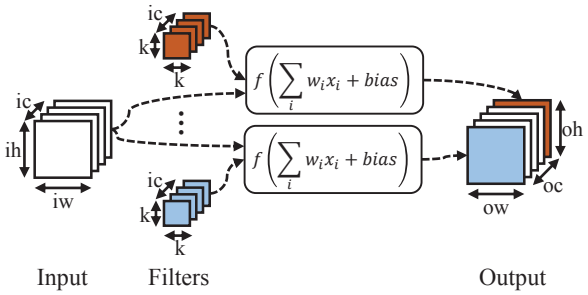
By better exploiting the hardware resources, it is now possible to achieve near-real-time responsiveness for relatively simple NNs (e.g., automatic e-mail responses [61]). Unfortunately, a majority of the real-time services still rely on the cloud as the reduced latency is still not low enough to fulfill their high performance demands. For instance, to achieve fast response times, practically all virtual assistants still offload the execution of their speech recognition NNs to the cloud instead of executing them on-device. But, now with the various computing resources available on modern high-end SoCs, we believe that fast and accurate on-device inference is feasible.

In this paper, we propose μLayer, a low latency on-device inference runtime using *cooperative single-layer acceleration* which executes a single NN layer using all of the available hardware resources. μLayer stems from two key findings: 1) the performance of the existing frameworks is limited by the single-processor performance as they execute an NN layer using only a single processor, and 2) the performance of the CPU and the GPU, the two most prevalent types of processors on mobile devices, is well-balanced. The first finding reveals that the existing frameworks fail to fully exploit the underlying hardware resources, and thus their performance is non-optimal. The second finding infers the high potential of cooperative single-layer acceleration; if the performance of one processor dominates those of the other processors, the multi-processor management overheads (e.g., memory synchronization) would easily offset the potential improvements. But, this is not the case for mobile devices.

To implement μLayer, we first propose *channel-wise workload distribution* which makes different computing resources process the disjoint sets of the output channels of an NN layer. By doing so, the channel-wise workload distribution incurs no redundant computation between the computing resources. Then, we propose *processor-friendly quantization* which makes CPUs and GPUs utilize different data types to maximize the per-processor performance when executing their portions of an NN layer. It exploits the native hardware support for the data types and NNs' ability to sustain their accuracy with fewer-bit values. Specifically, the processor-friendly quantization instructs the CPU to use 8-bit linear-quantized integers [37] and the GPU to use 16-bit floating point values (i.e., `half` data type in OpenCL) instead of the default 32-bit floating point values. After that, we propose *branch distribution* which exploits divergent data-parallel branches within an NN to further reduce the latency. By exploiting the data parallelism between the branches, we



(a) Structure of LeNet-5 [47] for digit recognition



(b) Operations of a convolutional layer

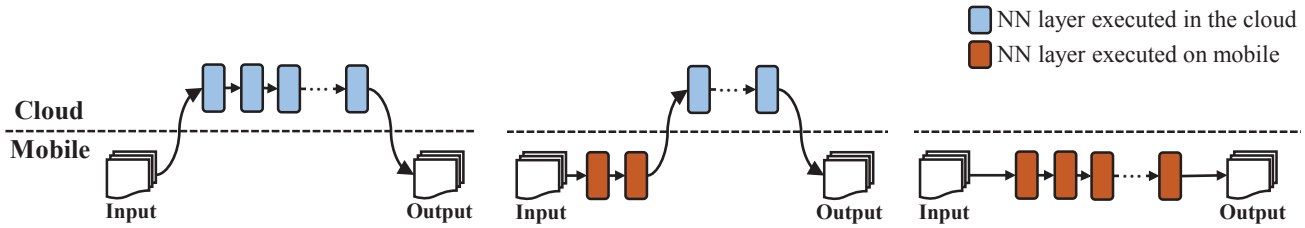
Figure 1. Multi-layer structure of NNs and the operations of a convolutional layer

can increase the utilization of both the CPU and the GPU to further reduce the inference latency.

μLayer incorporates the three mechanisms to accelerate on-device inference. Our experiments using modern SoCs and representative NNs show that μLayer greatly improves the speed by up to 59.9% and 69.6% on representative high-end and mid-range mobile SoCs, respectively, over the state-of-the-art CPU-GPU cooperative NN execution mechanism which distributes NN layers to the CPU and the GPU. μLayer also improves the energy efficiency by up to 58.1% and 57.2% on the high-end and the mid-range SoCs, respectively, over the state-of-the-art. These results clearly indicate that μLayer is highly effective in minimizing not only the latency, but also the energy consumption of NN execution.

In summary, the contributions of this paper are:

- **Identification of the High Potential of Cooperative Single-Layer Acceleration.** We show that the CPUs and the GPUs on modern mobile SoCs achieve well-balanced throughput, and then explore the acceleration of a single NN layer using both the CPU and the GPU. To the best of our knowledge, this work is the first to propose cooperative single-layer acceleration for mobile SoCs.
- **Novel Cooperative Single-Layer Acceleration Mechanisms.** By optimizing the NN execution from different aspects, our three cooperative single-layer acceleration mechanisms successfully minimize the latency, reducing the latency by up to 69.6% over the state-of-the-art.
- **Design & Implementation of μLayer.** We propose an optimized mobile NN framework named μLayer which minimizes the NN execution latency. We present its design and implementation-wise optimizations to mitigate processor management overheads (e.g., GPU command issuing, CPU-GPU memory synchronization).



(a) Cloud-only **(b)** Collaborative [41] **(c)** On-device (mobile-only)

Figure 2. Working models of existing mobile NN frameworks

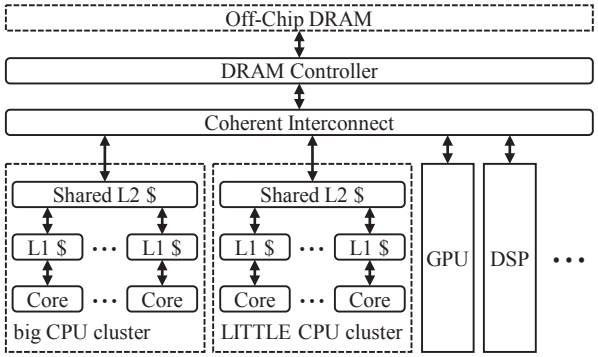


Figure 3. Diverse hardware resources on a modern SoC

2 Background

2.1 Neural Networks

Neural Networks (NNs), inspired by the way biological nervous systems process information, are capable of learning how to perform tasks without task-specific procedures or rules [51]. Within an NN, artificial neurons are connected to each other, and each inter-neuron connection is associated with a weight. Each neuron multiplies the signals (typically real numbers) by the weights of the associated connections, applies some non-linear function on the sum of the multiplication outputs, and then transmits the output to other neurons. In practice, training an NN refers to the process of adjusting the weights in a way that improves the accuracy of the NN. Among various types of NNs, we focus on Convolutional NNs (CNNs) as they are widely used across a large number of mobile services. We also focus on inference rather than training as training is mostly done offline.

A CNN consists of *layers* which perform different operations on a given input (Figure 1a). The intermediate outputs are typically three-dimensional neurons whose dimensions represent channels, height, and width. The layers, aimed at extracting spatially local features, are often grouped into three types: convolutional, fully-connected, and pooling.

Convolutional & Fully-Connected Layers. A convolutional layer computes the dot products between spatially local input neurons and *filters*, which extend through all input channels, across the width and height of input. Then, the layer accumulates biases to the dot products and applies an optional activation function, e.g., Rectified Linear Unit (ReLU), to the accumulated values. Figure 1b shows the operations of a convolutional layer which produces oc output channels from an ic-channel input. To produce an output channel, the layer applies a $k \times k \times ic$ filter to each $k \times k \times ic$ spatially local input neurons; repeating this process for oc filters produces oc output channels. A Fully-Connected (FC) layer establishes full connections between input and output neurons, and can be transformed into a convolutional layer whose number of output channels matches the number of the FC layer's output neurons.

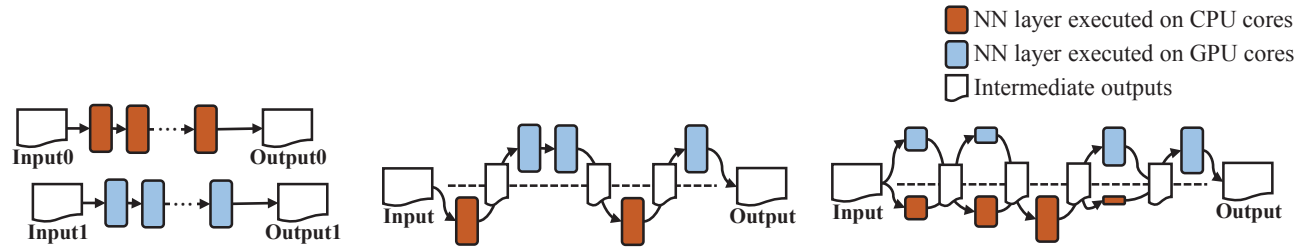
Pooling Layer. A pooling layer reduces the spatial size of the input by applying a global function (e.g., max, average) to spatially local input neurons. It differs from convolutional layers as no filters are involved in its operations and the global function does not extend through all input channels. Thus, the number of output channels equals to the number of input channels.

2.2 On-Device Inference

On-device inference refers to the execution of NNs using only the hardware resources available on a mobile device (Figure 2c). As no data is sent outside a mobile device and an internet connection is unnecessary, on-device inference can improve security, can withstand internet disconnections, and has potential to improve responsiveness by avoiding high wireless network delays. Traditionally, mobile devices relied on cloud-side servers for executing all (Figure 2a) or some (Figure 2b) NN layers as the hardware resources on the devices could not provide sufficient computational throughput; however, as recent mobile SoCs are equipped with diverse computing resources (Figure 3) such as CPUs, GPUs, and Domain-Specific Processors (DSPs), on-device inference has become a promising option over the conventional cloud-assisted inference.

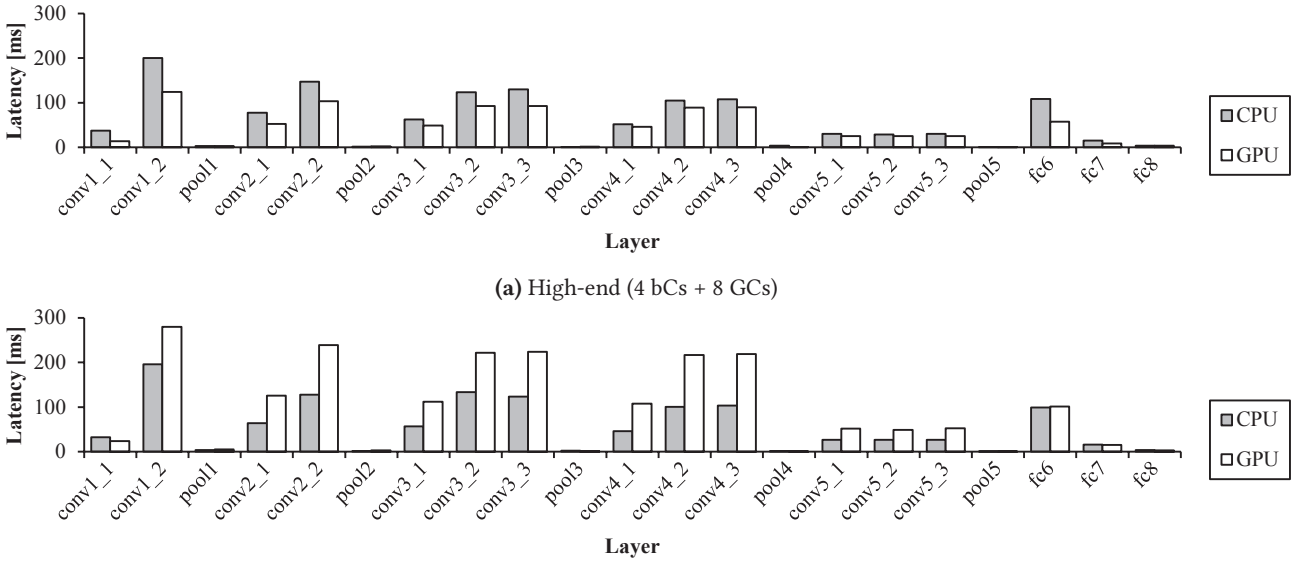
Depending on how a mobile NN framework executes the layers of an NN, the NN execution mechanisms of on-device inference can be classified into *network-to-processor mapping* and *layer-to-processor mapping*.

Network-to-Processor Mapping. The frameworks employing this mechanism (e.g., MCDNN [28]) distribute the execution of an NN on *multiple* inputs to different processors. For



(a) Network-to-processor mapping **(b)** Layer-to-processor mapping **(c)** Cooperative single-layer acceleration (Ours)

Figure 4. Illustration of the existing on-device inference mechanisms and our cooperative single-layer acceleration. The height of a layer (box) indicates the amount of the layer's computation executed on that processor.



(a) High-end (4 bCs + 8 GCs)

(b) Mid-range (8 LCs + 3 GCs)

Figure 5. Per-layer execution latency of VGG-16 on the CPUs and the GPUs of modern SoCs

example, such frameworks would run an image classification NN on the CPU for the first input image and on the GPU for the second input image (Figure 4a). The frameworks improve throughput by utilizing both the CPU and the GPU to process multiple inputs in parallel; however, the single-input latency gets bounded by the single-processor performance as each input is processed by a single processor.

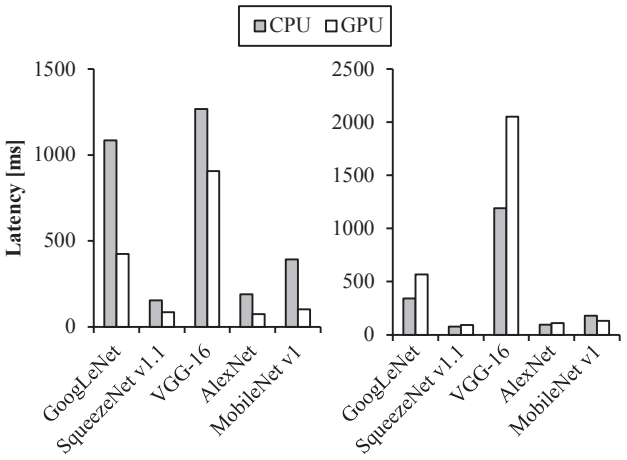
Layer-to-Processor Mapping. This mapping can reduce the single-input latency by distributing the execution of NN layers to different processors (Figure 4b). The frameworks using this mapping (e.g., DeepX [43]) would execute each layer on the processor achieving lower latency. By distributing layers, not inputs, the single-input latency can become lower than that of the network-to-processor mapping. Unfortunately, the single-input latency is still bounded by the single-processor performance as each layer is still processed by a single processor.

3 Cooperative Single-Layer Acceleration

As discussed in Section 2.2, the existing mobile NN frameworks utilize only a single processor to execute an NN layer, and bound their performance by the single-processor performance. To further reduce the latency by fully exploiting the underlying hardware resources, we propose *cooperative single-layer acceleration* which accelerates a single NN layer using both the CPU and the GPU, the two most prevalent processors on mobile SoCs, at the same time (Figure 4c). With the cooperative single-layer acceleration, the overall throughput becomes the sum of the CPU's and the GPU's throughput.

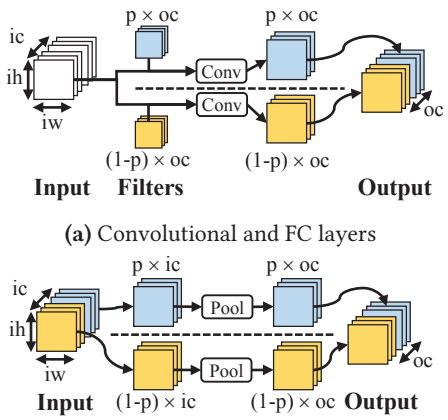
3.1 High Latency Improvement Potential

For our cooperative single-layer acceleration to be highly effective, the CPU and the GPU should achieve similar per-layer execution latency and throughput. Otherwise, the overheads associated with multi-processor management (e.g.,



(a) High-end (4 bCs + 8 GCs) **(b)** Mid-range (8 LCs + 3 GCs)

Figure 6. NN execution latency on modern SoCs



(a) Convolutional and FC layers

(b) Pooling layer

Figure 7. Channel-wise workload distribution of a layer

GPU command issuing, CPU-GPU memory synchronization) would easily offset the speed improvements. To identify whether this is the case for mobile SoCs, we profile the per-layer execution latency of VGG-16 [42] on two representative SoCs: Samsung Exynos 7420 [7] and Exynos 7880 [8]. Exynos 7420 represents high-end SoCs used by flagship smartphones; it is equipped with heterogeneous CPU cores, four high-performance cores (bCs) and four energy-efficient cores (LCs), and eight GPU cores (GCs). On the other hand, Exynos 7880 is equipped with eight LCs and three GCs, and is primarily used by mid-range smartphones. We utilize ARM Compute Library [5], which provides optimized NN layer implementations for ARM CPU and GPU cores, to execute the layers.

Our experiment using VGG-16 and the two SoCs shows that the CPUs and the GPUs achieve similar per-layer latency (Figure 5), inferring the high potential of the cooperative single-layer acceleration. On the high-end SoC, the GPU achieves an average speedup of only 1.40× over the CPU. Moreover, on the mid-range SoC, the octa-core CPU achieves 26.1% lower latency than the triple-core GPU. Such results are due to the fact that mobile GPUs primarily aim energy efficiency rather than throughput unlike server- and desktop-class GPUs. To verify whether the results also hold for other NNs, we perform a similar experiment using four additional representative NNs. The results draw a similar conclusion (Figure 6), showing that the cooperative single-layer acceleration has high potential across diverse NNs.

3.2 Channel-Wise Workload Distribution

Now that the cooperative single-layer acceleration has high potential, mobile NN frameworks should distribute the computation of a single NN layer to the CPU and the GPU in a way that maximizes the performance gains. For the purpose, we propose *channel-wise workload distribution* which makes the CPU and the GPU process disjoint sets of *output channels*. Splitting the workload in terms of output channels maximizes the performance gains by not incurring any redundant computation between the CPU and the GPU.

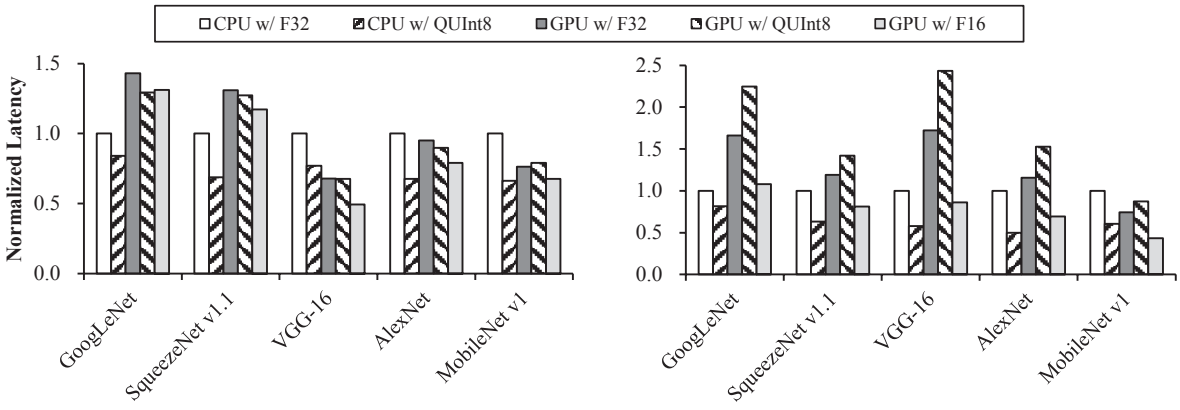
Figure 7 shows how the cooperative single-layer acceleration operates with the channel-wise workload distribution. We assume that the output channels of an NN layer is distributed to the CPU and the GPU in a ratio of $p : (1-p)$. First, for convolutional and FC layers, the filters are distributed, but the input data is shared as the filters extend through all the input channels (Figure 7a). Using the distributed filters and the shared input data, the CPU and the GPU generate their portions of the output channels. The generated output channels are then merged to form the complete output data. As the filters are distributed without any overlaps, no redundant computation between the CPU and the GPU exists. Second, for pooling layers, the input data is distributed as the global function is applied spatially, not across channels (Figure 7b). Then, the CPU and the GPU perform pooling on their portions of the input data and generate their portions of the output data. After that, the output data are merged. Similar to the case of convolutional and FC layers, no redundant computation occurs as the CPU and the GPU operate on completely disjoint sets of input data.

4 Processor-Friendly Quantization

The channel-wise workload distribution efficiently distributes the workload of an NN layer to both the CPU and the GPU in a way no redundant computation occur. Now that the workload is distributed, we seek to reduce the latency further by optimizing the per-processor performance.

4.1 Quantization

To maximize the utilization of CPUs and GPUs, prior studies propose to employ *quantization* which shrinks the default 32-bit single-precision floating-point values (`F32`) of NNs to occupy fewer bits through some transformations. Reducing



(a) High-end (4 bCs + 8 GCs) **(b)** Mid-range (8 LCs + 3 GCs)

Figure 8. Impacts of quantization on inference latency; normalized to the latency of the CPU with F32.

the bit width can greatly improve not only the speed but also the energy efficiency which is an important performance metric for mobile devices. Because of its advantages, mobile NN frameworks typically employ quantization when executing an NN [10, 28, 44].

To optimize the per-processor performance, we consider two quantization schemes: 16-bit half-precision floating-point values (F16) [35] and 8-bit linear quantization involving 8-bit integer values (QUInt8) [37]. The two schemes were chosen with an expectation that both CPUs and GPUs can greatly benefit from the data types using their native Arithmetic Logic Units (ALUs); GPUs are optimized for graphics applications which heavily utilize floating points (e.g., F16), and CPUs are equipped with vector ALUs capable of processing multiple 8-bit integers (e.g., QUInt8) in parallel.

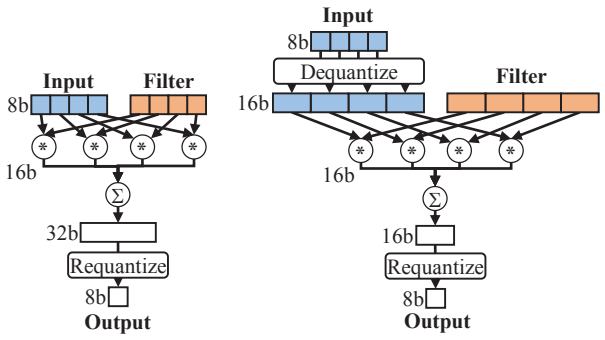
Half-Precision Floating-Point Values (F16). F16 expresses real numbers using 16 bits instead of 32 bits. It does so by reducing the numbers of exponent and significand bits of F32 by 3 and 13, respectively. Employing F16 transforms all arithmetic operations to use 16 bits, not 32 bits.

8-Bit Linear Quantization (QUInt8). Linear quantization maps 32-bit F32 values to fewer-bit representations through linear scaling. Specifically, 8-bit linear quantization maps a set of F32 values to 8-bit unsigned integers (QUInt8) where 0 and 255 map to the minimum and the maximum F32 values, respectively. By doing so, values not only shrink to a quarter the size, but also become integers which achieve higher throughput than floating-point values. As a side effect, linear quantization requires *requantization*, the process of converting 32-bit integers back to 8-bit ones [37]. This is because multiplying two 8-bit integers produces a 16-bit integer, and accumulating those 16-bit integers needs a 32-bit integer when executing convolutional and FC layers.

Figure 8 shows the reduction in the NN execution latency when the two quantization methods are applied. The results indicate that quantization is beneficial to both the CPU and the GPU; however, the processors favor different quantization mechanisms. First, GPUs greatly benefit from F16 as GPUs have native hardware support for achieving high-throughput floating-point operations. On the other hand, the latency tends to increase with QUInt8 due to the accumulation of 16-bit integers using a 32-bit integer; operating on 32-bit values reduces concurrency by half compared to 16-bit-only operations. Second, CPUs greatly benefit from QUInt8 but not from F16. The reasons turn out to be 1) the lack of vector ALU support for F16 in the CPUs we evaluated, and 2) the supported vector widths for QUInt8 is much wider than those of floating-point values. The CPU cores we evaluated lack native vector ALU support for F16, so they are forced to emulate F16 using F32; however, as F32 is the default data type for NNs, no performance difference can be observed. Even in the presence of native hardware support for F16 values, QUInt8 can still provide higher performance due to its fewer bit width; two QUInt8 operations can run in parallel using the bit width required for a single F16 operation.

4.2 Maximizing Per-Processor Throughput

Motivated by the experimental results, we propose *processor-friendly quantization* which makes the CPU and the GPU perform arithmetic operations using their preferred quantization schemes. The processor-friendly quantization orchestrates the CPU and the GPU to perform calculations using 8-bit QUInt8 integers and 16-bit F16 values, respectively. Aimed at minimizing both the latency and the energy consumption of moving the data between the cores and the memory, which is a major energy consumer on mobile devices [14, 60], the processor-friendly quantization works as follows. First, all input, filter, and output data are stored as QUInt8 to minimize the size of data movement between the cores and the memory. Second, for the CPU cores, the 8-bit



(a) CPU cores **(b)** GPU cores

Figure 9. Processor-friendly quantization

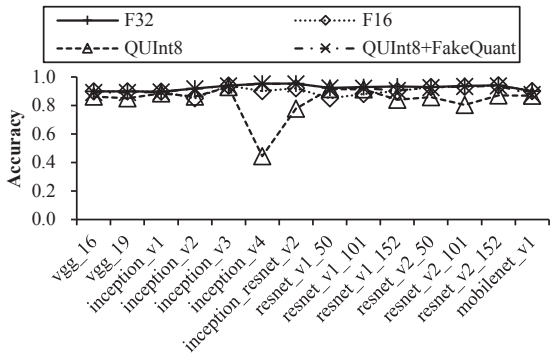
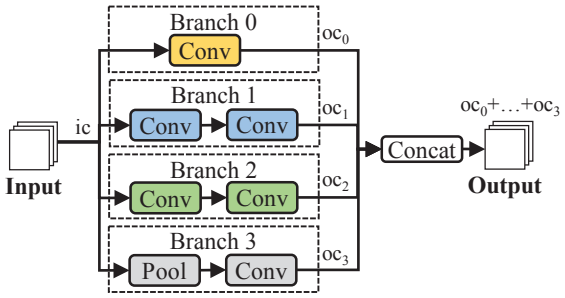


Figure 10. Impacts of quantization on the top-5 classification accuracy with the ImageNet dataset [62]

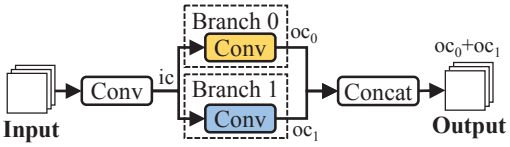
integers are processed as is using their vector ALUs (Figure 9a). Third, the GPU cores load the input and the filters as `QUInt8`, but process them as `F16` by converting the integers on-the-fly (Figure 9b). By doing so, the GPU cores can exploit their native hardware support for `F16` to minimize the latency. When storing the output back to the memory, both the CPU and the GPU requantize the output back to `QUInt8` using the pre-trained quantization information. The information can be obtained by learning the quantization range of the output during training [37]. Using the processor-friendly quantization, we can further reduce the single-layer latency by maximizing the per-processor performance.

4.3 Impacts on Inference Accuracy

Although NNs are known to sustain high inference accuracy with fewer-bit data types [26], whether the quantization mechanisms which the processor-friendly quantization utilizes also sustain accuracy remains as a valid concern. To validate that the accuracy remains high with the processor-friendly quantization, we conduct an experiment with various NNs [29–31, 36, 63, 63, 65–67] known to achieve high inference accuracy for the ImageNet dataset [62]. For the experiment, we extend TensorFlow-Slim [25] to generate `F16` and `QUInt8` versions of the NNs.



(a) Inception module of GoogLeNet [66]



(b) Fire module of SqueezeNet [34]

Figure 11. Example NNs having divergent branches

Figure 10 shows how the top-5 inference accuracy of the NNs differs when quantization is applied. The results show that employing `QUInt8` can incur a considerable accuracy loss for some NNs, e.g., 50.7 percentage points (%p) with Inception-v4. To prevent the large accuracy losses, we retrain the NNs to be aware of the 8-bit linear quantization by inserting TensorFlow's fake quantization operations [37]. The inserted operations greatly improve the accuracy of the linear-quantized NNs (`QUInt8+FakeQuant`), limiting the maximum accuracy loss to 2.7%p. The results show that the accuracy loss of the processor-friendly quantization to be marginal; the accuracy gets bounded by the fewer-bit data type (i.e., `QUInt8`), and the maximum accuracy loss of `QUInt8` is only 2.7%p. Note that prior studies often allow larger accuracy losses to improve performance (e.g., DeepMon [33] incurs 5–6% accuracy losses).

5 Branch Distribution

The channel-wise workload distribution and the processor-friendly quantization maximize the multi-processor performance and the per-processor performance, respectively. Although the two optimizations significantly reduce the latency, a recent trend in NNs toward reducing the amount of per-layer computation may limit their potential; the reduced amount of computation may not be large enough to fully benefit from the cooperative single-layer acceleration.

Some recent NNs (e.g., GoogLeNet) consist of *branches* which perform different sequences of operations on the same input data. In case of GoogLeNet, the key motivations to have the branches are: the input data to an NN (e.g., an image) can have an extremely large variation in size, increasing the number of layers is prone to overfitting, and naïvely stacking large convolutional layers is computationally expensive [66].

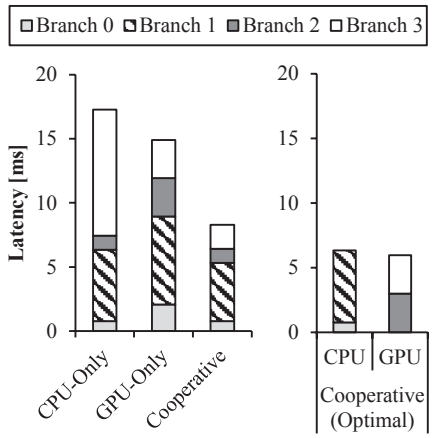


Figure 12. Potential latency benefits of branch distribution

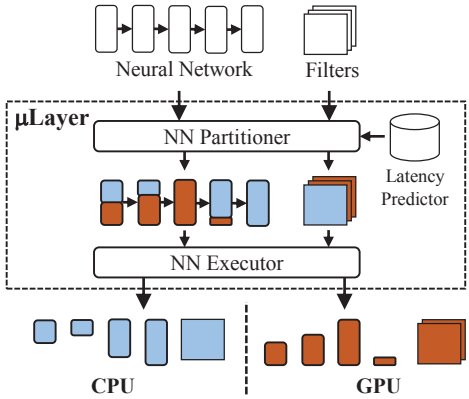


Figure 13. µLayer runtime architecture

To resolve the issues, GoogLeNet employs the Inception module which performs convolutions having different filter sizes and pooling on the same input data in parallel, and then concatenates the outcomes along the channel dimension (Figure 11a). Such branches can be seen often in other recent NNs as well, e.g., the Fire module of SqueezeNet (Figure 11b).

We find that such branches can make the channel-wise workload distribution achieve sub-optimal performance. The reason is that the channel-wise workload distribution *always* executes each NN layer using both the CPU and the GPU, incurring high CPU-GPU synchronization overheads. Accordingly, the latency of different branches, which can be hidden by running the branches in parallel, gets exposed. Figure 12 illustrates an example scenario where the channel-wise workload distribution executes the first Inception module of GoogLeNet on the high-end SoC. Due to the small filter sizes of the convolutional layers and that the layers are executed in a serialized manner, the channel-wise workload distribution with the processor-friendly quantization (`Cooperative`) improves the speed by 52.1% over CPU-only inference with the 8-bit linear quantization (`CPU-Only`). However, if we assign branches 0 and 1 to the CPU and branches 2 and 3 to the GPU, i.e., `Cooperative (Optimal)`, the execution latency becomes 6.3 ms, achieving a speed improvement of 63.4%. The results indicate that the branches whose latency can be hidden by executing them in parallel should be considered to minimize the latency.

To exploit such branches to further reduce the latency, we propose *branch distribution* which 1) identifies a set of parallelizable branches, and 2) executes the branches in parallel by assigning them to the CPU and the GPU. By doing so, the branch distribution allows parallel execution of divergent branches, expanding the optimization coverage of the channel-wise workload distribution and the processor-friendly quantization.

In order to derive the optimal branch-to-processor mapping, the branch distribution utilizes the per-processor execution latency of a branch. For example, to distribute the four branches of the Inception module, the branch distribution first collects the CPU- and the GPU-only execution latency of the four branches. Then, for a possible branch-to-processor mapping (e.g., branches 0 and 1 to the CPU and branches 2 and 3 to the GPU), the branch distribution estimates the total execution latency by calculating the sum of the per-processor, per-branch execution latency (i.e., the sum of the CPU-only execution latency of branches 0 and 1, and the sum of the GPU-only execution of branches 2 and 3). After estimating the execution latency of all possible branch-to-processor mappings, the branch distribution selects the mapping which incurs the lowest latency as the optimal mapping. Note that the branch distribution does not involve the channel-wise workload distribution; all the layers of a branch get executed on a single processor.

6 Implementation

In this section, we describe µLayer, a software runtime for mobile NN execution frameworks implementing the channel-wide workload distribution, the processor-friendly quantization, and the branch distribution. µLayer analyzes a given NN, applies the proposed optimizations to the NN, makes an execution plan for the NN, and executes the layers according to the execution plan. µLayer assumes that the 8-bit linear quantization is already applied to the given NN; existing tools (e.g., TensorFlow's fake quantization [37]) can transform non-quantized NNs into linear-quantized ones.

µLayer consists of three components: an NN partitioner, a latency predictor, and an NN executor (Figure 13). The NN partitioner is responsible for generating a cooperative execution plan. For each NN layer, the NN partitioner decides the split ratio p ($0 \leq p \leq 1$) for the channel-wise workload distribution. In our implementation, the NN partitioner considers p values of 0.75, 0.5, and 0.25. The NN partitioner refers to the latency predictor to identify the optimal p value. Given

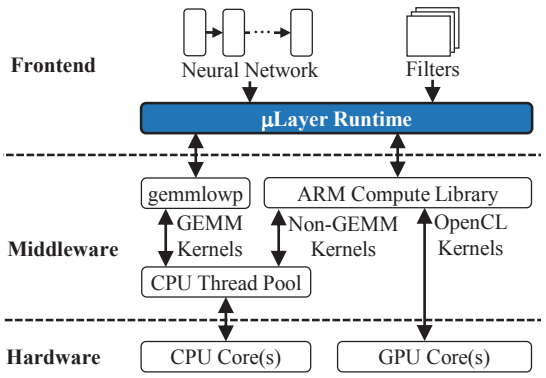


Figure 14. µLayer-augmented mobile NN framework

Figure 15. Our experimental setup to measure the energy consumption of mobile SoCs

Table 1. Evaluated NNs and our proposals' applicability

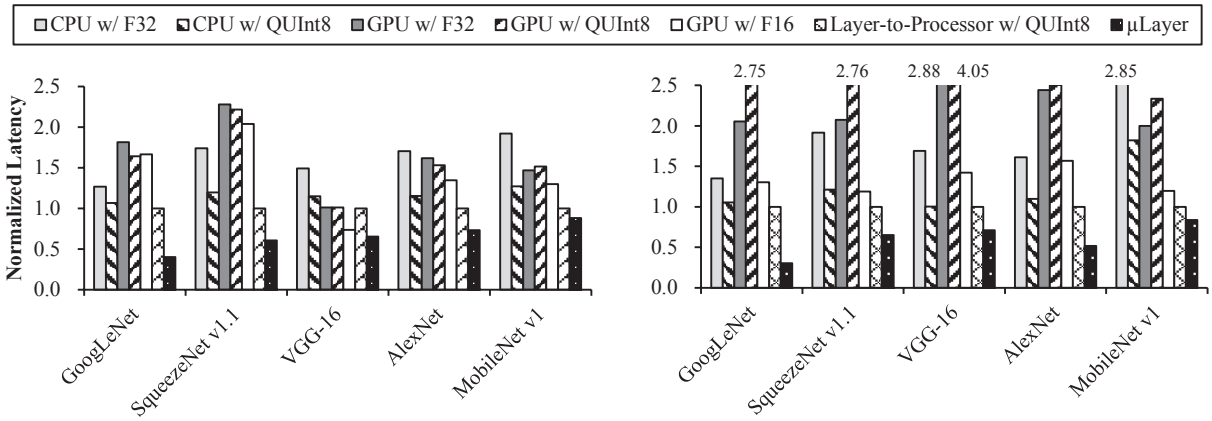
Name	Applicability		
	Ch. Dist. (Sec. 3.2)	Proc. Quant. (Sec. 4.2)	Br. Dist. (Sec. 5)
GoogLeNet [66]	✓	✓	✓
SqueezeNet v1.1 [34]	✓	✓	✓
VGG-16 [63]	✓	✓	
AlexNet [42]	✓	✓	
MobileNet v1 [31]	✓	✓	

the parameters of an NN layer (e.g., input and filter sizes) and the p value, the latency predictor estimates the execution latency of the layer. To do so, we extend the performance prediction model of Neurosurgeon [41] to take p values into an account. It first estimates the latency of the CPU- and the GPU-only execution using the logarithmic-based regression as described in [41]. Then, it scales the estimated latency with respect to the given p value. After that, the latency predictor reports the estimated latency to NN partitioner. The generated execution plan is then sent to NN executor.

The NN executor is responsible for executing the given NN with respect to the generated execution plan. Upon receiving the execution plan from the NN partitioner, the NN executor first uploads the filter values of the NN to CPU and GPU memory. When uploading the filter values to the GPU memory, the NN partitioner dequantizes the `QUInt8` filter values to `F16` values as the processor-friendly quantization makes the GPU operate on `F16` values. After that, the NN executor invokes the underlying middleware's API functions (e.g., OpenCL commands for the GPU) to execute the layer with respect to the optimal p value.

Figure 14 shows the software architecture of our example µLayer-augmented mobile NN framework. The framework utilizes ARM Compute Library (ACL) [5] and gemmlowp [9] as the underlying middlewares. ACL provides API functions which utilize NEON- and OpenCL-based kernels optimized for ARM Cortex CPUs and Mali GPUs. The framework mostly utilizes ACL; however, for executing GEneralized Matrix Multiplications (GEMMs), a key operation of convolutional and FC layers, on CPU cores, we use gemmlowp instead as it operates on `QUInt8` values by default and provides higher `QUInt8` GEMM performance. Note that the middleware can be chosen by developers according to their needs. For instance, if the target CPU implements x86 instruction set architecture instead of AArch32/64, it would be desirable to employ Intel Math Kernel Library [1] or AMD Math Library [6] instead.

As the cooperative execution demands multi-processor management, the associated overheads such as GPU invocation latency may offset the performance benefits. To mitigate the overheads, our framework exploits asynchronous GPU command issuing and shared CPU-GPU memory. First, the framework maximizes the use of asynchronous GPU commands to hide the GPU invocation overheads. For instance, when executing a convolutional layer, the framework first issues asynchronous GPU commands, executes the CPU-side operations, and then waits until the GPU-side operations are completed. By doing so, the GPU invocation overlaps with the CPU-side operations, reducing the overall latency. Second, to avoid high-latency CPU-GPU memory copies, the framework utilizes the shared CPU-GPU memory of mobile SoCs. On the SoCs, the CPU and the GPU share the same physical memory, allowing zero-copy memory transfers. To eliminate the CPU-GPU memory copies using zero-copy memory, the framework always allocates memory buffers through OpenCL's `clCreateBuffer` function with `CL_MEM_ALLOC_HOST_PTR` flag. Then, during execution, the framework maps the memory regions the CPU needs to access using `clEnqueueMapBuffer` function; input and output memory regions are mapped with `CL_MAP_READ` and `CL_MAP_WRITE_INVALIDATE_REGION` flags, respectively. In this way, no unnecessary data copies between the CPU and the GPU occur, eliminating the CPU-GPU data transfer overheads. Note that the mapping and the unmapping operations are also asynchronous; they happen in parallel with the CPU-side operations.



(a) High-end (4 bCs + 8 GCs) **(b)** Mid-range (8 LCs + 3 GCs)

Figure 16. NN execution latency of the single-processor mechanism, the layer-to-processor mechanism, and µLayer; normalized to the latency of the layer-to-processor mechanism.

7 Evaluation

7.1 Experimental Setup

To evaluate the effectiveness of µLayer, we measure the latency and the energy efficiency of our µLayer-augmented mobile NN framework on two representative modern mobile SoCs: Samsung Exynos 7420 (of Samsung Galaxy Note 5) and Samsung Exynos 7880 (of Samsung Galaxy A5). First, Exynos 7420 consists of four high-performance 2.1-GHz ARM Cortex-A57 cores, four energy-efficient 1.5-GHz ARM Cortex-A53 cores, and an octa-core 700-MHz ARM Mali-T760 GPU. Its hardware specification represents the mobile SoCs used by high-end smartphones. Second, Exynos 7880 represents the mobile SoCs of mid-range smartphones and consists of eight 1.9-GHz ARM Cortex-A53 CPU cores and a triple-core 962-MHz ARM Mali-T830 GPU. To measure the energy consumption of the mobile SoCs, we utilize the High Voltage Power Monitor (HVPM) from Monsoon Solutions, Inc. [57]; we detach the batteries from the smartphones, solder wires to them, and connect the wires to the HVPM as shown in Figure 15.

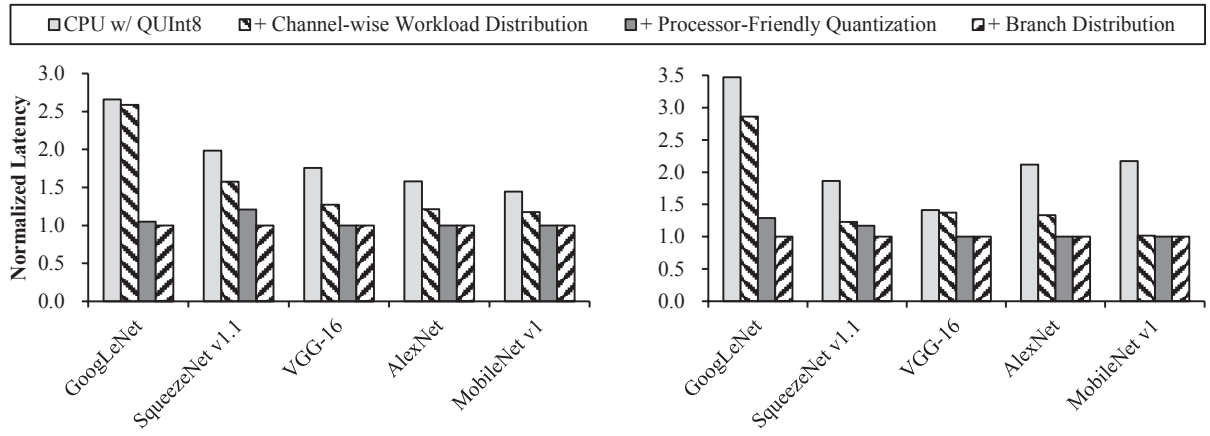
Table 1 lists the five representative NNs we use for evaluation and whether our proposals can be applied to them. The NNs represent three different NN classes. First, GoogLeNet [66] and SqueezeNet v1.1 [34] represent NNs having divergent branches. Second, VGG-16 [63] and AlexNet [42] represent early NNs having large filter sizes. Third, MobileNet v1 [31] represents small-scale NNs aimed at minimizing the amount of computation. All of the NNs are designed for image classification, specifically the ImageNet dataset.

7.2 Low NN Execution Latency

In this experiment, we evaluate the speed improvements of µLayer by comparing the NN execution latency against those of the single-processor and the layer-to-processor mechanisms. The single-processor mechanism executes an entire NN on either the CPU or the GPU. We consider all possible data types (i.e., `F32`, `F16`, `QUInt8`) for the mechanisms. The layer-to-processor mechanism executes each NN layer on the processor achieving lower latency. We compare µLayer against the mechanism using `QUInt8`, not `F32`, as using the integers achieves lower latency. Compared to the two existing mechanisms, µLayer should reduce the latency as it uses both the CPU and the GPU to execute a single NN layer.

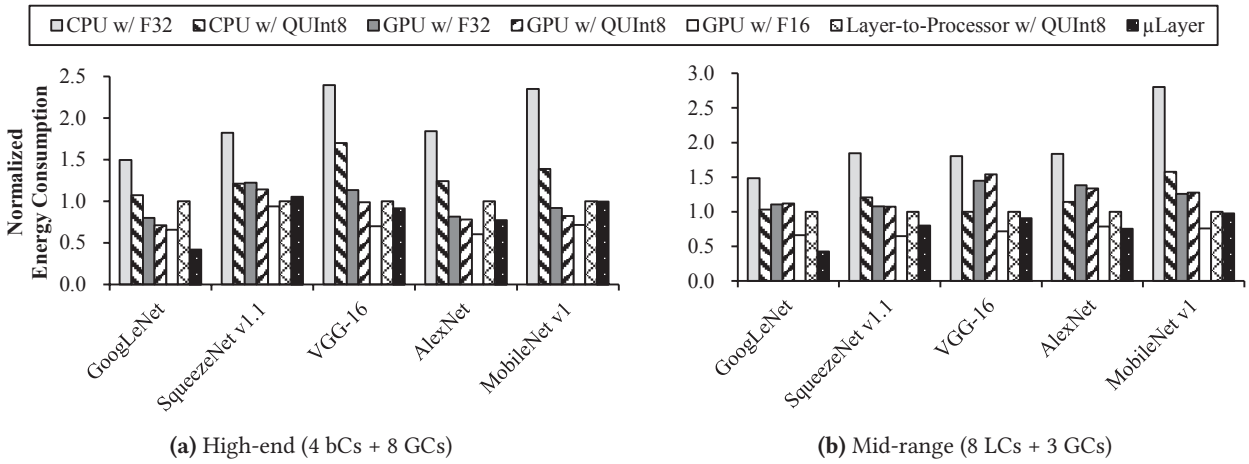
Figure 16 shows the NN execution latency of the two mechanisms and µLayer. The results show that µLayer significantly improves speed by up to 59.9% (high-end) and 69.6% (mid-range) over the layer-to-processor mechanism and achieves geometric mean speed improvements of 30.5% (high-end) and 35.3% (mid-range). Even for VGG-16 on the high-end SoC, the only case where the single-processor mechanism is faster than the layer-to-processor mechanism, µLayer reduces latency by 11.2%. The results clearly indicate that µLayer effectively reduces the NN execution latency over the prior mechanisms by fully exploiting all of the underlying hardware resources.

To analyze the contributions of µLayer's three optimizations in speed improvements, we compare how the NN execution latency changes as the optimizations are incrementally applied. Figure 17 shows how the latency reduces as we employ each of the optimizations. The results show that the amounts of the contributions vary across the NNs and the SoCs. First, the channel-wise workload distribution contributes the most for AlexNet having only a few, but large convolutional layers. Second, GoogLeNet benefits the most from the processor-friendly quantization as it consists of a large number of convolutional layers with small filter sizes. Third, the branch distribution further reduces the latency of GoogLeNet and SqueezeNet v1.1 by exploiting their divergent branches. In summary, the three optimizations exploit different optimization opportunities, achieving large speed improvements when utilized as a whole.



(a) High-end (4 bCs + 8 GCs) **(b)** Mid-range (8 LCs + 3 GCs)

Figure 17. Contribution of μLayer's optimizations to NN execution latency; normalized to the latency of the complete μLayer.



(a) High-end (4 bCs + 8 GCs) **(b)** Mid-range (8 LCs + 3 GCs)

Figure 18. Energy consumption of the single-processor mechanism, the layer-to-processor mechanism, and μLayer; normalized to the energy consumption of the layer-to-processor mechanism.

7.3 Low Energy Consumption

Despite a significant reduction in the NN execution latency, μLayer should achieve low energy consumption as it is a major concern of mobile devices; higher energy consumption degrades user experience by making the battery drain faster and demanding users to charge their mobile devices more often. As μLayer accelerates an NN layer using both the CPU and the GPU at the same time, dynamic power consumption naturally increases. This may lead to an increase in the energy consumption if the reduction in the latency is not large enough. Therefore, for μLayer to be widely deployed, it should incur low energy consumption.

To evaluate whether μLayer is energy efficient, we measure the amount of the energy consumed by the mobile SoCs during the execution of NNs. Figure 18 shows the energy consumed by the single-processor mechanism, the layer-to-processor mechanism, and μLayer. Compared to the state-of-the-art layer-to-processor mechanism, μLayer improves the energy efficiency by geometric means of 1.26× and 1.34× on the high-end and the mid-range SoCs, respectively. The improvements are due to 1) the lower execution latency by fully exploiting both the CPU and the GPU, and 2) the reduction in the memory bandwidth consumed by accessing data using 8-bit `QUInt8` instead of 32-bit `F32`. Moreover, the energy efficiency of μLayer is comparable to that of the single-processor mechanism. The results indicates that μLayer not only reduces the NN execution latency, but also achieves high energy efficiency, allowing mobile devices to easily employ μLayer without major energy consumption concerns.

8 Related Work

8.1 Executing NNs with Heterogeneous Processors

Mobile SoCs are equipped with CPUs along with diverse heterogeneous processors such as GPUs and DSPs. Some prior studies such as CNNdroid [46] and DeepSense [32] propose to execute NN layers on the GPU which is known to achieve higher computational throughput than the CPU. DeepMon [33] specializes in accelerating continuous vision applications by using the GPU and by exploiting inter-frame similarities. RSTensorFlow [12] achieves wide applicability by utilizing the GPU through Android's RenderScript, and MobiRNN [15] accelerates Recurrent Neural Networks using the GPU. On the other hand, DeepEar [45] exploits DSPs to enable low-power execution of audio sensing NNs.

Some other studies propose to distribute multiple runs of an NN to different hardware resources to improve performance. Depending on the remaining energy and cash budget (for using the cloud), MCDNN [28] executes an NN either on the mobile device or in the cloud. LEO [23] distributes multiple runs of sensing NNs to the CPU, the GPU, the DSP, and the cloud depending on their availability. Neurosurgeon [41] executes earlier layers on the mobile device and the following layers in the cloud to reduce the latency and the energy consumption.

Another class of studies distribute the layers of an NN to the heterogeneous processors to improve performance. For example, DeepX [43] splits an NN into multiple groups of layers, apply some optimizations (e.g., pruning) to the groups, and then distributes the groups to the heterogeneous processors. Mirhoseini et al. [55, 56] develop an NN which optimizes the layer-to-processor placement to maximize NN execution performance. The layer-to-processor mapping is supported by modern NN frameworks such as TensorFlow [11] and Caffe-HRT [2].

Unfortunately, none of the prior work accelerates a single NN layer using the heterogeneous processors at the same time, making their performance get bounded by the single-processor performance. On the other hand, μLayer fully exploits all of the heterogeneous processors to achieve significant speed improvements over prior studies.

8.2 Reducing the Computational Overheads of NNs

Reducing the computational overheads of NNs has been an active research area due to the limited computational throughput of mobile SoCs. First, quantization transforms the 32-bit floating points of the NNs into fewer-bit data types, reducing memory footprint sizes and possibly improving performance by increasing the utilization of the underlying hardware resources. For example, TensorFlow Lite [10] employs 8-bit linear-quantized integers [37] to improve the NN execution speed by up to 4× by exploiting CPU's vector ALUs. Second, some other studies [27, 50, 53, 54, 69, 70] seek to reduce the complexity of NNs through compression. Compressing an NN typically involves the elimination of near-zero inter-neuron connections which have little impacts on inference accuracy. μLayer is orthogonal to such optimizations as it does not apply any structural modifications to a given NN. In fact, the optimizations can be used with μLayer to achieve lower NN execution latency.

8.3 Neural Processing Units

The steadily increasing demands for higher NN execution performance have attracted a large interest from the computer architecture community. As a result, the community has proposed various NN accelerators, typically called Neural Processing Units (NPUs), which specialize in NN operations. Examples include the DianNao family [16–18, 20, 52]. Energy-efficient NPUs tailored for mobile devices (e.g., Intel Myriad X [58], Arm ML Processor [21], Google's Edge TPU [3]) have also been proposed. Now, some recent mobile SoCs (e.g., HiSilicon Kirin 970 [22], Rockchip RK3399Pro [4]) are equipped with a dedicated NPU which applications can exploit when executing NNs.

Although our work mainly focuses on CPU-GPU cooperative single-layer acceleration, we claim that μLayer can easily be extended to support NPUs (and also DSPs). First, the channel-wise workload distribution can be extended to distribute a layer's output channels to not only the CPU and the GPU, but also the NPU. Second, for the NPU, the processor-friendly quantization can fully exploit the NPU using an NPU-friendly quantization scheme (e.g., 8-bit linear quantization for Google's Tensor Processing Unit [39]). Third, the branch distribution can benefit from having the NPU by being able to run more branches in parallel. Accordingly, even in the presence of NPUs, the key ideas of our work still hold.

8.4 CPU-GPU Cooperative Processing

Outside the context of NN execution, there have been efforts to utilize both the CPU and the GPU at the same time to accelerate general-purpose computations. Examples of the studies on such CPU-GPU cooperative processing include the Single Kernel Multiple Devices (SKMD) system [48, 49], Fluidic Kernels [59], and adaptive heterogeneous scheduling [40]. Although the studies also utilize cooperative CPU-GPU processing, μLayer differs from the studies as it exploits the unique characteristics of NNs (e.g., the channel-wise workload distribution, the branch distribution).

9 Conclusion

To fulfill the high performance requirements of real-time NN-assisted services, mobile devices must fully utilize their hardware resources to achieve low NN execution latency. Existing mobile NN frameworks seek to reduce the latency by employing CPU-friendly optimizations such as vectorization and quantization, by utilizing heterogeneous hardware resources such as GPUs or DSPs, and by distributing layers

to different hardware resources. Unfortunately, their performance gets bounded by the single-processor performance.

In this paper, we aimed at reducing the latency by accelerating each NN layer using both the CPU and the GPU at the same time. Based on the high performance improvement potential of cooperative single-layer execution, we proposed three mechanisms (i.e., the channel-wise workload distribution, the processor-friendly quantization, the branch distribution) which fully utilize the underlying hardware resources. The resulting mobile NN framework, called μLayer, significantly reduces the latency by up to 69.6% and also improves the energy efficiency by up to 58.1%.

Acknowledgments

This work was partly supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (MSIT) (NRF-2015M3C4A7065647, NRF-2017R1A2B3011038), Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.1711080972, Neuromorphic Computing Software Platform for Artificial Intelligence Systems), and Creative Pioneering Researchers Program through Seoul National University. We also appreciate the support from Automation and Systems Research Institute (ASRI), Inter-university Semiconductor Research Center (ISRC), and Neural Processing Research Center (NPRC) at Seoul National University.

References

[1] 2017. Intel Math Kernel Library. https://software.intel.com/en-us/mkl.
[2] 2018. Caffe-HRT. https://github.com/OAID/Caffe-HRT
[3] 2018. Edge TPU. https://cloud.google.com/edge-tpu/
[4] 2018. Rockchip Released Its First AI Processor RK3399Pro NPU Performance up to 2.4TOPs. http://www.rock-chips.com/a/en/News/Press_Releases/2018/0108/869.html
[5] 2018. The ARM Computer Vision and Machine Learning library. https://github.com/ARM-software/ComputeLibrary
[6] 2019. AMD Math Library (LibM). https://developer.amd.com/amd-cpu-libraries/amd-math-library-libm/.
[7] 2019. Exynos 7 Octa 7420 Processor: Specs, Features | Samsung Exynos. https://www.samsung.com/semiconductor/minisite/exynos/products/mobileprocessor/exynos-7-octa-7420/
[8] 2019. Exynos 7 Series 7880 Processor: Specs, Features | Samsung Exynos. https://www.samsung.com/semiconductor/minisite/exynos/products/mobileprocessor/exynos-7-series-7880/
[9] 2019. gemmlowp: a small self-contained low-precision GEMM library. https://github.com/google/gemmlowp
[10] 2019. Introduction to TensorFlow Lite. https://www.tensorflow.org/mobile/tflite/
[11] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *Proc. 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.
[12] Moustafa Alzantot, Yingnan Wang, Zhengshuang Ren, and Mani B. Srivastava. 2017. RSTensorFlow: GPU Enabled TensorFlow for Deep Learning on Commodity Android Devices. In *Proc. 1st International Workshop on Deep Learning for Mobile Systems and Applications (EMDL)*.
[13] Valentin Bazarevsky and Andrei Tkachenka. 2018. Mobile Real-time Video Segmentation. https://ai.googleblog.com/2018/03/mobile-real-time-video-segmentation.html.
[14] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, and Onur Mutlu. 2018. Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks. In *Proc. 23rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
[15] Qingqing Cao, Niranjan Balasubramanian, and Aruna Balasubramanian. 2017. MobiRNN: Efficient Recurrent Neural Network Execution on Mobile GPU. In *Proc. 1st International Workshop on Deep Learning for Mobile Systems and Applications (EMDL)*.
[16] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning. In *Proc. 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
[17] Yunji Chen, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2016. DianNao Family: Energy-Efficient Hardware Accelerators for Machine Learning. *Commun. ACM* 59, 11 (2016).
[18] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, and Olivier Temam. 2014. DaDianNao: A Machine-Learning Supercomputer. In *Proc. 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.
[19] Thomas Deselaers, Daniel Keysers, Henry Rowley, Li-Lun Wang, Victor Cărbune, Ashok Popat, and Dhyanesh Narayanan. 2015. Google Handwriting Input in 82 languages on your Android mobile device. https://ai.googleblog.com/2015/04/google-handwriting-input-in-82.html.
[20] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. 2015. ShiDianNao: Shifting Vision Processing Closer to the Sensor. In *Proc. 42nd Annual International Symposium on Computer Architecture (ISCA)*.
[21] Iam Forsyth. 2018. Arm ML Processor: Powering Machine Learning at the Edge. https://community.arm.com/processors/b/blog/posts/arm-ml-processor
[22] Andrei Frumusanu. 2018. HiSilicon Kirin 970 - Android SoC Power & Performance Overview. https://www.anandtech.com/show/12195/hisilicon-kirin-970-power-performance-overview
[23] Petko Georgiev, Nicholas D. Lane, Kiran K. Rachuri, and Cecilia Mascolo. 2016. LEO: Scheduling Sensor Inference Algorithms across Heterogeneous Mobile Processors and Network Resources. In *Proc. 22nd Annual International Conference on Mobile Computing and Networking (MobiCom)*.
[24] Otavio Good. 2015. How Google Translate squeezes deep learning onto a phone. https://ai.googleblog.com/2015/07/how-google-translate-squeezes-deep.html.
[25] Sergio Guadarrama and Nathan Silberman. 2016. TensorFlow-Slim: a lightweight library for defining, training and evaluating complex models in TensorFlow. https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/slim
[26] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep Learning with Limited Numerical Precision. In *Proc. 32nd International Conference on Machine Learning (ICML)*.
[27] Song Han, Huizi Mao, and William J. Dally. 2016. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. In *Proc. 4th International Conference on Learning Representations (ICLR)*.
[28] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. 2016. MCDNN: An

Approximation-Based Execution Framework for Deep Stream Processing Under Resource Constraints. In *Proc. 14th ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*.

[29] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proc. 29th IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[30] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Identity Mappings in Deep Residual Networks. In *Proc. 14th European Conference on Computer Vision (ECCV)*.

[31] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. https://arxiv.org/abs/1704.04861

[32] Loc N. Huynh, Rajesh K. Balan, and Youngki Lee. 2016. DeepSense: A GPU-based Deep Convolutional Neural Network Framework on Commodity Mobile Devices. In *Proc. 2016 Workshop on Wearable Systems and Applications (WearSys)*.

[33] Loc N. Huynh, Youngki Lee, and Rajesh Krishna Balan. 2017. DeepMon: Mobile GPU-based Deep Learning Framework for Continuous Vision Applications. In *Proc. 15th ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*.

[34] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size. https://arxiv.org/abs/1602.07360.

[35] IEEE-SA Standard Board. 2008. IEEE Standard for Floating-Point Arithmetic.

[36] Sergey Ioffe and Christian Szegedy. 2015. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *Proc. 32nd International Conference on Machine Learning (ICML)*.

[37] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2018. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In *Proc. 30th IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[38] Melvin Johnson, Mike Schuster, Quoc V. Le, Maxim Krikun, Yonghui Wu, Zhifeng Chen, Nikhil Thorat, Fernanda Viégas, Martin Wattenberg, Greg Corrado, Macduff Hughes, and Jeffrey Dean. 2017. Google's Multilingual Neural Machine Translation System: Enabling Zero-Shot Translation. *Transactions of the Association for Computational Linguistics (TACL)* 5 (2017).

[39] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proc. 44th Annual International Symposium on Computer Architecture (ISCA)*.

[40] Rashid Kaleem, Rajkishore Barik, Tatiana Shpeisman, Brian T. Lewis, Chunling Hu, and Keshav Pingali. 2014. Adaptive Heterogeneous Scheduling for Integrated GPUs. In *Proc. 23rd International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

[41] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Linjia Tang. 2017. Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge. In *Proc. 22nd ACM International Conference on Architectural Support for Operating Systems and Programming Languages (ASPLOS)*.

[42] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Proc. 25th Conference on Neural Information Processing Systems (NIPS)*.

[43] Nicholas D. Lane, Sourav Bhattacharya, Petko Georgiev, Claudio Forlivesi, Lei Jiao, Lorena Qendro, and Fahim Kawsar. 2016. DeepX: A Software Accelerator for Low-Power Deep Learning Inference on Mobile Devices. In *Proc. 15th International Conference on Information Processing in Sensor Networks (IPSN)*.

[44] Nicholas D. Lane, Sourav Bhattacharya, Akhil Mathur, Claudio Forlivesi, and Fahim Kawsar. 2016. DXTK: Enabling Resource-efficient Deep Learning on Mobile and Embedded Devices with the DeepX Toolkit. In *Proc. 8th EAI International Conference on Mobile Computing, Applications and Services (MobiCASE)*.

[45] Nicholas D. Lane, Petko Georgiev, and Lorena Qendro. 2015. DeepEar: Robust Smartphone Audio Sensing in Unconstrained Acoustic Environments using Deep Learning. In *Proc. 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp)*.

[46] Seyyed Salar Latifi Oskouei, Hossein Golestani, MAtin Hashemi, and Soheil Ghiasi. 2016. CNNdroid: GPU-Accelerated Execution of Trained Deep Convolutional Neural Networks on Android. In *Proc. 24th ACM International Conference on Multimedia (MM)*.

[47] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-Based Learning Applied to Document Recognition. *Proc. IEEE* 86, 11 (1998).

[48] Janghaeng Lee, Mehrzad Samadi, Yongjun Park, and Scott Mahlke. 2013. Transparent CPU-GPU Collaboration for Data-Parallel Kernels on Heterogeneous Systems. In *Proc. 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

[49] Janghaeng Lee, Mehrzad Samadi, Yongjun Park, and Scott Mahlke. 2015. SKMD: Single Kernel on Multiple Devices for Transparent CPU-GPU Collaboration. *ACM Transactions on Computer Systems (TOCS)* 33, 3 (2015).

[50] Dawei Li, Xiaolong Wang, and Deguang Kong. 2018. DeepRebirth: Accelerating Deep Neural Network Execution on Mobile Devices. In *Proc. 32nd AAAI Conference on Artificial Intelligence (AAAI)*.

[51] Richard P. Lippmann. 1987. An Introduction to Computing with Neural Nets. *IEEE ASSP Magazine* 4, 2 (1987).

[52] Daofu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Temam, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. 2015. PuDianNao: A Polyvalent Machine Learning Accelerator. In *Proc. 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[53] Sicong Liu, Yingyan Lin, Zimu Zhou, Kaiming Nan, Hui Liu, and Junzhao Du. 2018. On-Demand Deep Model Compression for Mobile Devices: A Usage-Driven Model Selection Framework. In *Proc. 16th ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*.

[54] Partha Maji and Robert Mullins. 2017. 1D-FALCON: Accelerating Deep Convolutional Neural Network Inference by Co-optimization of Models and Underlying Arithmetic Implementation. In *Proc. 26th International Conference on Artificial Neural Networks (ICANN)*.

[55] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V. Le, and Jeff Dean. 2018. Hierarchical Planning for Device Placement. In *Proc. 6th International Conference on Learning Representations (ICLR)*.

[56] Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. 2017. Device Placement Optimization with Reinforcement Learning. In *Proc. 34th International Conference on Machine Learning (ICML)*.

[57] Monsoon Solutions, Inc. 2018. Mobile Device Power Monitor Manual Ver 1.19. http://msoon.github.io/powermonitor/PowerTool/doc/Power%20Monitor%20Manual.pdf

[58] Movidius, an Intel Company. 2018. Enhanced Visual Intelligence at the Network Edge: Intel Movidius Myriad X VPU with Neural Compute Engine. https://movidius-uploads.s3.amazonaws.com/1532110136-MyriadXVPU_ProductBrief_final_07.18.pdf

[59] Prasanna Pandit and R. Govindarajan. 2014. Fluidic Kernels: Cooperative Execution of OpenCL Programs on Multiple Heterogeneous Devices. In *Proc. 2014 International Symposium on Code Generation and Optimization (CGO)*.

[60] Dhinakaran Pandiyan and Carole-Jean Wu. 2014. Quantifying the Energy Cost of Data Movement for Emerging Smart Phone Workloads on Mobile Platforms. In *Proc. 2014 IEEE International Symposium on Workload Characterization (IISWC)*.

[61] Sujith Ravi. 2017. On-Device Machine Intelligence. https://ai.googleblog.com/2017/02/on-device-machine-intelligence.html.

[62] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision* 115, 3 (2015).

[63] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *Proc. 3rd International Conference on Learning Representations (ICLR)*.

[64] Siri Team. 2017. Hey Siri: An On-device DNN-powered Voice Trigger for Apple's Personal Assistant. *Apple Machine Learning Journal* 1, 6 (2017).

[65] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A. Alemi. 2017. Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. In *Proc. 31st AAAI Conference on Artificial Intelligence (AAAI)*.

[66] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going Deeper with Convolutions. In *Proc. 28th IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[67] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. 2016. Rethinking the Inception Architecture for Computer Vision. In *Proc. 29th IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[68] Vincent Vanhoucke. 2012. Speech Recognition and Deep Learning. https://ai.googleblog.com/2012/08/speech-recognition-and-deep-learning.html.

[69] Tien-Ju Yang, Andrew Howard, Bo Chen, Xiao Zhang, Alec Go, Vivienne Sze, and Hartwig Adam. 2018. NetAdapt: Platform-Aware Neural Network Adaptation for Mobile Applications. https://arxiv.org/abs/1804.03230

[70] Xiao Zeng, Kai Cao, and Mi Zhang. 2017. MobileDeepPill: A Small-Footprint Mobile Deep Learning System for Recognizing Unconstrained Pill Images. In *Proc. 15th ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*.