Co-design of deep neural nets and neural net accelerators for embedded vision applications

A. Amid
K. Kwon
A. Gholami
B. Wu
K. Asanović
K. Keutzer

Deep Learning is arguably the most rapidly evolving research area in recent years. As a result, it is not surprising that the design of state-of-the-art deep neural net models often proceeds without much consideration of the latest hardware targets, and the design of neural net accelerators proceeds without much consideration of the characteristics of the latest deep neural net models. Nevertheless, in this article, we show that there are significant improvements available if deep neural net models and neural net accelerators are co-designed. In particular, we show that a co-designed neural net model can yield an improvement of 2.6/8.3 × in inference speed and 2.25/7.5 × in energy as compared to SqueezeNet/AlexNet, while improving the accuracy of the model. We also demonstrate that a careful tuning of the neural net accelerator architecture to a deep neural net model can lead to a 1.9–6.3 × improvement in inference speed.

Introduction

For the past several decades, software application design and hardware design have been mostly abstracted from each other through general-purpose processors and programmable computing. Nevertheless, with the decline of Moore's law, systems-on-chip (SoCs) and specialized accelerators are requiring tighter integration between applications and hardware. With the evolving accuracy of deep neural nets (DNN), there is an increasing number of applications utilizing them for a variety of purposes. In particular, DNNs provide the preferred solution for most problems in computer vision. However, while the architectural design and implementation of accelerators for neural nets (NNs) and the design of novel DNN models are very popular topics, a review of literature in these areas indicates that few efforts have broken down the barriers to achieve real co-design.

In particular, hardware architectures and their circuit implementations are routinely evaluated on old or large DNN models, whose fat (in model parameters) and shallow (in layers) architectures bear little resemblance to typical DNN models for computer vision application. Compounding these deficiencies, these DNN-model/NN-accelerator pairs are evaluated on very modest-sized datasets such as MNIST [1] or CIFAR [2]. Furthermore, DNN accelerator designs do not always differentiate between training and inference in the cloud and inference at the edge, even though they impose very different constraints in terms of power and speed. Important application-related factors such as batch size (i.e., the number of data elements that are simultaneously processed) are routinely omitted from the analysis of such accelerators. Alternatively, when they are stated, batch-size values are given, which do not correspond well to any natural application domain. As a result, the utility of many of these NN accelerators on real application workloads is largely unproven.

At the same time, the design of DNN models principally focuses on accuracy on target benchmarks, with little consideration of speed and even less of energy. Moreover, the implications of DNN design choices on hardware execution are not always understood. Unfortunately, even those computer vision papers that do consider the speed of DNN computations often simply use a static calculation of the number of computations performed by a DNN [e.g., Multiply-and-Accumulations (MACs)] as a proxy for

Digital Object Identifier: 10.1147/JRD.2019.2942284

speed. However, as this article and other works [3, 4] show, this proxy is clearly not enough when designing DNNs or NN accelerators.

Thus, a significant gap exists between state-of-the-art NN accelerator design and state-of-the-art DNN model design. In this article, we simply present the results of a coarse-grain co-design approach for closing the gap and demonstrate that a careful tuning of the accelerator architecture to a DNN model can lead to a 1.9–6.3 × improvement in speed in running that model. We also show that integrating hardware considerations into the design of a neural net model can yield an improvement in accuracy, as well as improvements of 2.6 × in speed and 2.25 × in energy as compared to SqueezeNet [5], and, for purposes of comparison, improvements of 8.3 × and 7.5 × compared to AlexNet [6].

Embedded computer vision applications and their constraints

Typical computer vision applications running on embedded processors at the edge include surveillance, cell phone application with computer vision elements, and perception in autonomous driving.

Despite the variety of computer vision applications, there are a few basic primitives or kernels out of which these applications are built. For perception tasks where the goal is to detect and understand the environment, the most common tasks include image classification, object detection, and semantic segmentation, as shown in **Figure 1**. Looking at this figure from left to right: image classification (a) implicitly presumes there is a principal object in the image and aims at assigning one label from a fixed set of categories. A typical DNN model takes an image as input and computes a fixed-length vector as output. Each element of the output vector encodes a probability score of a certain category reflecting the likelihood of the presence of that category somewhere in the image. Object detection (b) places bounding boxes around objects in an image that belong to a list of pre-defined classes. Finally, in semantic segmentation, (c) each pixel in the image is classified.

The precise implementation constraints for an embedded computer vision application can vary widely, even for a single application area, such as surveillance or autonomous driving. To narrow our scope further, in this article, we are particularly concerned with the design problems for computer vision applications that run on in a limited form-factor, on battery power, and with no special support (such as fans) for heat dissipation but, nevertheless, have real-time latency constraints.

Altogether, coupling application constraints with constraints on form-factor, packaging, and battery life results in a number of recurrent constraints in embedded vision applications: accuracy, speed, power, energy, and cost.

Figure 1

Illustration of three fundamental computer vision tasks. (a) Image classification. (b) Object detection. (c) Semantic segmentation.

Accuracy: It is obvious that for a surveillance system or the perceptual system of an autonomous vehicle to be useful, it must achieve a certain accuracy. As there is no *a priori* definition of accuracy, accuracy in computer vision is inevitably linked to testing relative on a benchmark. The most widely used benchmarks are ImageNet [7] for image classification, COCO [8] for object detection, and CityScape [9] for semantic segmentation. There are a number of open-source benchmarks that are more tailored for particular applications, such as the KITTI [10] benchmark for object detection and semantic segmentation in autonomous driving. To be relevant, the population of the variety of objects of the images in the dataset together with their image resolution and general quality must be matched with the target application. A full consideration of this topic is beyond the scope of this article.

Speed: Even a car traveling only 30 miles per hour travels 44 ft in 1 second. To navigate accurately in traffic, the environment surrounding a car must be kept up to date. It is easy to see how this, in turn, translates into speed requirements on the processing of images from a video stream that originates from each camera in an autonomous vehicle and elucidates why accuracy on processing a still image is useless if it does not keep up with the stream of images. This translates into hard constraints, such as keeping up with a camera speed of 30 frames/s (i.e., one frame about every 33 ms). While other embedded applications, like cell phone apps, may appear to have much softer speed constraints, for a product developer of a cell phone app, the impatience of an adolescent translates into just as hard a constraint on application development as the relative speed of a car.

Power: As we will consider energy separately, the principal driver of power constraints is the requirement that the electronics running the embedded vision application do not require a fan or other active cooling device. This puts a hard limit on a typical embedded computer vision system of 5–10 W or less.

Energy: In any embedded vision system that runs on battery power, even an electric vehicle, the energy for computer vision applications will contend with other functions for battery life and must not degrade the user experience.

Cost: The cost of an integrated circuit is directly related to the die size. Limits on cost get most directly translated into limits on the number of accelerator units and amount of on-chip SRAM.

In addition to the common constraints described above, embedded applications pose a number of particular constraints. Many of these motivate the use of embedded vision solutions rather than cloud-hosted solutions. These include the following.

Always-on reliability: Functioning at all times without links to the cloud.

Privacy: The ability to process data locally without connecting to the cloud.

Limits to application size: In September 2017, Apple iOS11 imposed a hard limit of 150 MB on the size of downloads over a cellular network from its App Store. As a result, most applications try to keep their application size within this limit. As this is for the entire application, individual computer vision capabilities within an application must be much smaller. This, in turn, puts hard limits on the size of any DNN in the application.

Taken altogether, the set of constraints described in this section makes the running of an embedded vision application on an embedded processor or accelerator a familiar system design problem: The embedded vision system must achieve hard constraints on accuracy, speed, and power, and try to optimize total energy consumed and memory footprint. In the remainder of this article, we will consider best practices in the co-design of DNNs and NN accelerators to achieve these constraints.

Meeting the constraints in DNN models

Most of those DNN model designers that are concerned about efficiency at all seek simple hardware-agnostic measures to estimate the speed, energy, and power of the DNN model. In this section, we will describe several of these popular metrics.

MACs and FLOPs: Given that DNNs are known to be computationally intensive, a primary goal for optimizing the DNN structure at the model level is to reduce the computational complexity while maintaining a satisfactory accuracy. Without considering specific NN accelerators, the design of efficient DNN usually aims to reduce the computation, which is measured by the number of FLoating-point OPerations (FLOPs) or MACs. The implicit assumption is that there is a monotonic relationship between the number of computational operations (MACs or FLOPS) to overall speed, energy, and power values that constrain embedded applications.

Number of model parameters: The size of a DNN model is routinely given by the number of model parameters or weights that describe the model. These model parameters need to be stored in memory, and therefore affect the size and power consumed by the memory.

Limitations in evaluation of efficient DNN models

We will now describe some limitations of the previously mentioned hardware-agnostic proxies.

MACs and FLOPs

Recall that the use of MACs as a proxy or estimate for the complexity of a DNN model is common practice in computer vision research. Nevertheless, it assumes some ideal model of a computer, which is not the case in practice, as the ratio of arithmetic unit latency to memory access latency is highly skewed.

Modern processors employ various techniques such as parallelism-exploitation, memory hierarchies, and prefetching in an attempt to mitigate this issue. These techniques are implemented differently on different processor microarchitectures, and therefore exploiting the maximum available arithmetic capacity of a processor requires thorough understanding of these factors.

The theoretical peak computing power of a processor is computed based on the capabilities of its arithmetic units. However, the use of MACs or FLOPS as a performance measure assumes that the DNN inference speed depends only on the peak computing power of the hardware, which implicitly assumes that all computing units are active. Depending on the type and configuration of the DNN layer and the hardware architecture, the same theoretical FLOPs measurement can result in different execution times. The work in [11] measured the actual execution time of various convolution filters widely used in convolutional neural network (CNN) DNNs in CPU/GPU and showed a significant difference between the theoretical computational complexity and the actual execution time.

Some of the hardware factors that cause the difference in the actual execution time include the bandwidth limit of each level on the memory hierarchy and the utilization rate of the computing units. Other factors related to this nonideality that can be impacted by the neural network model include the degree of the parallelism and data reusability of the DNN layer.

These effects are further exacerbated with the introduction of modern dynamic DNN models. Some modern network models [12] employ layers that require dynamic control flows—i.e., execution of a layer depending on the outcome of other components of the network (for instance residual connections). These network properties will significantly affect overall run-time since control flow impacts the amount of parallelism that can be extracted from the network. In GPUs, this type of control divergence may double the run-time of a kernel. Control divergence has also been a longtime challenge in CPU microarchitecture, where branch prediction and speculative execution are well-researched areas.

Inaccurate run-time estimation is likely to indicate inaccurate energy consumption estimation as well.

Table 1 Various computer vision DNN building blocks and their computational characteristics.

	Parameter size	FLOPs	Memory Footprint	Arithmetic Intensity
Spatial convolution	MND_K^2	$MND_K^2D_F^2$	$D_F^2(M+N)+D_K^2MN$	$\frac{MND_K^2D_F^2}{D_F^2(M+N)+D_K^2MN}$
Factorized spatial convolution	MND_K	$MND_KD_F^2$	$D_F^2(M+N)+D_KMN$	$\frac{MND_KD_F^2}{D_F^2(M+N)+D_KMN}$
Pointwise convolution ($D_K=1$)	MN	MND_F^2	$D_F^2(M+N)+MN$	$\frac{MND_F^2}{D_F^2(M+N)+MN}$
Group convolution	MND_K^2/G	$MND_K^2D_F^2/G$	$D_F^2(M+N)+D_K^2MN/G$	$\frac{MND_K^2D_F^2/G}{D_F^2(M+N)+D_K^2MN/G}$
Depthwise convolution ($M=N=G$)	MD_K^2	$MD_F^2D_K^2$	$2D_F^2M+D_K^2M$	$\frac{MD_F^2D_K^2}{2D_F^2M+D_K^2M}$
Channel shuffle	0	0	$2D_F^2M$	0
Shift convolution	MN	MND_F^2	$D_F^2(M+N)+MN$	$\frac{MND_KD_F^2}{D_F^2(M+N)+D_KMN}$
Channel aggregation *	$(\sum_i M_i)ND_K^2$	$(\sum_i M_i)ND_K^2D_F^2$	$D_F^2(\sum_i M_i+N)+(\sum_i M_i)ND_K^2$	$\frac{(\sum_i M_i)ND_K^2D_F^2}{D_F^2(\sum_i M_i+N)+(\sum_i M_i)ND_K^2}$

*M denotes the number of input channels, N denotes the number of output filters, D_k denotes the kernel size, D_F denotes the input and output tensor's spatial dimension. *M_i denotes the input channel size from the ith layer.*

According to [13], much of the energy consumption in DNN inference is a result of data movement. Therefore, energy consumption is related to the amount of data and the manner in which it is used in processing the DNN, rather than only the pure computation (MACs or FLOPs). Furthermore, most hardware uses the aforementioned hierarchical memory structure in which energy consumption tends to be higher in interactions with the outer level (DRAM) and low in interactions with the inner level (register file). Thus, the proportion of the outer memory accesses is another important factor in evaluating energy consumption of neural network models.

It is therefore clear that a simple estimation of MACs or FLOPS cannot serve as an accurate proxy to energy or speed constraints. We corroborate this conclusion during the evaluation of SqueezeNext, in which we found multiple cases where a larger number of MACs resulted in reduced run-time and energy consumption, as seen in [14, Table 3].

Model parameters

Some additional pitfalls are related to the estimation of the model's memory footprint. Many DNN research publications put emphasis on the number of model parameters, while leaving out the importance of activation size on hardware performance. A notable example is DenseNet-type connections [15], which have been shown to reduce model parameters, but result in larger feature map size at the end of each *Dense* block, which can become a performance bottleneck. Another example is SqueezeNet-v1.1 which has the same number of parameters as SqueezeNet-v1.0 [5], but its pooling layers' position is placed earlier in the network to reduce the activation size. This simple change resulted in significant gains in speed and energy consumption [14].

Arithmetic intensity

Arithmetic intensity offers a more sophisticated proxy for estimating the speed and energy of DNN models because it integrates computation and memory access in a single model. Arithmetic intensity is generally described as the ratio between computation and memory traffic, and it is measured by the number of operations that are performed for each byte fetched from memory [OPs/Byte]. This is a very useful metric since it was designed to be independent from the hardware microarchitecture and depends only on the implementation of an algorithm or program. Intelligent DNN design entails balancing memory accesses and computation and modeling them by means of the roofline model [16]; however, we should not stop at estimating the arithmetic intensity of the model as a whole. Instead, as we see in **Table 1**, each of the various layer types typically used in DNN models has different values for arithmetic intensity.

Nevertheless, even arithmetic intensity has its limitations with respect to accurate estimation of speed and power. One can estimate the arithmetic intensity of a DNN model as the ratio of FLOPs to activation and parameter footprint. However, for the case of finite-size register file, the actual arithmetic intensity that determines the performance will depend on memory hierarchy, which includes the sizes and bandwidths of various levels of local memory. This requires analysis using non-hardware-agnostic operational intensity, which considers the total memory traffic. Another proposed method to analyze the performance for these cases is Execution-Cache-Memory [17].

Beyond hardware-agnostic evaluation

When designing for efficient embedded application, a DNN designer must understand all factors that can impact the overall performance, area, and energy efficiency. These may include the following.

1) Hardware: the number of computation units, inter-processor connectivity pattern, data type, memory hierarchy (e.g., size, bandwidth, latency, data reuse).

2) DNN: memory footprint, the number of operations, parallelism, data reusability.
3) Software: machine learning framework, scheduling, layer tiling, layer fusion

Therefore, we must also consider non-hardware-agnostic evaluation of a DNN along with a target microarchitecture. This can be done using a variety of tools: evaluation on actual processors, architectural simulators, microarchitectural simulators, etc.

Meeting design constraints with NN accelerators

The power, energy, and speed constraints for embedded vision applications discussed in previous sections naturally motivate the most efficient computing platforms to realize those constraints. There are indeed standard processors available today that are able to operate within these embedded constraints. However, our experience as DNN designers indicates that it is easier to achieve higher accuracy with deeper DNN models that use more computing power, i.e., use more MACs. Moreover, with a higher MACs/Watt ratio, it is easier to fit that computation within a fixed thermal design point. These considerations naturally motivate the investigation into neural net accelerators that are faster and more power-efficient than CPUs or GPUs. It has been shown that the energy overhead resulting from the programmable nature of processors can be on the order of 100 × [18] relative to specialized hardware executing the same function. This is a result of the energy cost of instruction fetching from on-chip and off-chip memory, register file accesses, and control path logic. This programmable overhead is also reflected in run-time performance, as program run-time is constrained by the number of instructions per program and the number of instructions processed per cycle. Other factors that are enabled by specialization, such as precision reduction and dataflow pipelines, provide additional energy and performance improvements. As a result, specialization in the form of dedicated accelerators has been found to be a useful tool in mitigating these previously mentioned overheads and improving energy efficiency.

Specifically, in the domain of deep neural networks, specialized hardware accelerators have been demonstrated by a number of researchers to provide various degrees of energy efficiency improvements. It is difficult to perform an exact comparison between different accelerators and processors due to the large number of variables. Nevertheless, the TPU [19] presented 30 × energy efficiency improvement compared to the Nvidia K80 GPU, and an 80 × energy efficiency improvement compared to an Intel Haswell CPU. Furthermore, using the power and energy efficiency measurements presented in other research prototypes [13, 20–22], and comparing them to recent top energy efficiency CPU rankings (November 2017 Green 500 list [23, 24] at 17 GFLOPs/W, and the Q1 2018 *SPECpower_ssj2008* [25] CPU results at 12.8 ssj_ops/W), we can estimate energy efficiency improvements on the order of 1000 × compared to state-of-the-art CPUs.

The typical approach to microarchitectural design of accelerators is to find a representative workload, extract characteristics, and tailor the microarchitecture to that workload [26]. An alternative approach proposed in an evolving series of research papers [27–29] is to first define the key structural and computational patterns of the application, and then use those to drive the architectural and microarchitectural design. Inference in DNNs has a simple feed-forward pipe-and-filter pattern in which each filter subscribes to the linear-algebra pattern. Of course, conventional CPU architectures can support this pattern, but if we want to improve speed with an accelerator, a natural approach is to unroll the computation spatially as much as we can within the limits of our hardware/silicon allocation. Unrolling the computation in two dimensions quickly results in envisioning a systolic [30] microarchitecture for a NN accelerator. So, next, to define details of the dataflow in the systolic architecture, we need to consider the alternatives. As can easily be seen in [31], systolic architectures or, more recently, spatial architectures (e.g., [13]) are a class of accelerator architectures that exploit the high computational parallelism using direct communication between an array of relatively simple processing elements (PEs). Compared to SIMD architectures, spatial architectures have relatively low on-chip memory bandwidth per PE, but they have good scalability in terms of routing resources and memory bandwidth. Convolutions constitute 90% or more of the computation in DNNs for embedded vision, which are therefore called CNN. Thanks to the high degree of parallelism and data reusability of the convolution, these systolic or spatial architectures are a natural option for accelerating these CNN/DNNs [13, 19, 22, 32, 33]. Hereafter, we restrict the type of NN accelerators we consider to these systolic/spatial architectures and follow the current convention of referring to them as spatial architectures.

Spatial NN accelerators can be generally characterized by the data format and precision of their PE (log, linear, floating point, etc.), the structure of the spatial PE array (size, interconnect topology, data-reuse, etc.), the memory and buffer hierarchy (SRAM, eDRAM, unified [13] versus dedicated [19] configurations, etc.), and additional custom features such as compression and sparsity exploitation [34, 35]. Each of these characteristics has an impact on the previously mentioned application constraints for NN accelerators.

In order to exploit the massive parallelism, NN accelerators contain a large number of PEs that run in parallel, each of which typically consists of a MAC unit and

a small buffer or register file for local data storage. Many accelerators employ a two-dimensional (2-D) array of PEs, ranging in size from as small as 8×8 [32] to as large as 256×256 [19]. However, there is a direct relationship between the number of PEs in a spatial array and the required memory bandwidth of the accelerator. Thus, NN accelerators provide several levels of memory hierarchy to provide data to the MAC unit of a PE, and each level is designed to take advantage of the data reuse of a convolutional layer to minimize access to the upper level. This hierarchy includes global buffers (on-chip SRAMs) ranging from tens of kilobytes to tens of megabytes, interconnections between PEs, and local register files in each PE. The memory hierarchy and the data reuse scheme are one of the most important features that distinguish NN accelerators. It is worth noting that some accelerators also have dedicated blocks to process NN layers other than convolutional layers [22, 32, 33]. However, since these layers have a very small computational complexity, they can usually be processed in a one-dimensional SIMD manner.

Limitations in modeling and evaluation of NN accelerators

We have previously mentioned that common design considerations of DNN models, such as MACs or model parameters, are not always correlated to actual application requirements such as inference run time or energy consumption. However, hardware NN accelerator design is also limited in the scope of its application level considerations, and the metrics for the evaluation of NN hardware accelerators are many times equally inconsistent.

Evaluation measures

The OPS/Watt metric has emerged as the figure-of-merit of choice in the NN accelerator implementation community. As an example, the 2017–2019 sessions of the International Conference on Solid State Circuits presented NN accelerators ranging from 2.1 to 140.3 TOPS/Watt [20–22, 36–41] while highlighting reduced precisions of only 1–16 bits and sparsity exploitation in pursuit of energy efficiency. Similarly, the 2017 and 2018 Symposia on VLSI Circuits published energy efficiency figures ranging from 2.3 to 765.6 TOPS/Watt [42–48], using a variety of reduced-precision techniques.

The OPS/Watt metric is a corollary to the popular FLOPS/Watt metric used for the evaluation of computer performance and energy efficiency. However, while FLOPS is a relatively well-defined metric with regards to the specification of the operations (single precision or double precision floating point operations), the "OP" used in neural net accelerator evaluation is not as well defined, a phenomenon clearly visible in the wide range of published results. While evaluations on the OPS/Watt figure-of-merit show orders of magnitude of improvement year after year, many of these DNN accelerators use popular reduced precision and pruning techniques which affect the accuracy and run-time of DNN models. In this context, presenting OPS/Watt figures without associated accuracy, latency, and supporting memory-system details renders these comparisons as highly controversial.

Furthermore, if an accuracy sensitivity analysis is performed in the accelerator implementation community, it is highly different than an accuracy sensitivity analysis performed in the ML community. This is because the reference design and DNN architecture of choice for evaluation in a large portion of hardware accelerators works [21, 22, 36, 40, 45, 47] is AlexNet [6], a network architecture that is considered by many as "past its prime." Alternatively, accelerators are evaluated on unspecified demonstration versions of CNN/RNN [19, 49], other large network architectures such as VGG [50], or using simple datasets such as MNIST or CIFAR-10 [37, 40, 43, 44]. While AlexNet was indeed a significant breakthrough in the ability of DNNs to obtain good accuracy on the ImageNet dataset, the computer vision community has since advanced to more sophisticated network architectures. Some popular modern DNN features such as residual layers or small convolutions inherently break dataflow and parallelism assumptions between layers, which may impact inter-layer data reuse utilized in many NN accelerators. While it is natural for hardware accelerator implementations to "lag" behind fast-paced ML advancements, evaluation on 6-year-old network architectures lacks the necessary information and tools to guide ML researchers.

Application context for batching

Spatial NN accelerators are designed to exploit parallelism using parallel PEs. As such, the availability of input-parallelism is important for the utilization of throughput-oriented processors and spatial NN accelerators. Exploitation of data parallelism in DNN training is effectively achieved by batching several training elements into single optimizer steps (commonly Stochastic Gradient Descent) for updating weight values. This type of input-data parallelism can also be exploited in certain inference scenarios with a high volume of input data (such as in data-centers, or video and signal processing on edge devices). However, in other inference scenarios on edge devices, there is not necessarily a large enough set of data to be processed in "batches." In an example scenario of an Internet of Things (IoT) device capturing an image once every several minutes to hours [51] and using a DNN to detect and identify objects, batching several images for inference can be irrelevant. Latency-sensitive applications are also limited in the amount of batching that can be used for inference. These types of inference engines can exploit only layer-level parallelism in the DNN model. Batching effectiveness can also be limited by device memory

capacity and energy constraints. In models that do not fit in on-chip memory, batching may indeed amortize the cost of data transfer between on-chip and off-chip memory. However, in cases where an entire neural net model fits in on-chip memory, it may be beneficial to reduce batch sizes, hence not requiring the eviction of model weights to off-chip memory. Recent calls by prominent machine learning researchers have also emphasized the need for smaller batch sizes in hardware DNN processing [52].

Feature map versus model sizes impact on memory design

When discussing memory utilization, we must also consider the differences in input sizes between different applications. While images in the ImageNet data set vary in size and resolution [7], a common preprocessing stage is to sample the image to 256×256 pixels, resulting in an image size of $\sim$196 KB. However, while the ImageNet benchmark was designed for image classification, newer DNN application domains such as autonomous driving require processing of higher-resolution images for detection and classification of smaller objects. These applications have different memory design considerations than current models, which are dominated by model parameter sizes. Many NN accelerators are designed to be able to fit specific models within on-chip memory. Some DNN models are good fits for these types of accelerators, such as when using SqueezeNet on the ImageNet dataset (385 KB for the larger layer). Nevertheless, when using high-resolution images (10 mega-pixels), memory utilization is dominated by feature maps (4 MB), and therefore the small number of model parameters of the SqueezeNet model is irrelevant. In such cases, given an unfit memory hierarchy, design energy will again be dominated by memory communication.

Mismatches between embedded DNNs and NN accelerators: The need for co-design

As mentioned in the previous section, many earlier works on NN accelerators are designed and/or evaluated using relatively old network architectures such as AlexNet or VGG on simple datasets. Convolutional layers with filter sizes of 3×3 or more are predominant in these networks. However, recent trends show that the 1×1 convolution has become an essential building block for efficient DNNs [5, 14, 15, 53–55]. Furthermore, other lightweight DNNs such as MobileNet [56] and ShuffleNet [57], which specifically target mobile and embedded systems, replace the 3×3 convolutions with a combination of a pointwise convolutions and a depthwise convolutions. These building blocks enable the implementation of lightweight DNNs in terms of computational complexity and number of parameters, but have a lower degree of parallelism and data reuse, which may degrade speed and energy efficiency on custom hardware due to the lack of inter-channel data transfers.

Eyeriss [13] proposed a useful taxonomy that classifies NN accelerators according to the type of data each PE locally reuses. Since the degree of data reuse increases as the memory hierarchy goes down, this type of classification can be used to represent the characteristic reuse scheme of NN accelerators. Four dataflows are presented in [13]: weight stationary (WS), output stationary (OS), row stationary, and no local reuse. This work will address two of them.

Weight stationary: The WS dataflow is designed to minimize the required bandwidth and the energy consumption of reading model weights by maximizing the accesses of the weights from the register file at the PE. The PE preloads a weight of the convolution filters to its register and then performs MAC operations over the entire input feature map. In a weight stationary dataflow, the result of each MAC operation is sent out of the PE in each cycle. TPU [19] is one such example of a weight stationary dataflow, which uses a general matrix-vector multiplier computation mapping.

Output stationary: The OS dataflow is designed to maximize the accesses of the partial sums within the PE. In each cycle, the PE computes parts of the convolution that will contribute to one output pixel and accumulates the results within the PE accumulator. Once all the computations for that pixel are finished, the final result is sent out of the PE, and the PE moves to work on a new pixel. One example of an early OS dataflow architecture is ShiDianNao [32], which maps a 2-D block of the output feature map to the PE array. Using a mesh interconnect topology, the corresponding weight is broadcasted to all PEs every cycle.

As domain-specific architectures, NN accelerators are designed to take advantage of data reusability inherent in the convolution layer. The type of data reused by different accelerators depends on the underlying dataflow (output stationary and weight stationary). Therefore, in situations where only a certain kind of data reuse pattern exists in a DNN model, accelerators implemented using different dataflows may exhibit different acceleration performance. We examine performance estimation results of various configurations of a convolutional layer on hypothetical WS and OS spatial dataflow architectures. The two reference accelerators consist of a 32×32 PE array, a 512-KB global buffer, and a DMA controller. The PE array can read 32 input activations, 32 weights, and 32 partial sums (if needed) from the buffer and can write 32 output activations to the global buffer every cycle. In the array, only connections between adjacent PEs are provided. We assume that data are represented as 16-bit integers, and all data can be double buffered in the global buffer (given the well-defined structure of a convolutional layer). In the WS

setting, we assume that the input data of the PE array is in the channel-major format. In the OS setting, each PE has eight accumulator registers to reuse the input feature map across eight different kernels. The effective DRAM bandwidth is assumed to be 16 GB/s.

We evaluate convolutional filters of sizes 1×1, 3×3, 5×5, as well as 3×3 and 5×5 depthwise convolutional filters, all with an input batch size set to one. We learn from our analysis that for standard 3×3 and 5×5 convolution filters, the performance of the OS dataflow degrades if the size of the feature map is smaller than 32×32. Since the OS dataflow maps the output feature map onto the PE array, the array is underutilized if the width and the height of the output feature map are not multiples of the size of the PE array. For example, 87.5% of the PEs are idle during the convolution on 8×8 feature maps. This is common in the processing of ImageNet classification networks, where the spatial dimension of the feature maps is reduced by a plurality of pooling layers.

An additional observation is that 1×1 convolutions do not perform well using an OS dataflow. This is since 1×1 convolution layers do not have data reuse in the filter spatial direction, which is utilized in the OS dataflow inter-PE connections. We also observe that for WS dataflows, the PE array may be underutilized in the case of a small number of input or output channels to a layer. This is since a simple mapping of a WS dataflow maps input channels and output channels to the horizonal and vertical directions of the PE, respectively, which causes underutilization of the array when processing a small number of channels. A representative example of this scenario is the first layer of image processing networks, whose input consists of only the three channels of an RGB or YCbCr image. Finally, in both dataflows, depthwise convolutions exhibit a large gap between the theoretical computational complexity ("MACs") and inference speed. This is due to low data reuse in the OS dataflow and low PE utilization in the WS dataflow. Nevertheless, the impact of the low data reuse in the OS dataflow has a more significant effect on the speed of depthwise convolution layers, making WS a better choice for such layers.

Recent work by Yang et al. [58] presents alternative evidence and conclusions regarding the importance of the dataflow taxonomy in relation to other components of a system architecture. Yang et al. emphasize the importance of a properly sized memory hierarchy and program tiling for energy consumption of DNN accelerators. Furthermore, they make specific suggestions of memory hierarchies consisting of multilevel register files, echoing the small register file size choices in [59]. We agree with the results of [58] that indicate a properly architected memory hierarchy is fundamental to speed and energy efficiency of a DNN accelerator, and we considered this in our design presented in [59]. Nevertheless, once a memory hierarchy is properly architected, we contend that there are still benefits to dataflow (albeit, at smaller orders of magnitude), both in inference speed and energy. Based on our understanding from communications with the authors of [58], the energy improvements presented in [59] are consistent with measurements in their work.

Co-design of DNNs and NN accelerators

In this section, we describe the co-design of DNNs and NN accelerators. Because the design of either a DNN or NN accelerator is a significant enterprise, the co-design of these is necessarily a coarse-grained process. Thus, we first describe the design of the Squeezelerator, a NN accelerator intended to accelerate SqueezeNet. We then continue with a discussion of the design of SqueezeNext, a DNN designed with the principles described in [60] and particularly tailored to execute efficiently on the Squeezelerator. Finally, we discuss the additional tune-ups of the Squeezelerator for SqueezeNext.

Based on the analysis of previous experimental results, we classify convolution layers into four categories: the first convolutional layer (i.e., the first convolutional layer of the DNN), pointwise convolution (also known as 1×1 convolution), $F \times F$ convolution (where $F > 1$), and depthwise convolution (DW). From the previous analysis, we conclude it is advantageous for 1×1 convolutional layers to be accelerated by using the WS dataflow, while the first convolutional layer and the depthwise convolutional layers would preferably use the OS dataflow. Our simulations indicate that 1×1 convolutional layers are 1.4–7.0$\times$ faster on a WS dataflow architecture than on an OS dataflow (depending on the size of the feature map and the number of channels). In contrast, relative to the WS dataflow architecture, the first convolutional layer is 1.6–6.3$\times$ faster on the OS dataflow architecture, and the depthwise convolutional layers are 19–96$\times$ faster on the OS dataflow architecture. In the case of the normal $F \times F$ convolutions (in particular, 3×3 convolutions), various factors such as the size of the feature map and the sparsity of the filters affect the preferable choice of dataflow.

Table 2 shows the relative percentage of computation devoted to each layer type in a variety of DNNs. There is a large variation in the percentages for each category over these DNN models, and as a result, the proportion of the layer operations that are well suited to the WS dataflow ranges from 0% to 95%. While initially focused on supporting SqueezeNet, this layer analysis led to the key design principal of the Squeezelerator: To achieve high efficiency for the entire DNN model, the accelerator architecture must be able to choose WS dataflow or OS on a layer-by-layer basis. Thus, the design of the Squeezelerator is based on the layer-by-layer simulation as described previously. As shown in **Figure 2**, the Squeezelerator consists of a PE array, a global buffer, a preload buffer, a

Table 2 Relative percentage of MAC operations of specific layer types out of the total number of MAC operations of a DNN model.

Network	Conv1	1×1	$F \times F$	DW
AlexNet	20%	0%	69%	0%
1.0 MobileNet-224	1%	95%	0%	3%
Tiny Darknet	5%	13%	82%	0%
SqueezeNet v1.0	21%	25%	54%	0%
SqueezeNet v1.1	6%	40%	54%	0%
SqueezeNext	16%	44%	40%	0%

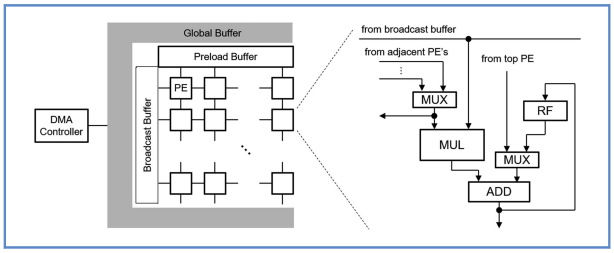

Figure 2

Block diagram of Squeezelerator (left) and PE (right).

stream buffer, and a DMA controller. Intended for SoC deployment, the PE array consists of $N \times N$ PEs (configurable for simulation purposes from $N = 8$ to 32), connected to each other in a 2-D mesh. All PEs are connected to a broadcast stream buffer, while the PEs located at the top and bottom rows of the mesh are additionally connected to the preload buffer and the global buffer, respectively. The preload buffer prepares the data to be transferred to the PE array before the operation starts, and a stream buffer prepares the data to be continuously transferred to the PE array during the operation. The global buffer consists of 128-KB on-chip SRAM and switching logic. Each PE contains a multiplexer for selecting one of several input sources, a 16-bit integer multiplier, an adder for accumulating the multiplication result, and a register file for storing the intermediate result of the computation. In order to support two dataflows, each design includes all the interconnections required for both dataflows. The area overhead is minimized by providing different data to the PE array in each mode. For example, the broadcast buffer provides the input activations in the WS mode, while it provides the weights in the OS mode.

Squeezelerator processes a DNN layer by layer, and it can be configured to select the dataflow style (OS or WS) for each layer with no overhead incurred by switching between dataflow styles. While the accelerator is running in the OS dataflow mode, each PE is responsible for different pixels in the 2-D block of the output feature map. Every cycle, the corresponding input and weight are supplied to each PE through inter-PE connection and from the broadcast buffer, respectively. The operation sequence is as follows: First, a 2-D block of the input feature map is preloaded into the PE array from the preload buffer. Then, the stream buffer broadcasts a weight to all the PEs, and each PE multiplies the input by the weight and accumulates the result in the local register file. For an $N \times N$ filter, this step is repeated N^2 times with different input and weight data. Instead of reading the input from the preload buffer every time, each PE receives the data from one of the neighboring PEs. The whole process is repeated with different input channels. When the computation for the output block is finished, the result of each PE is stored to the global buffer. This final step takes additional processing time. To reduce the energy consumed by the global buffer, PEs reuse each input they receive across different filters. In addition, the stream buffer broadcasts only non-zero weights to reduce the execution time by skipping unnecessary computations.

In the WS dataflow mode, a PE row and a PE column correspond to one input channel and one output channel, respectively. In this way, each PE is responsible for different elements of the weight matrix. Contrary to the OS mode, the weights are preloaded into the PE array. Then, the stream buffer broadcasts pixels from different input channels to the PE array, and each PE multiplies the input by its own weight. Each PE column sums the multiplication results by forming a chain of adders from the top PE to the bottom PE. This process is repeated until all the pixels in the input feature maps are accessed.

A performance estimator evaluates the execution cycles and the energy consumption of Squeezelerator. Results describe inference times of individual images (i.e., batch size = 1) from the ImageNet benchmark suite [7]. As mentioned previously, a batch size of one gives less opportunity for data reuse, but reflects typical usage in embedded vision applications for mobile phones or automotive perception. The time consumed by the PE array and the buffers reflects the micro-architecture, while the DRAM access time is approximated using latency and effective bandwidth (specifically, 100 cycles and 16 GB/s). In order to hide the data transfer time between the DRAM and the global buffer, we used double buffering [61]. If the memory footprint of the layer exceeds the capacity of the buffer, we used tiling for scheduling of the nested convolution loops. We follow the methodology used by [13] for energy estimation, which multiplies the number of MACs and memory accesses by unit energy. During simulation, we conservatively set the sparsity, i.e., the number of zero weights, of each DNN layer at 40%.

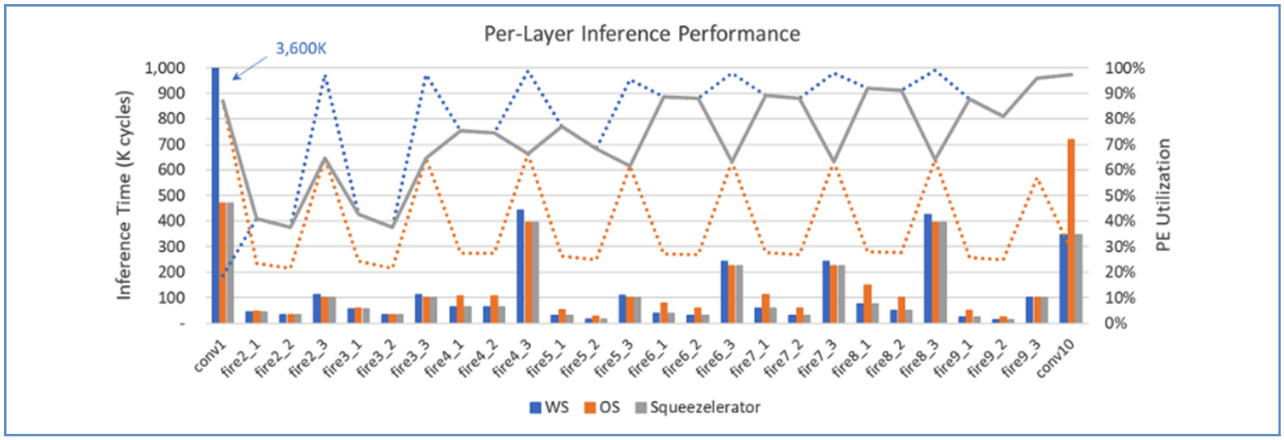

Figure 3

Per-layer inference time (bars) and utilization efficiency (lines) of SqueezeNet v1.0 on the reference WS/OS and Squeezelerator.

We first evaluate Squeezelerator with the target DNN, SqueezeNet v1.0. **Figure 3** shows the inference time and utilization per layer of SqueezeNet v1.0 for the reference OS and WS architectures and the proposed hybrid architecture. The overall trend of the Squeezelerator is similar to that of the WS architecture, but the performance of the first layer is noticeably improved. For most of the 3×3 convolutions, the Squeezelerator chooses to use an OS dataflow. Comparing the total processing time, the hybrid architecture shows performance improvement of 26% and 106% on SqueezeNet v1.0 compared to the reference OS and WS architectures, respectively. **Table 3** shows the performance improvement of the Squeezelerator over the reference architectures on a variety of lightweight DNNs as well as AlexNet (for comparison purposes). The improvement over the OS architecture has a high correlation with the proportion of the 1×1 convolutions in the network. The benefits of supporting two dataflow architectural styles are obvious in the case of MobileNet: Since a naive WS architecture does not efficiently accelerate the depthwise convolutions, these layers occupy much larger execution time than the pointwise convolutional layers, even though they account for only 3% of the total number of computations. At the same time, 1×1 convolutions, which account for 95% of the total computation in MobileNet, greatly reduce the acceleration performance of the OS architecture. Hence, the hybrid architecture achieves about 2× and 6× speedup on MobileNet compared to the OS and WS architectures. AlexNet shows the least performance improvement because it takes up 80% of energy and 73% of its run-time in the three fully connected layers, which cannot take advantage of hardware acceleration by either dataflow architecture. MobileNet shows small savings on the energy consumption relative to its significant performance improvement because DRAM access consumes a larger proportion of total energy consumption in this network than in other DNNs. This is related to the low data reusability of the pointwise convolutions and the depthwise convolutions. The energy reduction of SqueezeNet V1.0 and Tiny Darknet is due to their larger proportion of layers that is suited to the OS dataflow.

Table 3 Speed and energy improvements of squeezelerator over OS or WS architectures.

Network	Speedup		Energy reduction	
	vs OS	vs WS	vs OS	vs WS
AlexNet	1.00×	1.19×	-2%	6%
1.0 MobileNet-224	1.91×	6.35×	8%	6%
Tiny Darknet	1.14×	1.32×	0%	24%
SqueezeNet v1.0	1.26×	2.06×	6%	23%
SqueezeNet v1.1	1.34×	1.18×	8%	10%
SqueezeNext	1.26×	2.44×	0%	20%

We continue with our co-design process and describe SqueezeNext [12], a new family of neural networks for embedded systems, which was designed by performing detailed analysis with the architectural simulator for Squeezelerator. SqueezeNext was designed by studying previous DNN models such as SqueezeNet with the aim of using structure of layers to further reduce model parameters, as well as avoiding MobileNet's depthwise separable convolutions that have poor arithmetic intensity. Studying the hardware utilization of different layers of SqueezeNext on the Squeezelerator revealed that initial layers had low MACs/cycle counts, which noticeably affected hardware performance, as shown in **Figure 4**. One important optimization used in SqueezeNext is a filter size reduction for the first layer from 7×7 to 5×5; this layer has significant impact on inference time as its input feature

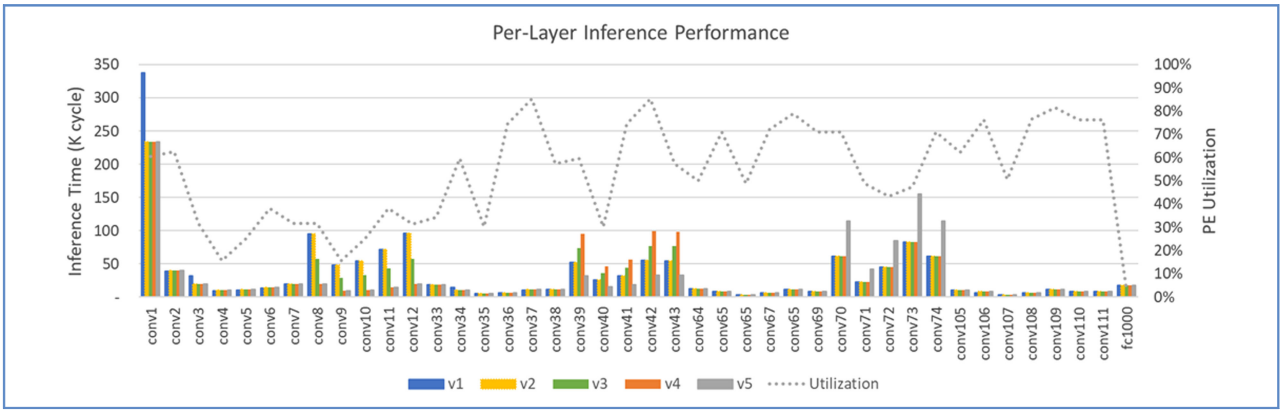

Figure 4

Per-layer inference time (lower is better) is shown along the left *y*-axis for five variants (v1–v5) of 1.0-SqNxt-23 architecture. PE utilization is shown by the dotted line. The initial layers have very low utilization, which adversely affects inference time and energy consumption.

map is relatively large. Another contributing factor to poor hardware utilization is the small number of channels in the initial layers. As indicated earlier, such scenarios result in not all PEs being utilized and limited opportunity for memory latency hiding. One solution to this would be to simply reduce the number of layers early in the DNN; however, a naive reduction may lead to a degradation in accuracy. Instead, we reduce the number of layers early in the DNN and assign more layers to later stages that allow for higher hardware utilization. While this simple change results in a very small change in the overall MACs used in inference, it reduces both energy and inference time [14]. Five different variants of these two classes of optimizations are shown in Figure 4. In fact, the optimized versions have slightly better accuracy as compared to the initial variant.

Following this design of SqueezeNext, we returned to the co-design of the Squeezelerator and fine-tuned the hardware utilization by doubling the number of registers in the register file of each PE from 8 registers to 16. The combination of these optimizations results in SqueezeNext being 2.59 × faster and 2.25 × more energy-efficient than SqueezeNet 1.0 (and 8.26 × and 7.5 × when compared to AlexNet) on the Squeezelerator, without any degradation in accuracy. **Figure 5** shows the spectrum of accuracy versus power and accuracy versus inference time for different DNN families. Ideally, we would like higher accuracy with lower power and inference time. As we can see, the SqueezeNext family using the Squeezelerator provides such favorable solutions, which allows the user to select the right DNN from this family based on the target application's constraints.

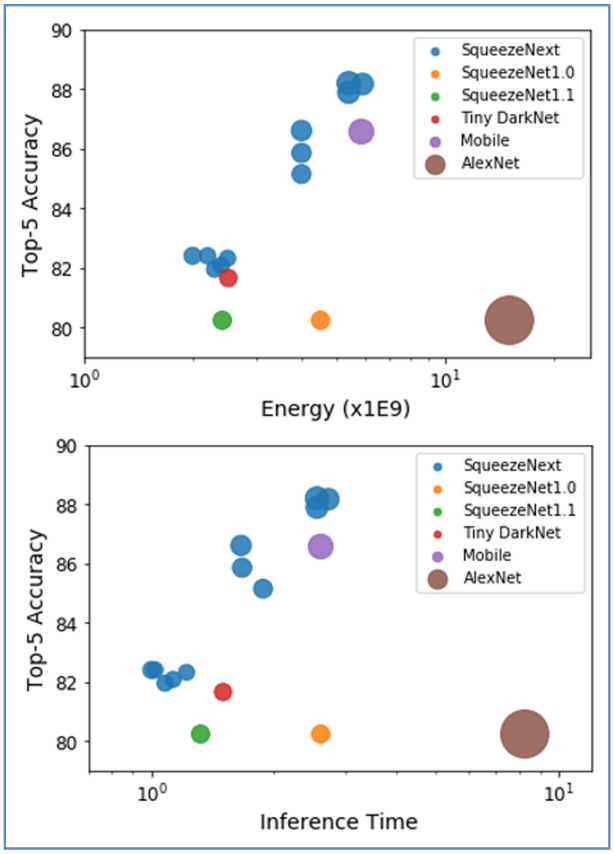

Figure 5

Spectrum of accuracy versus energy and inference speed for SqueezeNext, SqueezeNet (v1.0 and v1.1), Tiny DarkNet, and MobileNet. The size of model (model parameters) is represented by the size of the circle. SqueezeNext shows superior Paretto curves with respect to all three metrics (in both plots, higher and to the left is better).

Conclusion

Embedded vision applications bring power, energy, memory, and speed constraints. In this article, we surveyed

and analyzed the properties and limitations of efficient DNN and NN accelerators related to these constraints. As a result, we also illustrated a coarse-grain co-design approach for the design of DNNs and NN accelerators that meet these constraints. Our efforts at a NN accelerator led to the novel design of the Squeezelerator, which can perform either weight-stationary dataflow or output-stationary dataflow on a layer-by-layer basis. On popular DNNs for mobile applications, this accelerator design is 1.1–6.35 × faster than accelerators that use only a single dataflow architecture. To illustrate the additional value of tailored DNN design to the accelerator, we revisited the design of SqueezeNet and produced the SqueezeNext family: Some members of the SqueezeNext family are 2.26 × faster than SqueezeNet 1.0, improve the energy by 2.25 × , and are more accurate on image classification benchmarks (we achieve 59.2% top-1 versus 57.1% of SqueezeNet) [14]. We completed our design study by then revisiting the design of the Squeezelerator running SqueezeNext. As SqueezeNext has similar layer characteristics to SqueezeNet, only some fine-tuning of register file size was required to optimize local data reuse. Combining the Squeezelerator NN accelerator architecture and the SqueezeNext family of efficient DNNs allows for better design-point selection and proper consideration of the full set of constraints of embedded vision applications.

Acknowledgment

This work was supported by a gracious fund from Intel Corporation and Samsung and by ADEPT Lab affiliates Apple, Futurewei, Google, and Seagate. We would like to thank Intel VLAB team for providing us with access to their computing cluster. We also gratefully acknowledge the support of NVIDIA Corp. with the donation of the Titan Xp GPU used for this research. The views and opinions of authors expressed herein do not necessarily state or reflect those of the sponsors or affiliates.

References

1. Y. LeCun, C. Cortes, and C. J. C. Burges, MNIST Handwritten Digit Database, 2010.
2. A. Krizhevsky, "Learning multiple layers of features from tiny images," Univ. Toronto, Toronto, ON, Canada, Tech. Rep. TR-2009, 2009.
3. J. Huang, V. Rathod, C. Sun, et al., "Speed/accuracy trade-offs for modern convolutional object detectors," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2017, pp. 3296–3297.
4. S. Gupta and M. Tan, "EfficientNet-EdgeTPU: Creating accelerator-optimized neural networks with AutoML," 2019. [Online]. Available: https://ai.googleblog.com/2019/08/efficientnet-edgetpu-creating.html
5. F. N. Iandola, S. Han, M. W. Moskewicz, et al., "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5 MB model size," arXiv: 1602.07360, 2016.
6. A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proc. 25th Int. Conf. Neural Inf. Process. Syst.*, 2012, pp. 1097–1105.
7. J. Deng, W. Dong, R. Socher, et al., "Imagenet: A large-scale hierarchical image database," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2009, pp. 248–255.
8. T.-Y. Lin, M. Maire, S. Belongie, et al., "Microsoft Coco: Common objects in context," in *Proc. Eur. Conf. Comput. Vis.*, 2014, pp. 740–755.
9. M. Cordts, M. Omran, S. Ramos, et al., "The cityscapes dataset for semantic urban scene understanding," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 3213–3223.
10. A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? The Kitti vision benchmark suite," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2012, pp. 2254–3361.
11. "Evaluating efficiency of several types of convolutions," 2017. [Online]. Available: https://github.com/yu4u/conv-benchmark
12. X. Wang, F. Yu, Z.-Y. Dou, et al., "SkipNet: Learning dynamic routing in convolutional networks," in *Proc. Eur. Conf. Comput. Vision*, 2018, pp. 409–424.
13. Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," *ACM SIGARCH Comput. Archit. News*, vol. 44, no. 3, pp. 367–379, 2016.
14. A. Gholami, K. Kwon, B. Wu, et al., "SqueezeNext: Hardware-aware neural network design," in *Proc. 2018 IEEE/CVF Conf. Comput. Vis. Pattern Recognit. Workshops*, Salt Lake City, UT, USA, 2018, pp. 1719–1728, doi: 10.1109/CVPRW.2018.00215.
15. G. Huang, Z. Liu, L. Van Der Maaten, et al., "Densely connected convolutional networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2017, pp. 4700–4708.
16. S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, 2009.
17. J. Hufmann, J. Eitzinger, and D. Fey, "Execution-cache-memory performance model: Introduction and validation," arXiv: 1509.03119, 2015.
18. M. Horowitz, "Computing's energy problem (and what we can do about it)," in *Proc. IEEE Int. Solid-State Circuits Conf.*, 2014, pp. 1–12.
19. N. P. Jouppi, C. Young, N. Patil, et al., "In-datacenter performance analysis of a tensor processing unit," in *Proc. 44th Annu. Int. Symp. Comput. Archit.*, 2017, pp. 1–12.
20. J. Lee, C. Kim, S. Kang, et al., "UNPU: A 50.6 TOPS/W unified deep neural network accelerator with 1b-to-16b fully-variable weight bit-precision," in *Proc. IEEE Int. Solid-State Circuits Conf.*, 2018, pp. 218–220.
21. K. Ueyoshi, K. Ando, K. Hirose, et al., "QUEST: A 7.49TOPS multi-purpose log-quantized DNN inference engine stacked on 96MB 3D SRAM using inductive-coupling technology in 40nm CMOS," in *Proc. IEEE Int. Solid-State Circuits Conf.*, 2018, pp. 216–218.
22. B. Moons, R. Uytterhoeven, W. Dehaene, et al., "Envision: A 0.26-to-10TOPS/W subword-parallel dynamic-voltage-accuracy-frequency-scalable convolutional neural network processor in 28nm fdsoi," in *Proc. IEEE Int. Solid-State Circuits Conf.*, 2017, pp. 246–247.
23. "Green500 List for November 2017," 2017. [Online]. Available: https://www.top500.org/green500/lists/2017/11/
24. W. Feng and K. Cameron, "The Green500 list: Encouraging sustainable supercomputing," *Computer*, vol. 40, no. 12, pp. 50–55, 2007.
25. "First Quarter 2018 SPECpower_ssj2008 Results," 2018. [Online]. Available: https://www.spec.org/power_ssj2008/results/res2018q1/
26. J. L. Hennessy and D. A. Patterson, *Computer Architecture: a Quantitative Approach*. Amsterdam, The Netherlands: Elsevier, 2011.
27. K. Asanovic, R. Bodik, B. C. Cantanzaro, et al., "The landscape of parallel computing research: A view from Berkeley," Univ. California, Berkeley, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2006-183, 2006.
28. K. Asanovic, R. Bodik, J. Demmel, et al., "A view of the parallel computing landscape," *Commun. ACM*, vol. 52, no. 10, pp. 56–67, 2009.

29. K. Keutzer and T. Mattson, "A design pattern language for engineering (parallel) software," *Intel Technol. J.*, vol. 13, no. 3, 2010.
30. H.-T. Kung and C. E. Leiserson, "Systolic arrays (for VLSI)," in *Proc. Sparse Matrix Proc.*, 1979, vol. 1, pp. 256–282.
31. H.-T. Kung, "Why systolic architectures?" *IEEE Comput.*, vol. 15, no. 1, pp. 37–46, Jan. 1982.
32. Z. Du, R. Fasthuber, T. Chen, et al., "ShiDianNao: Shifting vision processing closer to the sensor," *ACM SIGARCH Comput. Archit. News*, vol. 43, no. 3, pp. 92–104, 2015.
33. W. Lu, G. Yan, J. Li, et al., "FlexFlow: A flexible dataflow accelerator architecture for convolutional neural networks," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2017, pp. 553–564.
34. S. Han, X. Liu, H. Mao, et al., "EIE: Efficient inference engine on compressed deep neural networks," in *Proc. 43rd Annu. Int. Symp. Comput. Archit.*, 2016, pp. 243–254.
35. A. Parashar, M. Rhu, A. Mukkara, et al., "SCNN: An accelerator for compressed-sparse convolutional neural networks," in *Proc. 44th Annu. Int. Symp. Comput. Archit.*, 2017, pp. 27–40.
36. G. Desoli, N. Chawla, T. Boesch, et al., "A 2.9 TOPS/W deep convolutional neural network SoC in fd-soi 28nm for intelligent embedded systems," in *Proc. IEEE Int. Solid-State Circuits Conf.*, 2017, pp. 238–239.
37. D. Bankman, L. Yang, B. Moons, et al., "An always-on 3.8 uJ/86% CIFAR-10 mixed-signal binary CNN processor with all memory on chip in 28nm CMOS," in *Proc. IEEE Int. Solid-State Circuits Conf.*, 2018, pp. 222–224.
38. J. Lee, J. Lee, D. Han, et al., "LNPU: A 25.3 TFLOPS/W sparse deep-neural-network learning processor with fine-grained mixed precision of FP8-FP16," in *Proc. IEEE Int. Solid-State Circuits Conf.*, 2019, pp. 142–143.
39. C. Kim, S. Kang, D. Shin, et al., "A 2.1 TFLOPS/W mobile deep RL accelerator with transposable PE array and experience compression," in *Proc. IEEE Int. Solid-State Circuits Conf.*, 2019, pp. 136–137.
40. J. Yue, R. Liu, W. Sun, et al., "A 65nm 0.39-to-140.3 TOPS/W 1-to-12b unified neural-network processor using block-circulant-enabled transpose-domain acceleration with 8.1 × higher TOPS/mm^2 and 6T HBST-TRAM-based 2D data-reuse architecture," in *Proc. IEEE Int. Solid-State Circuits Conf.*, 2019, pp. 138–139.
41. J. Song, Y. Cho, J.-S. Park, et al., "An 11.5 TOPS/W 1024-MAC butterfly structure dual-core sparsity-aware processing unit in 8nm flagship mobile SoC," in *Proc. IEEE Int. Solid-State Circuits Conf.*, 2019, pp. 130–131.
42. S. Yin, P. Ouyang, S. Tang, et al., "A 1.06-to-5.09 TOPS/W reconfigurable hybrid-neural-network processor for deep learning applications," in *Proc. Symp. VLSI Circuits*, 2017, pp. C26–C27.
43. F. N. Buhler, P. Brown, J. Li, et al., "A 3.43 TOPS/W 48.9 pJ/pixel 50.1 nJ/classification 512 analog neuron sparse coding neural network with on-chip learning and classification in 40nm CMOS," in *Proc. Symp. VLSI Circuits*, 2017, pp. C30–C31.
44. K. Ando, K. Ueyoshi, K. Orimo, et al., "BRein memory: A 13-layer 4.2 K neuron/0.8 M synapse binary/ternary reconfigurable in-memory deep neural network accelerator in 65 nm CMOS," in *Proc. Symp. VLSI Circuits*, 2017, pp. C24–C25.
45. Z. Yuan, J. Yue, H. Yang, et al., "Sticker: A 0.41-62.1 TOPS/W 8Bit neural network processor with multi-sparsity compatible convolution arrays and online tuning acceleration for fully connected layers," in *Proc. Symp. VLSI Circuits*, 2018, pp. 33–34.
46. M. Anders, H. Kaul, S. Mathew, et al., "2.9 TOPS/W reconfigurable dense/sparse matrix-multiply accelerator with unified INT8/INT16/FP16 datapath in 14nm tri-gate CMOS," in *Proc. Symp. VLSI Circuits*, 2018, pp. 39–40.
47. S. Yin, P. Ouyang, J. Yang, et al., "An ultra-high energy-efficient reconfigurable processor for deep neural networks with binary/ternary weights in 28nm CMOS," in *Proc. Symp. VLSI Circuits*, 2018, pp. 37–38.
48. B. Fleischer, S. Shukla, M. Jiegler, et al., "A scalable multi-teraOPS deep learning processor core for AI training and inference," in *Proc. Symp. VLSI Circuits*, 2018, pp. 35–36.
49. D. Shin, J. Lee, J. Lee, et al., "DNPU: An 8.1 TOPS/W reconfigurable CNN-RNN processor for general purpose deep neural networks," in *Proc. IEEE Int. Solid-State Circuits Conf.*, 2017, pp. 240–241.
50. K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," arXiv: 1409.1556, 2014.
51. T. Karnik, D. Kurian, P. Aseron, et al., "A cm-scale self-powered intelligent and secure IoT edge mote featuring an ultra-low-power SoC in 14nm tri-gate CMOS," in *Proc. IEEE Int. Solid-State Circuits Conf.*, 2018, pp. 46–48.
52. Y. LeCun, "Deep learning hardware: Past, present, and future," in *Proc. IEEE Int. Solid-State Circuits Conf.*, 2019, pp. 12–18.
53. K. He, X. Zhang, S. Ren, et al., "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.
54. C. Szegedy, W. Liu, Y. Jia, et al., "Going deeper with convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2015, pp. 1–9.
55. B. Wu, A. Wan, X. Yue, et al., "Shift: A zero FLOP, zero parameter alternative to spatial convolutions," in *Proc. 2018 IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Salt Lake City, UT, USA, 2018, pp. 9127–9135, doi: 10.1109/CVPR.2018.00951.
56. A. G. Howard, M. Zhu, B. Chen, et al., "Mobilenets: Efficient convolutional neural networks for mobile vision applications," arXiv: 1704.04861, 2017.
57. X. Zhang, X. Zhou, M. Lin, et al., "Shufflenet: An extremely efficient convolutional neural network for mobile devices," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2018, pp. 6848–6856.
58. X. Yang, M. Gao, J. Pu, et al., "DNN dataflow choice is overrated," arXiv: 1809.04070, 2018.
59. K. Kwon, A. Amid, A. Gholami, et al., "Co-design of deep neural nets and neural net accelerators for embedded vision applications," in *Proc. 55th Annu. Des. Autom. Conf.*, 2018, pp. 1–6.
60. F. Iandola and K. Keutzer, "Small neural nets are beautiful: Enabling embedded systems with small deep-neural-network architectures," in *Proc. 12th IEEE/ACM/IFIP Int. Conf. Hardware/Softw. Codesign Syst. Synthesis Companion*, 2017, p. 1.
61. D. Kim, R. Managuli, and Y. Kim, "Data cache and direct memory access in programming mediaprocessors," *IEEE Micro*, vol. 21, no. 4, pp. 33–42, Jul./Aug. 2001.

Received December 15, 2018; accepted for publication September 11, 2019

Alon Amid *University of California, Berkeley, Berkeley, CA 94720 USA* (*alonamid@berkeley.edu*). Mr. Amid received a B.Sc. degree in electrical engineering from Technion – Israel Institute of Technology, Haifa, Israel. He is currently working toward a doctoral degree in the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, Berkeley, CA, USA. His current research interests include parallel and distributed computing, energy-efficient processors and architectures, and hardware-software co-design.

Kiseok Kwon *Samsung Research, Samsung Electronics, Seoul, South Korea* (*kiseok.kwon@samsung.com*). Mr. Kwon received a B.S. degree in electrical engineering from Yonsei University, Seoul, South Korea, and an M.S. degree in electrical engineering from Korea Advanced Institute of Science and Technology, Daejeon, South Korea. He is a Senior Engineer with Samsung Research, Seoul. He has held multiple positions at Samsung relating to embedded systems and computer architecture, and has been a Visiting Scholar with the BAIR Lab, University of California, Berkeley, Berkeley, CA, USA.

Amir Gholami *University of California, Berkeley, Berkeley, CA 94720 USA* (*amirgh@berkeley.edu*). Dr. Gholami received a B.Sc.

degree from Tehran Polytechnic, Tehran, Iran, and MSE and Ph.D. degrees from UT Austin, Austin, TX, USA. He is a Postdoctoral Research Fellow in the BAIR Lab. He worked as an intern at AMD and Nvidia. His current research interests include large-scale training of neural networks, stochastic second-order methods, and robust optimization. He is a Melosh Medal finalist, and the recipient of UT Austin's best doctoral dissertation award in 2018, best student paper award in SC'17, Gold Medal in the ACM Student Research Competition, as well as best student paper finalist in SC'14.

Bichen Wu *University of California, Berkeley, Berkeley, CA 94720 USA* (*bichen@berkeley.edu*). Mr. Wu received a B.E. degree from Tsinghua University, Beijing, China, in 2013. He is currently working toward a Ph.D. degree in the Berkeley AI Research Lab, University of California, Berkeley, Berkeley, CA, USA. His research focuses on efficient deep learning, AutoML, neural architecture search, and autonomous driving. He was the recipient of the best paper award at the 13th Embedded Vision Workshop at CVPR2017, and he serves as a reviewer for machine learning conferences including CVPR, ICCV, and ICLR.

Krste Asanović *University of California, Berkeley, Berkeley, CA 94720 USA* (*krste@berkeley.edu*). Dr. Asanović received a Ph.D. degree in computer science from University of California, Berkeley, Berkeley, CA, USA, in 1998. He joined the faculty at MIT, receiving tenure in 2005. He returned to join the faculty at Berkeley in 2007. His main research areas include computer architecture, VLSI design, parallel programming, and operating system design. He is currently the Co-Director of the Berkeley ADEPT Lab tackling the challenge of deploying custom silicon to meet new application demands. He leads the free RISC-V ISA project, is the Chairman of the RISC-V Foundation, and has recently co-founded SiFive Inc. to support commercial use of RISC-V processors. He is also an Associate Director with the Berkeley Wireless Research Center. He was the recipient of the NSF CAREER award. He is an ACM Fellow and an IEEE Fellow.

Kurt Keutzer *University of California, Berkeley, Berkeley, CA 94720 USA* (*keutzer@berkeley.edu*). Dr. Keutzer received a Ph.D. degree in computer science from Indiana University, Bloomington, IN, USA. He joined AT&T Bell Laboratories as a Member of Technical Staff. He later joined Synopsys, Inc., where he eventually rose to become CTO and SVP of Research. Following that, he became a Professor of EECS with the University of California, Berkeley, Berkeley, CA, USA. His contributions to electronic design automation were recognized at the 50th Design Automation Conference, where he was noted as a Top 10 most cited author, as an author of a Top 10 most cited paper, and as one of only three people to have won four best paper awards at that conference. He was the recipient of the Best Paper Award at the 13th IEEE Embedded Computer Vision Workshop at CVPR and at the 47th International Conference on Parallel Processing. He has authored six books and more than 200 refereed publications. He was named a Fellow of the IEEE in 1996.