Chapter 18
Machine Learning at the Edge

Marian Verhelst and Boris Murmann

18.1 The Need for Machine Learning at the Edge

Over the last decade, electronic devices have started to ubiquitously populate our environment. Billions of connected electronic devices such as drones, smart watches, wearable health patches, smart speakers, together form the Internet-of-Things (IoT) [1]. These devices are typically equipped with around a dozen of sensors, to continuously observe the environment and act accordingly. Similarly, also in smartphones the number of integrated sensors keeps rising, to feed the devices with more information about the user and the environmental context [2].

These data collection devices, often denoted as "edge devices," capture raw sensory data streams for further processing. Recent developments resulted in algorithms capable of extracting more accurate information from such sensory data than ever before, through the usage of neural networks and other machine learning models [3]. Yet, this comes at the expense of more computationally complex algorithms, requiring many billions of computations per second, with gigabytes of storage needs [4].

Increasing computational needs are, however, in strong conflict with the limited resource budgets of edge devices: As typically powered by batteries, their energy budget is highly constrained. Furthermore, size and cost constraints limit the amount of affordable memory space and compute power. As a result, until recently, the edge devices were mainly responsible for sensory data capture, with some light preprocessing for data reduction. The compressed data could subsequently be sent to a data center, where ample compute power and memory resources are available. The

M. Verhelst (✉)
KU Leuven, Leuven, Belgium
e-mail: marian.verhelst@kuleuven.be

B. Murmann
Stanford University, Stanford, USA
e-mail: murmann@stanford.edu

B. Murmann and B. Hoefflinger (eds.), *NANO-CHIPS 2030*,
The Frontiers Collection, https://doi.org/10.1007/978-3-030-18338-7_18

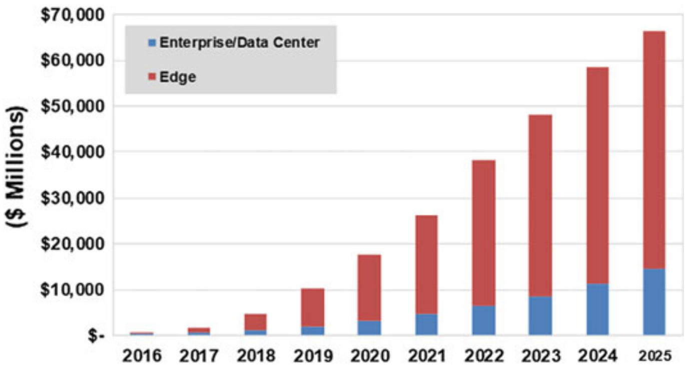


Fig. 18.1 Deep learning chip revenue for edge and data center applications. *Source* Tractica [7]

recent rise of data center activity and investments in machine learning equipment within the data centers is the consequence of this operating scheme [5, 6].

However, increasingly, users and applications shy away from such cloud-centric deployment. The desire to keep sensory data of edge devices private, as well as the energy and latency cost to send all data to the cloud, pushes for device-centric solutions, in which data is kept and processed locally as much as possible [2]. This requires edge devices to become intelligent devices that can autonomously process and interpret data in real time. This emerging operational paradigm will cause a shift in machine learning focus from the data center to the edge. As Tractica predicts (see Fig. 18.1 [7]), edge-based AI chipsets for mobile phones, smart speakers, cars, drones, AR/VR headsets, surveillance cameras and other devices will by 2025 account for more than \$50 billion in revenue, or 3.5× larger than in the data center.

To serve this emerging market, heterogeneous compute platforms are required. Special purpose processors help the traditional CPU and GPU compute platforms deployed in the edge towards resource-constrained ML processing of large volumes of sensory data. The market of such machine learning accelerators, ASIPs or ASICs, is hence expected to see the fastest growth (see Fig. 18.2 [7]), with currently already more than 70 specialty AI companies working on some sort of chip-related AI technology [8].

Up until now, this recent evolution has already resulted in a very broad landscape of customized machine learning processors, covering a wide performance space. Figure 18.3 depicts the performance of a range of state-of-the-art neural network processors [9]. State-of-the-art solutions are capable of achieving processing efficiencies of 1–100 TOPS/Watt, enabling processing at several TOPs/second within the edge devices' power budget. Yet, it is important to note that these different state-of-the-art solutions rely on very different algorithmic, architectural and technology assumptions, and cannot be fairly compared purely at the hardware level without considering other system aspects.

In this chapter, we argue and demonstrate the importance of considering the whole stack in a machine learning edge solution (see Fig. 18.4): From algorithm

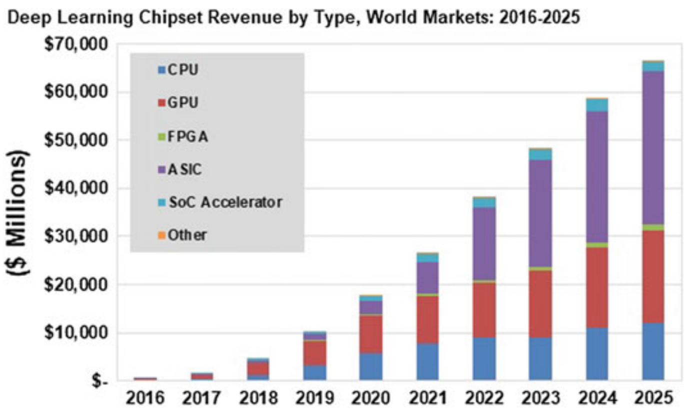


Fig. 18.2 Deep learning chip revenue by type. *Source* Tractica [7]

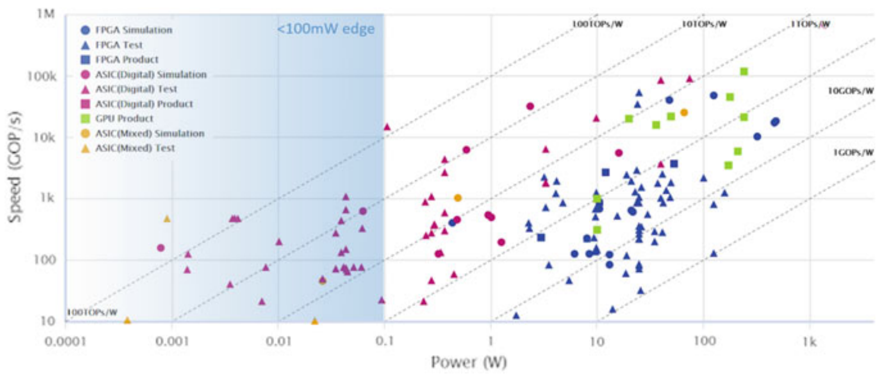


Fig. 18.3 Neural network processor comparison, highlighting the power region <100 mW, interesting for edge devices. Adapted from [9]

and dataflows (Sect. 18.2), over architectures (Sect. 18.3), to circuits and technology options (Sect. 18.4). Only such a vertically integrated approach allows to fairly benchmark different solutions relative to each other (Sect. 18.5) and perform true system optimizations towards efficient deployment of edge intelligence (Sect. 18.6). Throughout these sections, we will mostly focus on neural networks as the main machine learning model. We conclude the chapter with an outlook towards recently emerging trends, such as training at the edge, and newly emerging machine learning models (Sect. 18.7).

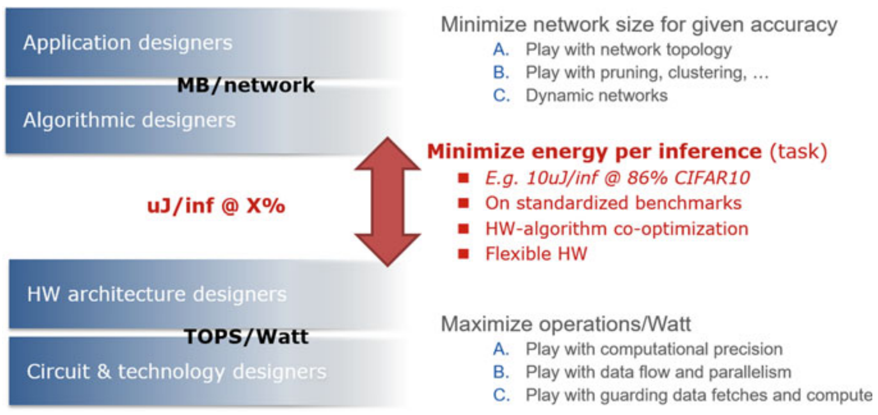
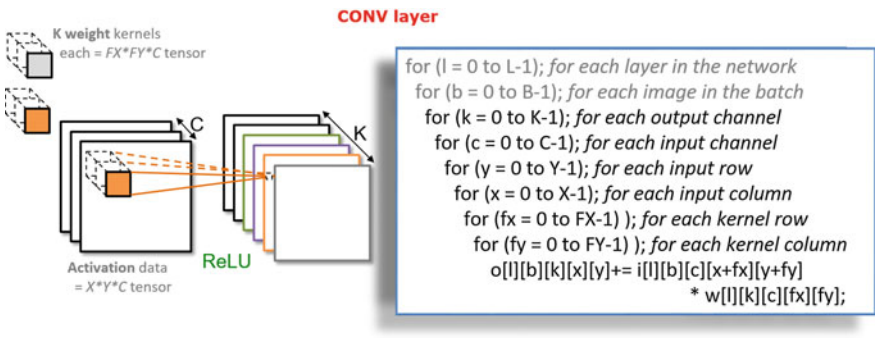

Application designers
MB/network
Algorithmic designers
uJ/inf @ X%
HW architecture designers
TOPS/Watt
Circuit & technology designers
Minimize network size for given accuracy
A. Play with network topology
B. Play with pruning, clustering, …
C. Dynamic networks
Minimize energy per inference (task)
E.g. 10uJ/inf @ 86% CIFAR10
On standardized benchmarks
HW-algorithm co-optimization
Flexible HW
Maximize operations/Watt
A. Play with computational precision
B. Play with data flow and parallelism
C. Play with guarding data fetches and compute

Fig. 18.4 Efficient edge solutions should not be optimized from a sole algorithmic perspective (minimal MB/network), nor from a sole hardware perspective (maximal TOPS/Watt), yet should jointly consider the complete design stack to come to efficient system level solutions

18.2 The Rich Algorithmic Landscape of ML at the Edge

Machine learning models are currently in high flux. In the field of deep neural networks, a wide range of model topologies is currently under exploration. Each model is carefully built out of a sequence of neural network layers. The most generic neural network topology element is a convolutional layer. Such a layer takes in a three-dimensional data tensor and produces a three-dimensional output data tensor through convolving the input tensor with a series of 3D weight kernels [10]. This is illustrated with the relevant data dimensions highlighted in Fig. 18.5. The convolutional operation of one neural network layer can be captured in eight nested for-loops, with a multiply-accumulate operation at the core (see Fig. 18.5). Since in edge devices real time operation requires every input data item to be processed as soon as it comes



```
for (l = 0 to L-1); for each layer in the network
 for (b = 0 to B-1); for each image in the batch
  for (k = 0 to K-1); for each output channel
   for (c = 0 to C-1); for each input channel
    for (y = 0 to Y-1); for each input row
     for (x = 0 to X-1); for each input column
      for (fx = 0 to FX-1) ); for each kernel row
       for (fy = 0 to FY-1) ); for each kernel column
        o[l][b][k][x][y]+= i[l][b][c][x+fx][y+fy]
                              * w[l][k][c][fx][fy];
```

Fig. 18.5 For each item in a batch, each convolutional layer represents six nested for-loops per inference

in, batching is not tolerated and a batch size of one is typically used, making B = 1. As processing efficiency is of such crucial importance in edge devices, research here is focused on algorithmic transformations that impact model size and execution cost without affecting model accuracy. We briefly survey model compaction, model quantization and model pruning techniques, and give an outlook to the future in this area.

Model topology and model compaction: The index ranges of the aforementioned for-loops are determined by the layer and network topology and (as we show later) strongly influence the network's execution efficiency in hardware. Network designers hence use these dimensions in a quest to construct the most compact or efficient models which can fit in small sized embedded memories. Such model compaction research led for instance to the introduction of bottleneck layers [11]. Here, a three-dimensional convolutional layer is replaced by a stack of three layers in which the first and last layer only perform a one-dimensional convolution (FX = FY = 1) to reduce the number of channels (see Fig. 18.6a). Experiments have proven that such structures maintain good modeling capabilities while drastically reducing model computations and coefficients in the three-dimensional convolution (middle layer), thus lowering compute and memory needs. This technique is often combined with the usage of parallel network layers, which are concatenated further in the network, as in the Inception module in Fig. 18.6c [11].

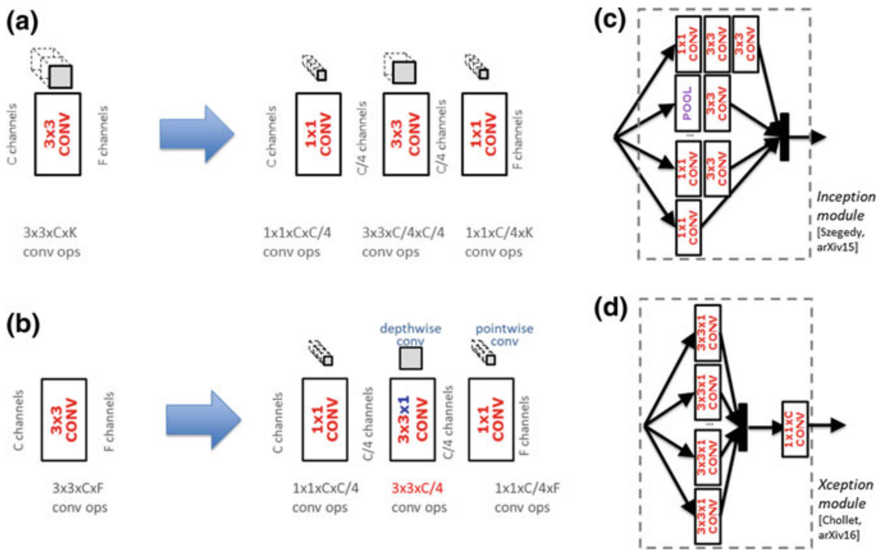


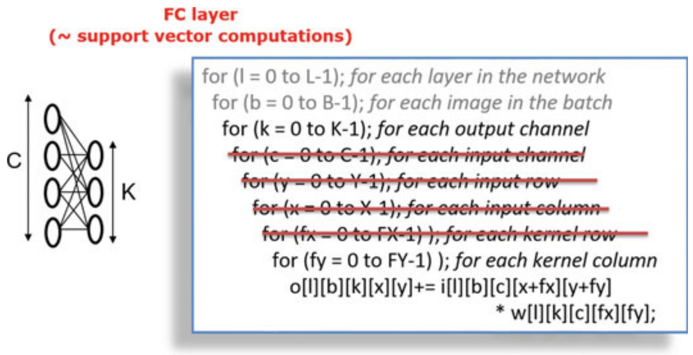
Fig. 18.6 Evolution toward reduced-dimension neural network layers: (**a**) bottleneck layers and (**b**) depth-wise/point-wise layers. These techniques are combined with parallel layers that are subsequently fused, such as in (**c**) the inception and (**d**) Xception modules

To further reduce the computational load of the remaining three-dimensional convolution, the middle layer of this stack was subsequently replaced by a two-dimensional convolution, which only convolves within one channel (removing the for-loop across C, Fig. 18.6b). The resulting "depthwise-pointwise" technique [12] was successfully used in the creation of MobileNet [13], a lightweight network to perform object recognition on mobile phones. In a next generation, MobileNetV2, this technique was further combined with feedforward connections across network layers [14].

Recently, a new paradigm shift emerges: Network topologies are no longer optimized by hand, but are the result of automated neural network search, also denoted by AutoML. Here, reinforcement learning, evolutionary algorithms and/or random sampling strategies are used to find more compact and better performing networks [15–21]. The focus in this field of research is on finding the best performing networks from an accuracy point of view, while minimizing the amount of GPU compute time required for the network search. Very few works [22–24], however, take the neural networks execution efficiency on edge devices into account in the cost function when searching for the most optimal networks. This is discussed further in Sect. 18.6.

From previous discussion, it should be clear that a wide variety of convolutional topologies exists for neural network layers, which are often combined, concatenated and interconnected in many different and irregular ways. When developing hardware architectures, we hence must ensure sufficient flexibility to support the mapping of all these different topologies and dataflows (see also Sect. 18.6).

Beyond convolutional layers, other types of neural network and non-neural network models must also be supported. Yet, interestingly enough, they can often be rewritten in the form of the generic convolutional model in Fig. 18.5. For example, the fully connected neural network layer [10], often found at the end of classification networks, can be rewritten in the same form of the convolutional layer with X = Y = FX = FY = 1, as indicated in Fig. 18.7. Likewise, other machine learning kernels, such as the support vector machine (SVM) [25] and one-class SVMs (often used in



```
for (l = 0 to L-1); for each layer in the network
 for (b = 0 to B-1); for each image in the batch
  for (k = 0 to K-1); for each output channel
   ~~for (c = 0 to C-1); for each input channel~~
    ~~for (y = 0 to Y-1); for each input row~~
     ~~for (x = 0 to X-1); for each input column~~
      ~~for (fx = 0 to FX-1) ); for each kernel row~~
       for (fy = 0 to FY-1) ); for each kernel column
         o[l][b][k][x][y]+= i[l][b][c][x+fx][y+fy]
                         * w[l][k][c][fx][fy];
```

Fig. 18.7 For each item in a batch, each fully connected neural network layer represents two nested loops. Similarly, each support vector machine evaluation represents two nested loops per inference

anomaly detection), demonstrate similar matrix-vector multiplication kernels, fitting the same framework. This is good news, as it simplifies the development of a generic hardware platform for such machine learning workloads (see Sect. 18.3). Yet, these computational layers have fewer effective nested for-loops, which results in fewer opportunities for efficient hardware mapping, as seen later in this chapter.

Model quantization: Researchers have found that neural network models carry some redundancy, making them to a certain extent robust to perturbations. This enables further model efficiency and storage reduction techniques that exploit sparsity and reduced precision operation. Regarding computational precision, limited-precision fixed point data representations for both weights and activations were shown to be sufficient for nearly all inference tasks, drastically cutting model memory weight storage and MAC complexity [26–28]. Operation down to 8, 4 or even fewer bits has been demonstrated for many machine learning benchmarks, with ternary (-1, 0, 1) and binary (-1, 1) neural networks as the extremes [29]. The best results are achieved when using the dynamic fixed point format [30], and quantizing the network during the training process [31, 32], instead of first training a floating point network and quantizing it afterwards, or smartly unifying the dynamic range of all weights during training [33, 34]. Active research tries to find efficient ways to determine the minimum bit width representation necessary to achieve a target accuracy level for a given task, which at the moment is still relying largely on inefficient exhaustive searches. It is important to realize that this optimum is heavily interwoven with the selected network topology and cannot be looked at in isolation [35].

Model pruning: Instead of just quantizing the weights of a network, one can also remove some weights completely, which is called "pruning the network." Many pruning techniques exist, ranging from after-training techniques that just remove smallest weights of a network [36–38], to during-training regularization techniques that try to force as many coefficients as possible to become approximately zero [39]. This results in sparse neural network models, whose zero values can be exploited to further reduce the model's storage and computational footprint. Several model compression formats have been proposed, such as the Compressed Sparse Column (CSC) format, which encodes the sparse matrices and vectors into fewer words by skipping zero-valued data [36]. The processor must of course be equipped with the corresponding decoding logic to be able to interpret this data [40].

Interdependencies: It is important to realize that all aforementioned optimizations, such as model compaction, model quantization and model pruning are strongly interwoven. It is observed that compact models tend to be less sparse, and less tolerant to quantization [35]. Finding the most efficient model hence requires balancing all three techniques. As this results in an enormous algorithmic search space, current research is strongly invested in exploring this space as efficiently as possible. Breakthroughs have been achieved using automated machine learning (AutoML) techniques exploiting Bayesian optimization, evolutionary algorithms and reinforcement learning [15–21]. Yet, quantization and hardware inference cost has received limited attention in this field.

Processor consequences and outlook: The optimization techniques adopted for networks strongly influence the execution efficiency on the processor hardware. As an example, model compaction techniques typically result in models with smaller (FX, FY) filter kernels or (X, Y) activation sizes, causing a drop in data reuse opportunity [41]. Similarly, pruning breaks the processing regularity that made traditional deep learning processors so efficient. As a result, the smallest model is not necessarily the most efficient one for execution at the edge [35]. This gives rise to new, more hardware-aware algorithmic techniques, such as structured sparsity or dynamic neural networks. To understand this better, let's take a closer look at edge processing architectures.

18.3 The Rich Architectural Landscape of ML at the Edge

From CPU to GPU to NPU: As neural networks are characterized by massively parallel MAC operations, their processing requires widely parallel execution. On traditional Von Neumann CPUs this is achieved by exploiting vector processing instructions for parallel MAC execution [42] (Fig. 18.8). Recently, CPUs have been equipped with additional fused (integer) multiply add (FMA) instructions, which allow to also efficiently accumulate multiplication results. Yet, these processors lack sufficient computational resources to achieve more than 100× parallelization factors, limiting performance to a few hundred GOPs per processing core.

For this reason, GPUs have been extensively used as the main neural network inference platform. They are equipped with many parallel execution units and can achieve 1000× or more parallel MAC operations. Moreover, over the last few years, GPUs have moreover a rapid evolution to serve neural network inference workloads even better. First of all, recent implementations support small word length fixed-point data types instead of only supporting floating point operations. Secondly, traditional GPU architectures did not support efficient spatial and temporal reuse of data across processing elements (see also below). The recent inclusion of tensor cores, which spatially unroll the multiplication of two 4 × 4 matrices in one timestep, alleviates this issue for certain layer topologies [43]. Still, flexibility and efficiency across kernel sizes and models remains an issue, and (embedded) GPU power consumption exceeds the power budget of many edge solutions.

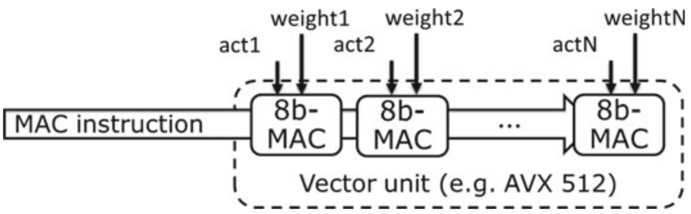


Fig. 18.8 Traditional vector processing units can be used to achieve parallelization for neural network processing. Yet, they are not fully exploiting the neural network data flow properties

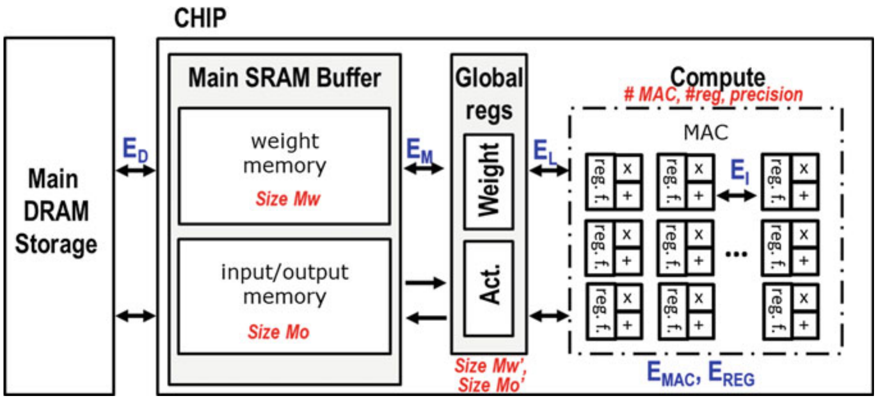


Fig. 18.9 NPU architectural template, which is parametrized across many design dimensions, ranging from the number of parallel MACs and their interconnectivity, to the levels of memory and their sizes and interconnectivity

For this reason, more and more specialized, custom processor cores are appearing, optimized towards neural network inference in resource constrained devices [44]. These class of processors is often denoted as "NPU," or Neural Processing Unit. NPUs consist of a widely parallel datapath equipped with MACs with or without local storage, together with a hierarchy of several optimized memory layers, as shown in Fig. 18.9. Several efficiency techniques are exploited across NPU designs, which we shall discuss in more detail: (1) Spatial and temporal data reuse; (2) hierarchical memories and local storage; (3) sparse or dense processing; (4) reduced precision processing.

Spatial and temporal data reuse: A large fraction of NPU power consumption is spent on data fetches. Good NPU designs therefore try to maximize not only the number of parallel MACs that can be executed in every single clock cycle, but also minimize the average number of data fetches per usefully executed MAC. This can be achieved through spatial or temporal data reuse across different layers of granularity (see Table 18.1). Spatial data reuse exploits the use of multi-dimensional data paths to reuse fetched weights and/or activations across many parallel MAC operations within a processing element (PE) array. Figure 18.10 shows the architecture of the Envision processor [45], in which every weight is multiplied with 16 input activations in parallel, while every input activation is multiplied with 16 weights (of different output channels) in parallel. Also, the number of data stores can be reduced spatially,

Table 18.1 Data reuse opportunities classified across granularity and their spatial/temporal nature

	Intra-PE	Inter-PE (Intra-PE array)	Inter-PE array
Spatial reuse	–	Multi-dimensional datapaths Accumulation trees	Broad-/multi-casting networks
Temporal reuse	Stationarity	Systolic arrays	Systolic/streaming processors

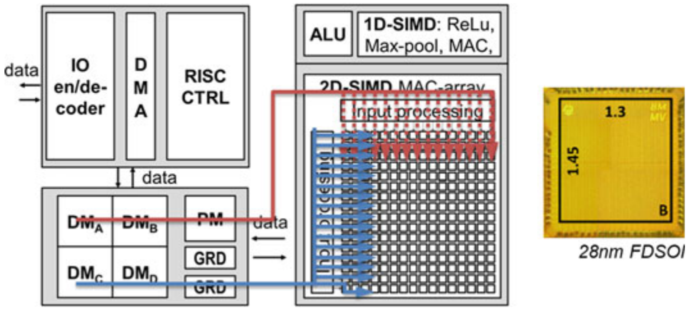


Fig. 18.10 Envision processing architecture, exploiting spatial reuse of input activation data (red) and weight data (blue). Chip photo on the right

by introducing summation trees that accumulate results across PEs before sending them back to memory.

Besides purely reusing data spatially within a single clock cycle, data can also be reused temporally, across clock cycles. Here, a distinction can be made between processing architectures that reuse data across subsequent clock cycles within the same processing element (stationary techniques), and architectures that reuse data across subsequent clock cycles within neighboring processing elements or datapaths (systolic processing architectures). A very common stationarity approach is to keep the MAC output locally and accumulate within a PE in subsequent clock cycles. Envision [45] is an example of such an "output-stationary" approach. Other implementations keep the weights local within a PE across cycles ("weight stationary"), such as the weight stationary TPU processor of Google [46], or the BinarEye processor [47].

Systolic architectures, on the other hand, exploit the fact that it is cheaper to exchange data between neighboring processing elements, instead of sending them to a larger remote memory. Also, systolic principles can be applied at different levels of granularity: At the lowest level, neighboring PEs can pass partial accumulation results and/or weights to each other, as for example done in the aforementioned TPU processor [46] and the Eyeriss chip [48]. But, also across larger clusters, data can be forwarded from processor to processor, with only small streaming buffers in between, avoiding data transfers in and out of large memories. This is done e.g. in [49]. These processor architectures break with the traditional Von Neumann architecture and tightly intertwine processing elements and memory blocks.

Hierarchical memories: To further reduce energy spent on memory fetches and stores, the memory hierarchy is further optimized. Instead of using one large central memory, data is stored as close as possible to the place where it is generated and consumed, while using a memory block that is as small as possible. This results in hierarchical memory structures, while small local memories, complemented with several layers of larger shared memories further up in the hierarchy, as shown in Fig. 18.11. The challenge here is to determine the optimal memory sizes at each level in the hierarchy, not for a single network topology, but across many network topologies (see further discussion below).

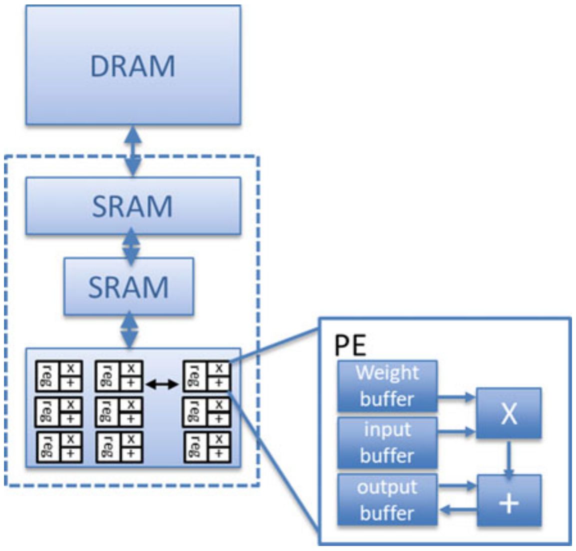


Fig. 18.11 Hierarchical processor memory hierarchy

Sparse or dense: As discussed in Sect. 18.2, neural network models typically exhibit a certain degree of sparsity, which can be exploited in the processing hardware. Indeed, when doing a multiply accumulate operation with one of the multiplication inputs being zero, the accumulation result remains unchanged. The most straightforward way to exploit such sparsity, is to maintain the regular dense processing grid, yet simply clock and data-gate all units that encounter a zero-valued input. Processors such as Envision [45], or Eyeriss [48] support this approach. The operating scheme allows saving power when executing sparse networks, and only comes with very little overhead logic to support the clock and data gating. Yet, the approach only brings (modest) power savings, and does not lead to increased throughput for sparse workloads. Indeed, all idle MAC units are wasting useful processing resources.

This is overcome in sparse NPU processors, which target skipping all zero-valued operations and assign their computational resources only to useful computations. Such an approach allows to automatically speed up processing when the networks are very sparse. Yet, the approach is penalized by large architectural overhead for data decoders, scheduling logic, and irregular data routing. Moreover, data reuse opportunities drop drastically in such processors, often limiting the amount of effective parallel operations that can take place. As a result, such processors prove to be beneficial only when the sparsity is large enough. Parashar et al. [50], have shown this break-even point to lie around 40% sparsity (60% density) for both weights and activations (see Fig. 18.12). While older networks had very high sparsity (e.g., 80% or more for AlexNet), newer networks exhibit different characteristics. The recent network compaction techniques result lower sparsity, ranging between 10 and 70% for networks like GoogleNet, 10–50% for MobileNet and only 10% for MobileNet [51].

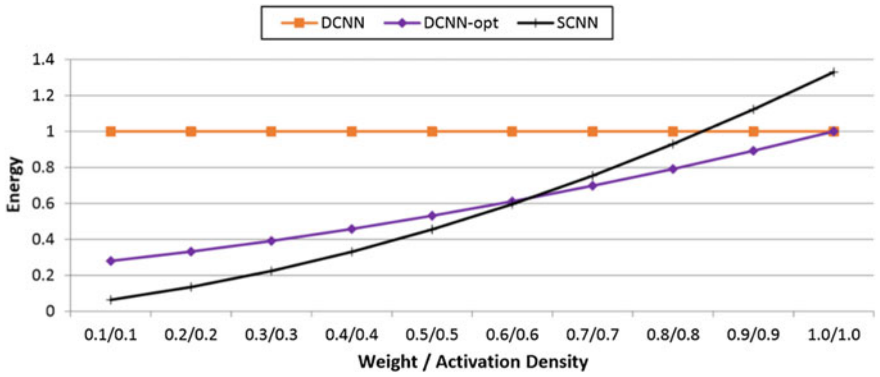


Fig. 18.12 GoogLeNet performance and energy as a function of density for a non-sparsity-aware processor (DCNN), a sparsity-aware processor with datapath gating (DCNN-opt) and a sparse execution processor (SCNN) (from [50], ©IEEE 2017)

Reduced precision: Another algorithmic property that can be exploited at the hardware level is the robustness to reduced computational precision. As discussed in Sect. 18.2, neural networks can be trained to operate with low-resolution fixed-point number representations. Figure 18.13 illustrates this, assuming m-bit integer activation values and n-bit integer weights, drastically reducing the multiplier complexity, area and power consumption. Precision scaling can be done symmetrically ($m = n$) or asymmetrically ($m \neq n$) [52]. Data types used in inference accelerators are often INT8, and more and more frequently also INT4, or even ternary or binary (INT1) values. As can be seen from Fig. 18.14, reduced precision processing typically results in both performance and efficiency boosts.

It is important to realize that different neural networks have different optimal fixed-point word lengths [51]. Even between layers of the same network, optimal quantization values might differ, typically requiring more bits for full resolution input layers for image processing. As a result, a widely deployable NPU processor needs internal MAC units that can operate at different precision settings. The settings should be easily configured, e.g. through a simple processor instruction. Moreover, the overhead of this configurability at the MAC level should be limited to maintain

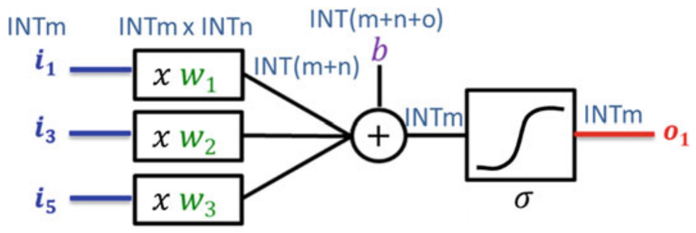


Fig. 18.13 Operating with low-resolution fixed-point number representations

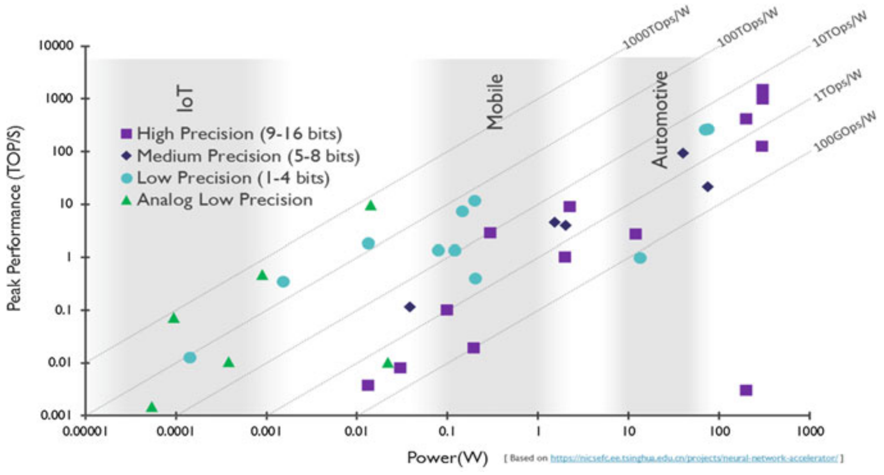


Fig. 18.14 Sample of recent NPU implementations, indicating the precision of the internal MAC units. *Source* [9, 53]

good efficiency across all precision levels. Many precision-scalable MAC designs have been proposed in the literature, each of which coming with their own merits and downsides [52, 54, 55]. Table 18.2 summarizes the main precision scalability architectures in a taxonomy introduced in [55]. 1D scalable designs demonstrate good scalability at weight-only asymmetric scaling, while 2D scalable designs perform well when one wants to scale both activations and weights. 2D scaling can be performed symmetrically across weight and activations, or asymmetrically. This scaling, however, always comes at the expense of increased memory bandwidth in low precision modes, with increased bandwidth pressure on the memory stores when using sum apart techniques, and pressure on the memory loads for the sum together techniques. Across all operating modes, bit serial techniques do not seem to pay off, based on this comparative study. A more elaborate survey can be found in [55].

Table 18.2 Variable precision MAC taxonomy (from [55]) and reported implementations exploiting the various techniques

Architecture types		1D scalable (weight only)	2D asymmetric scalable	2D symmetric scalable
Spatial	Sum apart	[56] (DNPU)		[57] (DVAFS)
	Sum together		[58] (BitFusion)	[54]
Temporal	Serial	[59] (UNPU)	[60] (LOOM)	
	Multi-bit serial	[52]	[60] (LOOM)	

Challenges and outlook: The break from traditional Von Neumann processing architectures, and inclusion of support for multi-dimensional data reuse, sparse processing and reduced precision operation, have pushed the efficiency of NPU processing to 1–2 orders of magnitude beyond CPU and GPU solutions (see Figs. 18.3 and 18.14). Going forward, the challenge is to ensure support for a wide range of new and upcoming neural network paradigms, such as dynamic networks [61], dilated networks [62], shiftnets [63], wavenets [64], etc. These networks are characterized by (sometimes even dynamically) varying kernel sizes, low data reuse factors, and complex layer interconnectivities. Making processors that are flexible enough to maintain good execution efficiency across the complete set of workloads, while keeping configuration overhead low, is the main research challenge at the moment. To achieve these properties, the importance of early processor modeling, and lean dataflow optimizations is rapidly rising, leading to a new class of schedulers, mappers, and compilers that are discussed in Sect. 18.6.

In the future, we'll see these architectures evolve further towards more distributed processing, with small, yet flexible buffers between precision-scalable processing elements. As the memory access remains the main bottleneck, emerging technologies that integrate the memory and computations are rapidly gaining importance and are thus discussed in the next section.

18.4 The Rich Circuit/Technology Landscape of ML at the Edge

The previous sections looked at efficient neural network computation mainly from the perspective of algorithms and architecture, corresponding to the upper three layers of Fig. 18.4. However, a wide range of options are also available at the circuit and technology level, which complicates the search for an optimal implementation even further. In this section, we briefly review the most common innovation vectors.

Analog and mixed-signal computing: There is a rich history of research that promotes the purely analog implementation of neural networks and other machine learning algorithms. This path typically follows neuromorphic principles [65, 66], which build on our (very limited) understanding of the human brain and its "integrate and fire" neurons that are amenable to an analog circuit implementation. While the resulting neurons represent an intriguing and biologically plausible emulation of the units found in the human brain, the networks constructed with them tend to lack scalability. It is fundamentally difficult to array and cascade a large number of analog building blocks and deal with the accumulation of noise and component mismatch. Additionally, and perhaps more significantly, it is challenging to build the required analog memory cells [67]. For this reason, present explorations in neuromorphic design are dominated by digital emulations, such as IBM's TrueNorth processor [68]. A more detailed discussion of such efforts is found in Chap. 22 of this book.

Since purely analog implementations are difficult to scale, could one instead assemble a processor that uses purely digital storage and adds in analog/mixed-signal compute for potential efficiency gains? As shown in [69], mixed-signal computing can indeed be lower energy than digital for low resolutions, typically below 8 bits. The most straightforward way to exploit this would be to embed mixed-signal compute macros into the PE blocks of a mainly digital processor. This was considered in [70, 71], which point to the conclusion that the idea will in practice lead to diminishing returns. In an optimized digital design that conforms to the template of Fig. 18.11, most of the energy is spent on memory access and data movement [72] making even large improvements in the arithmetic units nearly irrelevant. To fully harvest the benefits of mixed-signal processing, one must consider customized architectures. One possible direction is to employ analog and mixed-signal circuits as feature extractors that are placed in front of a digital neural network. This approach is discussed further in Chap. 17 of this book. Another opportunity is to exploit mixed-signal circuits through memory-like processing elements and in-memory computing, which we discuss next.

Memory-like processing elements: Is it possible to re-architect a digital ML processor architecture to benefit more strongly from a mixed-signal compute fabric? This question was the baseline for the research described in [73], which exercises two of the re-use principles stated in Table 18.1 with a mixed-signal mindset: Intra-PE temporal re-use of weights (weight stationarity) and Inter-PE accumulation. The main observation here was that the latter can be done in a particularly efficient way using charge sharing on a wire, instead of a digital accumulation tree. The resulting switched-capacitor PE cell is shown in Fig. 18.15. The overall network that was designed to use this PE is based on the BinaryNet topology from [29], which makes multiplication trivial (XNOR). This enabled a cell size that allowed the on-chip integration of a 64 $\times$ 1024 PE array that computes 64 output activations in one shot. We term this approach "memory-like" since the PE locally stores one bit and otherwise contains only simple add-on-circuits.

Figure 18.16 compares the total neuron energy of a custom digital design with the described mixed-signal approach. The latter shows an improvement of about 4.2$\times$. However, when accounting for other energy consumers (including weight

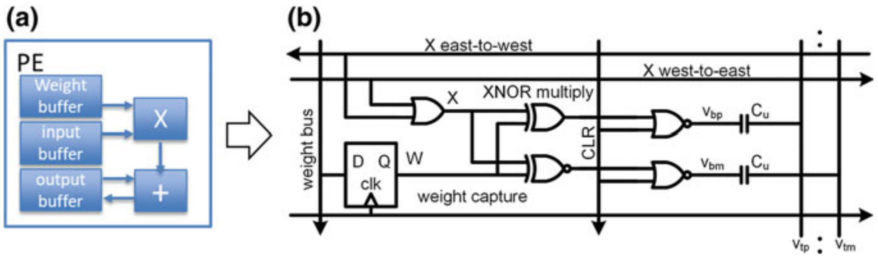


Fig. 18.15 **a** Conventional processing element (PE) versus **b** memory-like mixed-signal PE (single-bit implementation, from [73] ©IEEE 2019)

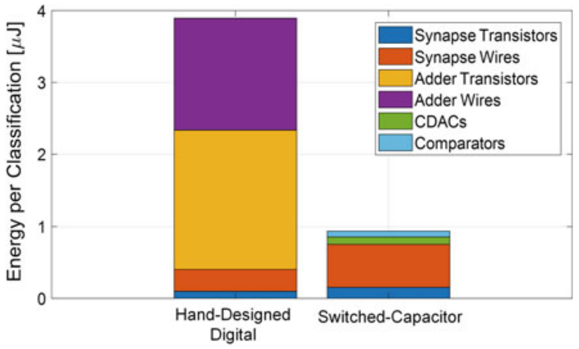


Fig. 18.16 Comparison of total neuron energy (digital versus mixed signal) (from [73] ©IEEE 2019)

and activation memory access), the system-level savings reduce to approximately 1.8×. While this benefit is still significant, this exercise makes it clear that order-of magnitude improvements are hard to come by, unless an even more radical approach is pursued. This brings us to the topic of in-memory computing, which aims to minimize the overhead that diminished the returns from the mixed-signal compute fabric in the example above.

In-Memory Computing: In-memory computing is a relatively old idea [74] that aims to co-integrate memory and compute into a single dense fabric. Conceptually, one could view the memory-like PE in Fig. 18.15 as a compute-in-memory cell. However, its size is relatively large, so that a denser piece of memory is required in its periphery to store the weights and activations of a modern neural network. To overcome this issue, denser cells can be designed as illustrated in Fig. 18.17. The most obvious way to increase density is to handle the memory bit with a standard 6-T SRAM cell (see Fig. 18.17b) as done in [75]. In addition, the logic can be simplified and single-ended signaling can be explored to further reduce the area. While the differential memory-like cell measures 24,000 F^2 (where F is the half pitch the process technology), the SRAM-based cell has an area of only 290 F^2. Further cell size reductions are possible by migrating to emerging memory technologies (see Fig. 18.17c), as discussed in the next sub-section.

At present, SRAM-based in-memory computing is receiving significant attention in the research community [76] and many circuit and network architecture options are being explored. In [77], a complete processor with in-memory compute acceleration

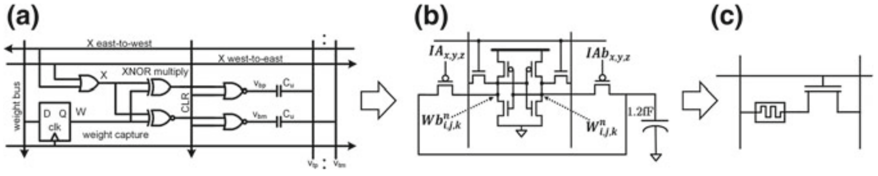


Fig. 18.17 **a** Memory-like PE (from [73] ©IEEE 2019), **b** in-memory computing cell based on SRAM (from [75] ©IEEE 2019), **c** in-memory computing cell based on resistive memory technology

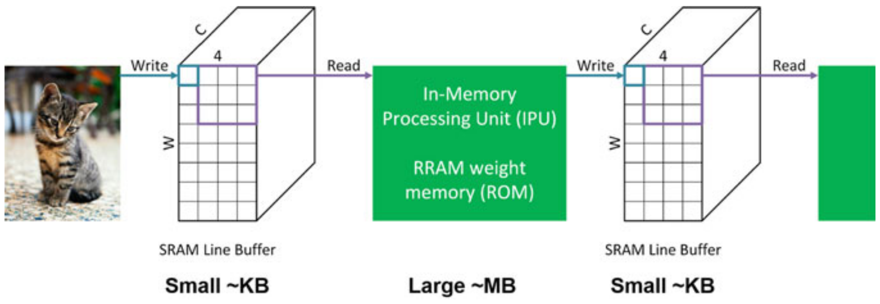


Fig. 18.18 Streaming architecture for neural network processing with emerging memory

is presented. While this design achieves high efficiency within its compute tiles, the overall system efficiency is held back by memory reads from external DRAM, which is typically required for models that exceed several megabytes in size. A promising remedy for this issue lies in embracing emerging memory technologies for in-memory compute.

Emerging Memory: A wide variety of emerging memory technologies are currently under investigation (see Chap. 19 of this book). For instance, Resistive Random Access Memory (RRAM) technology promises to deliver densities that are comparable to DRAM, while being non-volatile and potentially offer multi-level storage. This could open up a future where relatively large machine learning models (>10 MB) can be stored on a single chip to eliminate costly DRAM access. In addition, these memory types are compatible with in-memory-computing by exploiting current summation on the bitlines [78]. While there are many possible ways to incorporate emerging nonvolatile memory into a machine learning processor [79], one attractive option is a streaming topology as shown in Fig. 18.18. Here, large in-memory compute tiles are pipelined between small SRAM line buffers that hold only the current input working set [80]. This scheme can thereby avoid the energy penalty of reading from large SRAMs, which represents a significant energy overhead in the above-discussed processor with memory-like PEs.

Presently, the art of designing of machine learning processors using emerging memory is still in its infancy. Key issues include access to process technology as well as challenges with the relatively poor retention and endurance of emerging memory technologies (see e.g., [81]). Consequently, most existing demonstrators are only sub-systems and use relatively small arrays (see e.g., [82]). However, one important aspect that has already become clear from these investigations is that the D/A and A/D interfaces required at the array boundaries can be a significant showstopper. For example, a state-of-the-art ADC consumes about 1 pJ per conversion at approximately 4–8 bits of resolution [83]. If amortized across 100 memory rows, the energy overhead is 10 fJ per MAC operation, a number that is close to a relatively straightforward digital MAC implementation in 16 nm CMOS [84]. The solution is to work with taller arrays and to push for innovations in the interface and array circuit design (see e.g. [85]), which can lead us to single-digit fJ per MAC.

3D Integration: Given the above-discussed problems of data movement and memory access in large neural networks, it is clear that 3D integration has the potential to play a major role in making NPUs significantly more efficient. The reader is referred to an in-depth discussion of this subject in Chaps. 9 and 10.

Challenges and outlook: While using analog and mixed-signal computing in neural networks is attractive in principle, it is not straightforward to realize large performance gains (e.g., order of magnitude) at the system level. This is simply because a complete NPU has many components and improving only a subset leads to diminishing returns. At present, the most promising option is to pursue mixed-signal processing within in-memory compute tiles and to rely on standard digital processing on the outside. Future work must assess how flexible and programmable such a processor can be, and how much efficiency it may lose due to data sparsity, which can presumably be better managed with a fully digital fabric. Just as with fully digital NPUs, the research on alternative architectures must be guided by a solid system-level benchmarking strategy that will systematically uncover such efficiency losses during the conception of the architecture. The next section therefore looks at this particular aspect.

18.5 Evaluating ML Processors

As discussed in the previous sections, over the past decades, hundreds of custom NPU processing schemes, architectures and technological enhancements have been proposed. It is good practice to benchmark the different solutions relative to one another, and identify which innovations bring actual value. Yet, the main challenge is to determine the right benchmarking metrics.

System-level benchmarks: The only metrics that really matter to an edge device user are: (1) the energy per inference; (2) the latency or throughput per inference; and (3) the cost per inference (determined by chip area and external memory size). To be able to compare different systems, these must be compared on a known standardized benchmarking task, achieving a given target accuracy. Recently, there has been a lot of effort from the MLperf community [86] to pull off such benchmarking. While the current focus is mostly on training tasks in the cloud, it is expanding towards inference benchmarks, also for the edge.

These system-level benchmarks can be improved through different algorithmic, architecture and circuit level techniques. Designers working at these levels tend to use benchmarking metrics focusing at lower level aspects, for instance:

- The number of MACs/inference or model coefficients at *algorithmic* level
- The number of MACs/second or the number of MACs/Watt at *architectural* level
- The number picojoules per memory fetch of per MAC at the *circuit/technology* level.

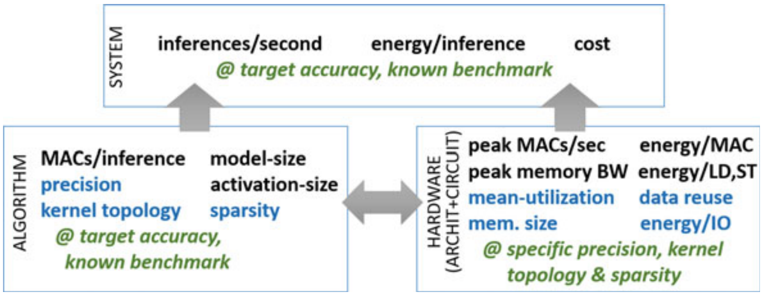


Fig. 18.19 Typical benchmarking metrics at system level, algorithmic level and hardware level (in black), complemented with performance-influencing metrics (blue) and constraints (green) that are often forgotten

Figure 18.19 summarizes some frequently used benchmarks (in black) at these different levels. The figure also highlights important parameters (in blue), and constraints (in green) that are often forgotten at these different levels. It is of crucial importance to see that benchmarks at different levels strongly depend on each other and are often conflicting. For example, one can achieve a very low number of model weights by going to high precision, highly sparse model kernels. Yet, at the hardware level, this will result in very low MAC utilization and high energy per MAC. Similarly, good hardware benchmarks can be achieved by going to very low precision, and highly regular large in-memory compute arrays. Yet, this will result in models that are requiring more MACs and larger model sizes to achieve the same benchmarking accuracy [87].

CIFAR10 example of cross-layer implications: To illustrate this, we compare different solutions for the CIFAR10 benchmark. Table 18.3 shows benchmarking performance across different design levels for three different solutions:

- A high accuracy, 4-bit algorithm running on the Envision chip [45]
- A medium accuracy 4-bit model running on the Envision chip [45]
- A medium accuracy 1-bit model running on the BinarEye chip [88].

It is interesting to observe that at the hardware level, the BinarEye chip [88] seems to beat all performance metrics, showing highest peak performance, at best energy efficiency and with most embedded memory available. At algorithmic level, however, the network capable of execution on the Envision platform show to require less MACs and exhibit more sparsity. However, as their topology cannot be perfectly mapped to the flexible Envision datapath, it cannot achieve maximum utilization of the processor. The network trained for BinarEye on the other hand, was matched to the datapath to achieve 100% utilization. The result of this trade-off shows that for equal accuracy, two solutions consume roughly the same amount of energy to run one CIFAR10 inference. At the system level, also taking external memory accesses into account, BinarEye wins due to the larger embedded memory of BinarEye and the smaller model size of the mapped CIFAR10 model. The table also clearly shows that large energy savings can be achieved if one wants to give in a bit of task accuracy, e.g.

Table 18.3 Benchmarking performance of CIFAR10 task across platforms. Numbers extrapolated from measurements

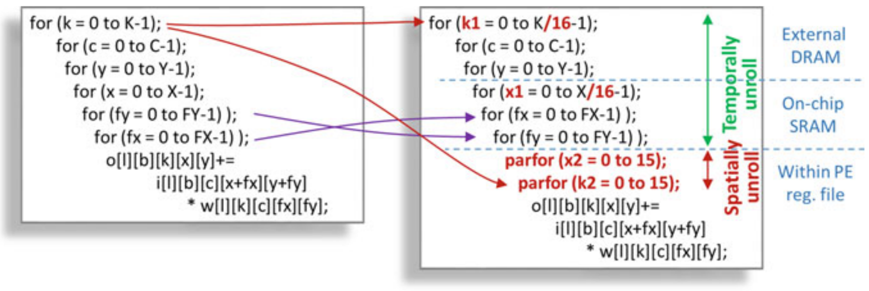
Platform and task	System level			Algorithm level					Hardware level			
	Inference/s	System energy/inf (uJ)	Chip energy/inf (uJ)	Kernel topology	Precision (bit)	MAC/inference	Sparsity (%)	Model size (MB)	TMAC/Watt	Peak GMAC/s	Mean utilization (%)	Mem size (kB)
Envision, CIFAR10, 90%	90	400	230	3 × 3 × C	4	6.00E + 08	30–60	12	2.6	102	53	144
Envision, CIFAR10, 86%	1350	25	15	3 × 3 × C	4	4.00E + 07	30–60	1.2	2.6	102	53	144
BinarEye, CIFAR10, 86%	120	14	13	2 × 2 × C	1	2.00E + 09	0	0.259	115	2800	100	328

comparing the system level benchmarks for CIFAR10 90% and 86% in Table 18.3. It is hence of crucial importance to always compare data points achieving similar accuracies on known benchmarks to be able to make a fair comparison.

Challenges and outlook: The previous example should make it clear that it is impossible to judge hardware platforms, resp. algorithmic innovations based on hardware-centric, resp. algorithm-centric performance metrics. There is a very strong influence between design decision across different layers. The challenge is hence on being able to report system-level benchmarking improvements for newly proposed algorithmic or hardware innovations, without having to go through the complete optimization across all layers every time. This requires a new set of cross-layer tools and frameworks, as discussed in Sect. 18.6.

18.6 Cross Domain Optimizations, Mapping and Deployment Frameworks

Neural network mapping: When one wants to assess the performance of a specific neural network model on a specific hardware topology, it is necessary to schedule the model's execution on consecutive processing cycles using a mapping supported by the platform. Only the detailed schedule reveals how many data transfers are needed to execute the algorithm, and which layers of the memory hierarchy are involved. For a specific neural network layer, such scheduling starts from the layer's six nested loops, shown in Fig. 18.5. These nested for-loops can be manipulated using loop splitting and loop reordering, denoted as dataflow transformations [89, 90]. Finally, each resulting for-loop should be characterized as a spatial or temporal enrolled loop (in line with what the hardware supported), and its internal data variables should be allocated to a specific level in the hardware's memory hierarchy. Figure 18.20 illustrates this operation for an algorithm mapped on the Envision processor, which supports two-dimensional parallelism along the X and K dimension.



```
for (k = 0 to K-1);
  for (c = 0 to C-1);
    for (y = 0 to Y-1);
      for (x = 0 to X-1);
        for (fy = 0 to FY-1) );
          for (fx = 0 to FX-1) );
            o[l][b][k][x][y]+=
                  i[l][b][c][x+fx][y+fy]
                        * w[l][k][c][fx][fy];
```

```
for (k1 = 0 to K/16-1);
  for (c = 0 to C-1);
    for (y = 0 to Y-1);
      for (x1 = 0 to X/16-1);
        for (fx = 0 to FX-1) );
          for (fy = 0 to FY-1) );
            parfor (x2 = 0 to 15);
              parfor (k2 = 0 to 15);
                o[l][b][k][x][y]+=
                      i[l][b][c][x+fx][y+fy]
                            * w[l][k][c][fx][fy];
```

Fig. 18.20 Data transformation and hardware mapping example, targeting the Envision hardware configuration

Yet, this specific set of dataflow transformations is not the only possible option. Many possible loop orderings, loop splitting and loop unrolling options could have been exercised to map the specific network layer on the hardware platform. For realistic networks, there can easily be millions of different valid solutions. While all these mappings would be functionally identical, their resulting system level performance and efficiency benchmark won't be. The challenge is hence to try all possible dataflow transformation, quickly assess their impact at the system level, and pick the best one. It is needless to say that this cannot be done by hand, and automated frameworks are required to support such mapping.

At the moment, several frameworks start to emerge to automate such explorations [90–93]. As shown in Fig. 18.21 (top), these frameworks typically take in a neural network layer representation, together with the constraints imposed by the hardware platform. Based on this information, they are capable of efficiently finding all functionally equivalent data transformations supported by the hardware, and computing the resulting number of compute cycles, and memory accesses required within the platform. This information can then be fed to a high-level processor performance model to find the resulting system level performance of the selected mapping. By repeating this for all possible mappings, the framework can derive a Pareto-optimal set of algorithmic mappings or find the best mapping subject to an application level constraint, such as maximum latency. The selected mapping can subsequently be compiled into micro code to be executed on the platform, as for example integrated in the TVM framework [92].

Challenges and outlook: While these frameworks start to emerge, they are still immature, and many challenges remain. Next, the three most critical challenges are discussed: (1) cross-layer mappings; (2) model-HW co-optimization; (3) exploration space bounding.

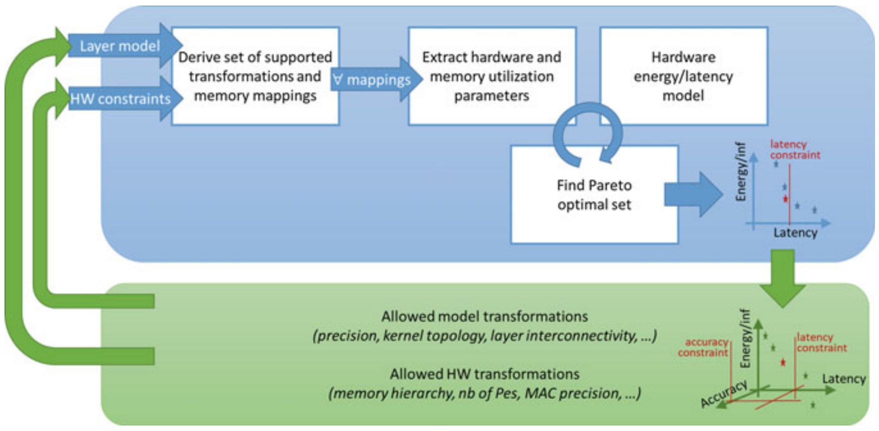

Layer model
HW constraints
Derive set of supported transformations and memory mappings
∀ mappings
Extract hardware and memory utilization parameters
Hardware energy/latency model
Find Pareto optimal set
Energy/inf
latency constraint
Latency
Allowed model transformations
(precision, kernel topology, layer interconnectivity, ...)
Allowed HW transformations
(memory hierarchy, nb of Pes, MAC precision, ...)
accuracy constraint
Energy/inf
latency constraint
Latency
Accuracy

Fig. 18.21 Automated mapping and performance estimation framework assuming a given network model and HW configuration (top), which can be extended with automated neural network model search and optimal hardware topology search (bottom)

1. *Cross-layer mappings*: Current frameworks focus on mapping and scheduling a single neural network layer. This, however, limits the degrees of freedom the mapper has, and excludes interesting solutions such as depth-first network execution, which iterates across layers before executing all tiles of a specific layer [94]. Yet, it is hard to include this into the exploration framework, as it blows up the exploration space.
2. *Model-HW co-optimization*: The framework discussed earlier (Fig. 18.21 (top)) assumes a given network topology, and hardware constellation. Yet, during the design phase, the designer can modify the neural network model, and its computational precision. As shown earlier, many different neural networks can be constructed for the same task achieving the same task accuracy, yet with widely varying hardware mapping consequences. Exploring neural network models, with the hardware mapping tradeoffs in the loop allows to find the optimal neural network topology given the system level benchmarks, instead of just the best algorithmic level benchmarks. This is partially pursued in studies such as MNASnet [23], and the minimum energy QNN study [87], yet still with very crude energy models. Truly integrating this with more realistic hardware models will undoubtedly bring more breakthroughs in the near future.

   When the hardware platform is not decided yet, or the target chip has not yet been taped out, also the hardware configuration can be modified in this iterative exploration loop. As such, the best hardware-model-mapping combination can be found to serve a given task within its application constraints. This is pursued in the Maestro framework [93] and the EyerissV2 studies [41]. Of course, these additional exploration options again increase the search space drastically.
3. *Exploration space bounding*: All aforementioned improvements of the automated exploration, mapping and compilation framework result in yet another increase of the possible exploration space. When only looking at hardware configuration modifications, while keeping the model fixed, the Maestro framework already has to assess millions of design points. On the other hand, also many millions of options have to be searched when only assessing model transformations without considering hardware modifications. It is clear that exhaustively searching this complete design space is simply infeasible. Research towards smart sampling techniques, exploiting Bayesian optimization, or reinforcement learning have been successfully applied to model explorations. It is expected that in the near future they will also start to be successfully applied on joint hardware-model-mapping optimizations. This will undoubtedly give rise to an even more interesting interplay in which novel processor architectures fuel these new dataflow mappings and models, which in turn lead to new processing paradigms.

18.7 Outlook: Towards True Autonomous Intelligence

Looking further out into the future, edge devices will increasingly evolve into truly autonomous intelligent devices: Devices which can not only execute a pre-trained inference model, but can also increase their own knowledge, can reason, and synergistically collaborate with other devices.

Learning at the edge: Several application use cases envision the edge devices to be more than a pure inference engine. The next step is to make the edge NPU also capable of performing update learning on the deployed network model. This capability would allow the edge device to learn for example a user-customized speech interface that works better and better the more it is being used by a specific person. Or, an anomaly detector would be able to use this online training capability to better distinguish anomalies within its specific environment. Many challenges are related to online, in-device learning:

- *At the algorithmic level*, researchers are exploring learning methods that prevent the network to forget previously acquired knowledge [95]. Moreover, researchers are actively exploring whether learning can also be done without the need for full floating-point data types and compute intensive backpropagation steps, e.g. using techniques such as direct feedback alignment [96, 97].
- *At the hardware level*, the support for edge training will require the additional support for higher precision data types within the NPU, and higher precision weight storage. Since the weight matrices have to be read out in transposed form during back-propagation, several recent designs are experimenting with transpose memories, which can be efficiently read out in in a column-parallel manner as well as in a row-parallel scheme [98, 99].
- *At the circuit level*, researchers are looking at ways to embrace emerging resistive memory cells for in-device learning [100]. One direction is to perform standard memory R/W access and to minimize writes to overcome hard endurance limits [101]. Another approach that makes more direct use of the device's physics and treat each device as a "nanokernel" with local feedback during training [102].

Reasoning: Neural networks have shown excellent results in pattern matching and regression tasks, yet they are insufficient towards achieving all intelligence needs of our envisioned future autonomous edge devices. Their main shortcomings are their lack of explainability, their difficulty to integrate expert knowledge or constraints and their inability to support probabilistic reasoning tasks. Other machine learning models, such as Bayesian reasoning, logic reasoning and probabilistic graphical models (PGM, [103]) do possess these features, but come with their own shortcomings, such as their high dataflow irregularity, their inability to efficiently deal with raw data and long training times. Yet, more and more it becomes clear that these two machine learning formalisms form an interesting tandem, in which neural networks can be used as pattern matching layers operating on raw sensor data. The network outputs are then forwarded to reasoning layers on top, which based on these observations make complex decisions in a transparent way. On the algorithmic side, researchers have started to actively explore this using for example Logic Tensor Network models [104], Bayesian Deep Learning models [105] and frameworks such as deep problog [106]. On the hardware side, more challenges are also coming, as the reasoning models are characterized by very different dataflow patterns compared to neural networks, which do not execute efficiently on an NPU, nor CPU or GPU. A new type of processor might yet again have to be invented [107].

Synergistic collaboration: Finally, edge devices are equipped with wireless connections, and hence do not have to operate in isolation. They can exchange data and models among each other, and as such smartly collaborate to perform training and inference on the most suited device at that moment. This will again increase the mapping exploration space discussed in Sect. 18.6 and will now also require incorporating latency and energy complications of sharing data between devices into account into the system cost models. Interestingly, the optimal assignment can change dynamically over time depending on each device's energy availability and current workload, giving rise to real-time scheduling and optimization opportunities. From the hardware side, this will spark an exciting integration of machine learning processors and security hardware, as all models and data that will be exchanged are privacy- and authentication-sensitive.

18.8 Conclusions

Innovations towards more efficient processing of machine learning workloads in edge devices are arriving at a high pace, mostly focused around neural network-based inference. Breakthroughs are realized at the algorithmic level, hardware level and circuit/technology level. Yet, it also becomes increasingly clear that innovations at one level have significant implications at the other levels. As a result, benchmarking initiatives push for system level benchmarks, which jointly consider all levels in an integrated way. To further optimize the complete system stack, integrated frameworks enable to find the most efficient mapping of a neural network model on a given hardware platform. Even one step further, these frameworks can be used to actively explore the algorithmic and hardware design space towards optimal algorithm-hardware co-design. Moreover, new emerging technology options will give rise to very different processor and memory configuration options, and hence new classes of optimal model topologies.

Many challenges remain to effectively enable such cross-layer optimization that covers the complete exploration space, and integrate this in an automated model development, model mapping, and compilation framework. Moreover, workloads will in the future no longer be limited to plain neural network inference but will be expanded with on device learning and the integration with logic and probabilistic reasoning. This will undoubtedly give rise to many more exciting innovations at the algorithmic, architecture and circuit levels.

References

1. A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, M. Ayyash, Internet of things: a survey on enabling technologies, protocols, and applications. IEEE Commun. Surv. Tutorials **17**(4), 2347–2376 (2015)
2. M. Satyanarayanan, P. Simoens, Y. Xiao, P. Pillai, Z. Chen, K. Ha, W. Hu, B. Amos, Edge analytics in the internet of things. IEEE Pervasive Comput. **14**(2), 24–31 (2015)
3. H. Li, K. Ota, M. Dong, Learning IoT in edge: deep learning for the Internet of Things with edge computing. IEEE Netw. **32**(1), 96–101 (2018)
4. A. Canziani, A. Paszke, E. Culurciello, An Analysis of Deep Neural Network Models for Practical Applications. arXiv preprint arXiv:1605.07
5. Caulfield, A.M., Chung, E.S., Putnam, A., Angepat, H., Fowers, J., Haselman, M., Lo, D. et al., A cloud-scale acceleration architecture, in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture* (IEEE Press, 2016), p. 7
6. N. Strom, Scalable distributed DNN training using commodity GPU cloud computing, in *Sixteenth Annual Conference of the International Speech Communication Association* (2015)
7. Tractica report, *Deep Learning Chipsets* (2018). https://www.tractica.com/research/deep-learning-chipsets/
8. Semiconductor Engineering, *AI Chip Architectures Race To The Edge* (2018). https://semiengineering.com/ai-chip-architectures-race-to-the-edge/
9. K. Guo, W. Li, K. Zhong, Z. Zhu, S. Zeng, S. Han, Y. Xie, P. Debacker, M. Verhelst, Y. Wang, Neural Network Accelerator Comparison. [Online]. https://nicsefc.ee.tsinghua.edu.cn/projects/neural-network-accelerator/
10. I. Goodfellow, Y. Bengio, A. Courville, *Deep Learning* (MIT press, Cambridge, 2016)
11. C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, Z. Wojna, Rethinking the inception architecture for computer vision (2015). arXiv preprint arXiv:1512.00567
12. F. Chollet, *Xception: Deep Learning with Depthwise Separable Convolutions* (2016). arXiv preprint arXiv:1610.02357
13. A. Howard et al., MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications (2017). arXiv:1704.04861
14. M. Sandler et al., *MobileNetV2: Inverted Residuals and Linear Bottlenecks*. arXiv:1801.04381
15. C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, K. Murphy, Progressive neural architecture search, in *ECCV2018*
16. E. Real, A. Aggarwal, Y. Huang, Q.V. Le, Regularized evolution for image classifier architecture search, in *The Thirty-Third AAAI Conference on Artificial Intelligence* (2019)
17. S. Xie, A. Kirillov, R. Girshick, K. He, *Exploring Randomly Wired Neural Networks for Image Recognition* (2019). arXiv:1904.01569
18. X. Chu, B. Zhang, J. Li, Q. Li, R. Xu, ScarletNAS: Bridging the Gap Between Scalability and Fairness in Neural Architecture Search (2019). arXiv:1908.06022
19. X. Zhang, Z. Li, C. Change Loy, D. Lin, *PolyNet: A Pursuit of Structural Diversity in Very Deep Networks* (2019). arXiv:1611.05725
20. Google's AutoML, https://research.googleblog.com/2017/11/automl-for-large-scale-image.html?m=1
21. Q. Yao et al., Taking the Human out of Learning Applications: A Survey on Automated Machine Learning. arXiv: 1810.13306
22. Y. He, J. Lin, Z. Liu, H. Wang, L.J. Li, S. Han, Amc: Automl for model compression and acceleration on mobile devices, in *Proceedings of the European Conference on Computer Vision (ECCV)* (2018), pp. 784–800
23. M. Tan, B. Chen, R. Pang, V. Vasudevan, Q.V. Le, Mnasnet: Platform-aware neural architecture search for mobile (2018). arXiv preprint arXiv:1807.11626
24. T.-J. Yang, et al., Netadapt: platform-aware neural network adaptation for mobile applications, in *ECCV* (2018)
25. J.A. Suykens, J. Vandewalle, Least squares support vector machine classifiers. Neural Process. Lett. **9**(3), 293–300 (1999)

26. S. Gupta, A. Agrawal, K. Gopalakrishnan, P. Narayanan, Deep learning with limited numerical precision, in *CoRR*, vol. abs/1502.02551 (2015)
27. M. Courbariaux, Y. Bengio, J.-P. David, Training deep neural networks with low precision multiplications (2014). arXiv preprint arXiv:1412.7024
28. R. Krishnamoorthi, Quantizing deep convolutional networks for efficient inference: a whitepaper. arXiv:1806.08342
29. M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, Y. Bengio, Binarized neural networks: training deep neural networks with weights and activations constrained to $+1$ or -1 (2016). arXiv preprint arXiv:1602.02830
30. N. Mellempudi, A. Kundu, D. Das, D. Mudigere, B. Kaul, Mixed low-precision deep learning inference using dynamic fixed point (2017). arXiv preprint arXiv:1701.08978
31. I. Hubara et al., Quantized neural networks: training neural networks with low precision weights and activations. ArXiv1609.07061
32. B. Jacob et al., Quantization and training of neural networks for efficient integer-arithmetic-only inference, in *CVPR* (2018)
33. M. Nagel, M. van Baalen, T. Blankevoort, M. Welling, Data-free quantization (DFQ) through weight equalization and bias correction (2019). arXiv:1906.04721v1
34. E. Meller, A. Finkelstein, U. Almog, M. Grobman, Same, same but different—recovering neural network quantization error through weight factorization (2019). arxiv:1902.01917
35. B. Moons, K. Goetschalckx, N. Van Berckelaer, M. Verhelst, Minimum energy quantized neural networks (2017). arXiv preprint arXiv:1711.00215
36. S. Han, J. Pool, J. Tran, W. Dally, Learning both weights and connections for efficient neural network, in *Advances in Neural Information Processing Systems* (2015), pp. 1135–1143
37. J. Xue, J. Li, Y. Gong, Restructuring of deep neural network acoustic models with singular value decomposition, in *INTERSPEECH* (2013)
38. T.-J. Yang, Y.-H. Chen, V. Sze, Designing energy-efficient convolutional neural networks using energy-aware pruning, in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2017)
39. W. Wei, Learning structured sparsity in deep neural networks, in *NIPS2016*
40. S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M.A. Horowitz, W.J. Dally, EIE: efficient inference engine on compressed deep neural network (2016). arXiv preprint arXiv:1602.01528
41. Y.-H. Chen et al., Eyeriss v2: a flexible accelerator for emerging deep neural networks on mobile devices, in *JETCAS* (2019)
42. Y. Liu, Y. Wang, R. Yu, M. Li, V. Sharma, Y. Wang, Optimizing CNN model inference on CPUs (2018). arXiv: 1809.02697
43. S. Markidis, S.W. Der Chien, E. Laure, I.B. Peng, J.S. Vetter, Nvidia tensor core programmability, performance & precision, in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (IEEE, May 2018), pp. 522–531
44. B. Moons, D. Bankman, M. Verhelst, *Embedded Deep Learning: Algorithms, Architectures and Circuits for Always-on Neural Network Processing* (Springer, 2019). ISBN 978-3-319-99223-5
45. B. Moons, R. Uytterhoeven, W. Dehaene, M. Verhelst, Envision: A 0.26-to-10tops/w subword-parallel dynamic-voltage-accuracy-frequency-scalable convolutional neural network processor in 28 nm fdsoi, in *2017 IEEE International Solid-State Circuits Conference (ISSCC)* (IEEE, 2017), pp. 246–247
46. N.P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, In-datacenter performance analysis of a tensor processing unit, in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)* (IEEE), June 2017, pp. 1–12
47. B. Moons, D. Bankman, L. Yang, B. Murmann, M. Verhelst, BinarEye: an always-on energy-accuracy-scalable binary CNN processor with all memory on chip in 28 nm CMOS, in *IEEE Custom Integrated Circuits Conference (CICC)* (2018), pp. 1–4
48. Y.H. Chen, T. Krishna, J.S. Emer, V. Sze, Eyeriss: an energy-efficient reconfigurable accelerator for deep convolutional neural networks. IEEE J. Solid-State Circ. **52**(1), 127–138 (2016)

49. G. Desoli, N. Chawla, T. Boesch, S. Singh, E. Guidetti, F. De Ambroggi, T. Majo, P. Zambotti, M. Ayodhyawasi, H. Singh, N. Aggarwal, A 2.9 TOPS/W deep convolutional neural network SoC in FD-SOI 28 nm for intelligent embedded systems
50. A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S.W. Keckler, W.J. Dally, SCNN: an accelerator for compressed-sparse convolutional neural networks, in *Proceedings of ISCA '17*, Toronto, ON, Canada, 24–28 June 2017
51. M. Nikolić, M. Mahmoud, A. Moshovos, Y. Zhao, R. Mullins, Characterizing sources of ineffectual computations in deep learning networks, in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (IEEE, 2019), pp. 165–176
52. V. Camus, C. Enz, M. Verhelst, Survey of precision-scalable multiply-accumulate units for neural-network processing, in *2019 IEEE 1st International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, Mar 2019
53. S. Cosemans, Advanced memory, logic and 3D technologies for in-memory computing and machine learning, in *ISSCC2019 Forum Talk*
54. L. Mei, M. Dandekar, D. Rodopoulos, J. Constantin, P. Debacker, R. Lauwereins, M. Verhelst, Sub-word parallel precision-scalable MAC engines for efficient embedded DNN inference, in *2019 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)* (IEEE, 2019), pp. 6–10
55. L. Mei, V. Camus, C. Enz, M. Verhelst, Review and benchmarking of precision-scalable multiply-accumulate unit architectures for embedded neural-network processing, in *2020 IEEE Journal on Emerging and Selected Topics in Circuits and Systems* (2020)
56. D. Shin, J. Lee, J. Lee, H.-J. Yoo, *DNPU: An 8.1 TOPS/W Reconfigurable CNN-RNN Processor for General-Purpose Deep Neural Networks*
57. B. Moons, R. Uytterhoeven, W. Dehaene, M. Verhelst, DVAFS: trading computational accuracy for energy through dynamic-voltage-accuracy-frequency-scaling, in *Design, Automation & Test in Europe Conference & Exhibition (DATE)* (IEEE, 2017), pp. 488–493
58. Sharma et al., BitFusion: bit-level dynamically composable architecture for accelerating deep neural networks, in *ISCA18*
59. J. Lee, C. Kim, S. Kang, D. Shin, S. Kim, H.J. Yoo, UNPU: A 50.6TOPS/W unified deep neural network accelerator with 1b-to-16b fully-variable weight bit-precision, in *2018 IEEE International Solid-State Circuits Conference (ISSCC)* (2018), pp. 218–220
60. S. Sharifymoghaddam et al., Loom: exploiting weight and activation precisions to accelerate convolutional neural networks, in *DAC Conference* (2018)
61. L. Liu, J. Deng, *Dynamic Deep Neural Networks: Optimizing Accuracy-Efficiency Trade-offs by Selective Execution* (2017). arXiv preprint arXiv:1701.00299
62. A. Coucke, M. Chlieh, T. Gisselbrecht, D. Leroy, M. Poumeyrol, T. Lavril, Efficient keyword spotting using dilated convolutions and gating, in *ICASSP 2019–2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (IEEE, 2019), pp. 6351–6355
63. Z. Yan, X. Li, M. Li, W. Zuo, S. Shan, Shift-net: image inpainting via deep feature rearrangement, in *Proceedings of the European Conference on Computer Vision (ECCV)* (2018), pp. 1–17
64. A.V.D. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, K. Kavukcuoglu, Wavenet: a generative model for raw audio (2016). arXiv preprint arXiv:1609.03499
65. R.A. Nawrocki, R.M. Voyles, S.E. Shaheen, A mini review of neuromorphic architectures and implementations. IEEE Trans. Electron Devices **63**(10), 3819–3829 (2016)
66. C. Mead, Neuromorphic electronic systems. Proc. IEEE **78**(10), 1629–1636 (1990)
67. E.A. Vittoz, Future of analog in the VLSI environment, in *IEEE International Symposium on Circuits and Systems* (1990), pp. 1372–1375
68. P.A. Merolla, J.V. Arthur, R. Alvarez-Icaza, A.S. Cassidy, J. Sawada, F. Akopyan, B.L. Jackson, N. Imam, C. Guo, Y. Nakamura, B. Brezzo, I. Vo, S.K. Esser, R. Appuswamy, B. Taba, A. Amir, M.D. Flickner, W.P. Risk, R. Manohar, D.S. Modha, A million spiking-neuron integrated circuit with a scalable communication network and interface. Science **345**(6197), 668–673 (2014)

69. B. Murmann, D. Bankman, E. Chai, D. Miyashita, L. Yang, Mixed-signal circuits for embedded machine-learning applications, in *Asilomar Conference on Signals, Systems and Computers* (Nov 2015), Asilomar, CA
70. D. Bankman, B. Murmann, An 8-bit, 16 input, 3.2 pJ/op switched-capacitor dot product circuit in 28-nm FDSOI CMOS, in *Proceedings of IEEE Asian Solid-State Circuits Conference* (Nov 2016), Toyama, Japan, pp. 21–24
71. A.S. Rekhi, B. Zimmer, N. Nedovic, N. Liu, R. Venkatesan, M. Wang, B. Khailany, W.J. Dally, C.T. Gray, Analog/mixed-signal hardware error modeling for deep learning inference, in *Proceedings of Design Automation Conference* (2019), pp. 1–6
72. V. Sze, Y. Chen, J. Emer, A. Suleiman, Z. Zhang, Hardware for machine learning: challenges and opportunities,in *IEEE Custom Integrated Circuits Conference (CICC)* (2017), Austin, TX, pp. 1–8
73. D. Bankman, L. Yang, B. Moons, M. Verhelst, B. Murmann, An always-on 3.8 uJ/86% CIFAR-10 mixed-signal binary CNN processor with all memory on chip in 28-nm CMOS. IEEE J. Solid-State Circ. **54**(1), 158–172 (2019)
74. W.H. Kautz, Cellular logic-in-memory arrays. IEEE Trans. Comput. **C-18**(8), 719–727 (1969)
75. H. Valavi, P.J. Ramadge, E. Nestler, N. Verma, A 64-tile 2.4-Mb in-memory-computing CNN accelerator employing charge-domain compute. IEEE J. Solid-State Circ. **54**(6), 1789–1799 (2019)
76. N. Verma et al., In-memory computing: advances and prospects. IEEE Solid-State Circ. Mag. **11**(3), 43–55 (2019)
77. H. Jia, Y. Tang, H. Valavi, J. Zhang, N. Verma, A microprocessor implemented in 65 nm CMOS with configurable and bit-scalable accelerator for programmable in-memory computing (2018). arXiv preprint, arXiv:1811.04047
78. H. Tsai, S. Ambrogio, P. Narayanan, R.M. Shelby, G.W. Burr, Recent progress in analog memory-based accelerators for deep learning. J. Phys. D Appl. Phys. **51**(28), 283001 (2018)
79. S. Mittal, A survey of ReRAM-based architectures for processing-in-memory and neural networks, in *Machine Learning & Knowledge Extraction* (2018)
80. M. Dazzi, A. Sebastian, P.A. Francese, T. Parnell, L. Benini, E. Eleftheriou, 5 parallel prism: a topology for pipelined implementations of convolutional neural networks using computational memory (2019). arXiv preprint, arXiv:1906.03474
81. Y. Lin et al., Performance impacts of analog ReRAM Non-ideality on neuromorphic computing. IEEE Trans. Electron Devices **66**(3), 1289–1295 (2019)
82. S. Yin, X. Sun, S. Yu, J.S. Seo, High-throughput in-memory computing for binary deep neural networks with monolithically integrated RRAM and 90 nm CMOS (2019). arXiv preprint arXiv:1909.07514
83. B. Murmann, ADC performance survey 1997–2019, [Online]. http://web.stanford.edu/~murmann/adcsurvey.html
84. W.J. Dally et al., Hardware-enabled artificial intelligence, in *Symposium on VLSI Circuits* (2018), pp. 1–2
85. D. Bankman, J. Messner, A. Gural, B. Murmann, RRAM-based in-memory computing for embedded deep neural networks, in *Asilomar Conference on Signals, Systems and Computers*, Asilomar, CA, Nov 2019
86. https://mlperf.org/
87. B. Moons, K. Goetschalckx, N. Van Berckelaer, M. Verhelst, Minimum energy quantized neural networks. arXiv preprint arXiv:1711.00215
88. B. Moons, D. Bankman, L. Yang, B. Murmann, M. Verhelst, BinarEye: an always-on energy-accuracy-scalable binary CNN processor with all memory on chip in 28 nm CMOS, in *Custom Integrated Circuits Conference (CICC)* (IEEE, 2018), pp. 1–4
89. A. Stoutchinin, F. Conti, L. Benini, Optimally scheduling CNN convolutions for efficient memory access (2019). arXiv preprint arXiv:1902.01492
90. X. Yang, M. Gao, J. Pu, A. Nayak, A. Liu, S.E. Bell, J.O. Setter, K. Cao, H. Ha, C. Kozyrakis, M. Horowitz, DNN dataflow choice is overrated (2018). arXiv preprint arXiv:1809.04070

91. A. Parashar, P. Raina, Y.S. Shao, Y.H. Chen, V.A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S.W. Keckler, J. Emer, Timeloop: a systematic approach to DNN accelerator evaluation, in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (IEEE, 2019), pp. 304–315
92. https://tvm.ai/
93. H. Kwon, M. Pellauer, T. Krishna, Maestro: an open-source infrastructure for modeling dataflows within deep learning accelerators (2018). arXiv preprint arXiv:1805.02566
94. K. Goetschalckx, M. Verhelst, Breaking high resolution CNN bandwidth barriers with enhanced depth-first execution, in *JETCAS 2019*
95. R. Aljundi, F. Babiloni, M. Elhoseiny, M. Rohrbach, T. Tuytelaars, Memory aware synapses: learning what (not) to forget, in *Proceedings of the European Conference on Computer Vision (ECCV)* (2018), pp. 139–154
96. A. Nøkland, Direct feedback alignment provides learning in deep neural networks, in *Advances in Neural Information Processing Systems* (2016), pp. 1037–1045
97. C. Frenkel, M. Lefebvre, D. Bol, Learning without feedback: direct random target projection as a feedback-alignment algorithm with layerwise feedforward training (2019). arXiv preprint arXiv:1909.01311
98. J. Yue, R. Liu, W. Sun, Z. Yuan, Z. Wang, Y.N. Tu, Y.-J. Chen, A. Ren, Y. Wang, M.-F. Chang, X. Li, H. Yang, Y. Liu, 7.5 A 65 nm 0.39-to-140.3 TOPS/W 1-to-12b unified neural network processor using block-circulant-enabled transpose-domain acceleration with 8.1 $\times$ Higher TOPS/mm 2 and 6T HBST-TRAM-based 2D data-reuse architecture, in *2019 IEEE International Solid-State Circuits Conference-(ISSCC)* (IEEE, 2019), pp. 138–140
99. D. Han, J. Lee, J. Lee, H.J. Yoo, A low-power deep neural network online learning processor for real-time object tracking application. IEEE Trans. Circuits Syst. I Regul. Pap. **66**(5), 1794–1804 (2018)
100. S. Yu, Neuro-inspired computing with emerging nonvolatile memory. Proc. IEEE **106**, 260–285 (2018)
101. A. Gural et al., Low-rank training of deep neural networks for emerging memory technology, unpublished work
102. H. Li, P. Raina, H.-S. P. Wong, Neuro-inspired computing with emerging memories: where device physics meets learning algorithms, in *Proceedings of SPIE 11090, Spintronics XII, 110903L*, Sep 2019
103. L.E. Sucar, Probabilistic graphical models, in *Advances in Computer Vision and Pattern Recognition* (Springer London, London, 2015)
104. L. Serafini, A.D.A. Garcez, Logic tensor networks: deep learning and logical reasoning from data and knowledge (2016). arXiv preprint arXiv:1606.04422
105. H. Wang, D.Y. Yeung, Towards Bayesian deep learning: a framework and some existing methods. IEEE Trans. Knowl. Data Eng. **28**(12), 3395–3408 (2016)
106. R. Manhaeve, S. Dumancic, A. Kimmig, T. Demeester, L. De Raedt, Deepproblog: neural probabilistic logic programming, in *Advances in Neural Information Processing Systems* (2018), pp. 3749–3759
107. N. Shah, L. Galindez, W. Meert, M. Verhelst, Acceleration of probabilistic reasoning through custom processor architecture and compiler, in *Design and Test Conference Europe (DATE)* (2020)