ShiDianNao: Shifting Vision Processing Closer to the Sensor

Zidong Du[†♯] Robert Fasthuber[‡] Tianshi Chen[†] Paolo Ienne[‡] Ling Li[†] Tao Luo[†♯]
Xiaobing Feng[†] Yunji Chen[†] Olivier Temam[§]
[†]State Key Laboratory of Computer Architecture, Institute of Computing Technology (ICT), CAS, China
[♯]University of CAS, China [‡]EPFL, Switzerland [§]Inria, France
{duzidong,chentianshi,liling,luotao,fxb,cyj}@ict.ac.cn
{robert.fasthuber,paolo.ienne}@epfl.ch olivier.temam@inria.fr

Abstract

In recent years, neural network accelerators have been shown to achieve both high energy efficiency and high performance for a broad application scope within the important category of recognition and mining applications.

Still, both the energy efficiency and performance of such accelerators remain limited by memory accesses. In this paper, we focus on image applications, arguably the most important category among recognition and mining applications. The neural networks which are state-of-the-art for these applications are Convolutional Neural Networks *(CNN), and they have an important property: weights are shared among many neurons, considerably reducing the neural network memory footprint. This property allows to entirely map a CNN within an SRAM, eliminating all DRAM accesses for weights. By further hoisting this accelerator next to the image sensor, it is possible to eliminate all remaining DRAM accesses, i.e., for inputs and outputs.*

In this paper, we propose such a CNN accelerator, placed next to a CMOS or CCD sensor. The absence of DRAM accesses combined with a careful exploitation of the specific data access patterns within CNNs allows us to design an accelerator which is 60× *more energy efficient than the previous state-of-the-art neural network accelerator. We present a full design down to the layout at 65 nm, with a modest footprint of 4.86 mm*^{2} *and consuming only 320 mW, but still about* 30× *faster than high-end GPUs.*

1. Instructions

In the past few years, accelerators have gained increasing attention as an energy and cost effective alternative to CPUs and GPUs [20, 57, 13, 14, 60, 61]. Traditionally, the main downside of accelerators is their limited application scope, but recent research in both academia and industry has highlighted the remarkable convergence of trends towards recognition and mining applications [39] and the fact that a very small corpus of algorithms—i.e., neural network based algorithms—can tackle a significant share of these applications [7, 41, 24]. This makes it possible to realize the best of both worlds: accelerators with high performance/efficiency and yet broad application scope. Chen et al. [3] leveraged this fact to propose neural network accelerators; however, the authors also acknowledge that, like many processing architectures, their accelerator efficiency and scalability remains severely limited by memory bandwidth constraints.

That study aimed at supporting the two main state-of-the-art neural networks: *Convolutional Neural Networks (CNNs)* [35] and *Deep Neural Networks (DNNs)* [32, 48]. Both types of networks are very popular, with DNNs being more general than CNNs due to one major difference: in CNNs, it is assumed that each neuron (of a feature map, see later Section 3 for more details) *shares* its weights with all other neurons, making the total number of weights far smaller than in DNNs. For instance, the largest state-of-the-art CNN has 60 millions weights [29] versus up to 1 billion [34] or even 10 billions [6] for the largest DNNs. Such weight sharing property directly derives from the CNN application scope, i.e., vision recognition applications: since the set of weights feeding a neuron characterizes the *feature* this neuron should recognize, sharing weights is simply a way to express that any feature can appear anywhere within an image [35], i.e., translation invariance.

Now, this simple property can have profound implications for architects: It is well known that the highest energy expense is related to data movement, in particular DRAM accesses, rather than computation [20, 28]. Due to its small weights memory footprint, it is possible to store a whole CNN within a small SRAM next to computational operators, and as a result, there is no longer a need for DRAM memory accesses to fetch the model (weights) in order to process each input. The only remaining DRAM accesses become those needed to fetch the input image. Unfortunately, when the input is as large as an image, this would still constitute a large energy expense.

However, CNNs are dedicated to image applications, which, arguably, constitute one of the broadest categories of recog-

ISCA'15, June 13 - 17, 2015, Portland, OR, USA

http://dx.doi.org/10.1145/2749469.2750389

nition applications (followed by voice recognition). In many real-world and embedded applications—e.g., smartphones, security, self-driving cars—the image directly comes from a CMOS or CCD sensor. In a typical imaging device, the image is acquired by the CMOS/CCD sensor, sent to DRAM, and later fetched by the CPU/GPU for recognition processing. The small size of the CNN accelerator (computational operators and SRAM holding the weights) makes it possible to hoist it next to the sensor, and only send the few output bytes of the recognition process (typically, an image category) to DRAM or the host processor, thereby almost entirely eliminating energy costly accesses to/from memory.

In this paper, we study an energy-efficient design of a visual recognition accelerator to be directly embedded with any CMOS or CCD sensor, and fast enough to process images in real time. Our accelerator leverages the specific properties of CNN algorithms, and as a result, it is 60× more energy efficient than DianNao [3], which was targeting a broader set of neural networks. We achieve that level of efficiency not only by eliminating DRAM accesses, but also by carefully minimizing data movements between individual processing elements, from the sensor, and the SRAM holding the CNN model. We present a concrete design, down to the layout, in a 65 *nm* CMOS technology with a peak performance of 194 GOP/s (billions of fixed-point OPerations per second) at 4.86 mm^2, 320.10 *mW*, and 1 GHz. We empirically evaluate our design on ten representative benchmarks (neural network layers) extracted from state-of-the-art CNN implementations. We believe such accelerators can considerably lower the hardware and energy cost of sophisticated vision processing, and thus help make them widespread.

The rest of this paper is organized as follows. In Section 2 we precisely describe the integration conditions we target for our accelerator, which are determinant in our architectural choices. Section 3 is a primer on recent machine-learning techniques where we introduce the different types of CNN layers. We give a first idea of the mapping principles in Section 4 and, in Sections 5 to 7, we introduce the detailed architecture of our accelerator (ShiDianNao, Shi for vision and DianNao for electronic brain) and discuss design choices. In Section 8, we show how to map CNNs on ShiDianNao and schedule the different layers. In Section 9, the experimental methodology is described. In Section 10, we implemented ShiDianNao and compared the results to the state-of-the-art in terms of performance and hardware costs. In Section 12, conclusions are given. Related work is discussed in Section 11.

2. System Integration

Figure 1 shows a typical integration solution for cheap cameras (closely resembling an STM chipset [55, 56]): An image processing chip is connected to cameras (in typical smartphones, two) streaming their data through standard *Camera Serial Interfaces (CSIs)*. Video processing pipelines, controlled by a microcontroller unit, implement a number of essential func-

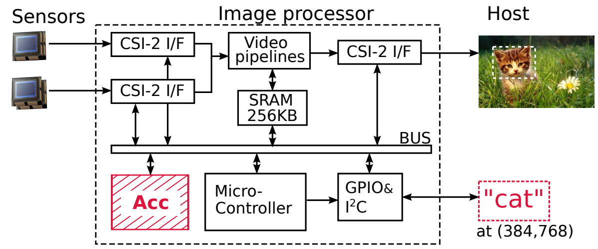

Figure 1: Possible integration of our accelerator in a commercial image processing chip.

tions such as Bayer reconstruction, white balance and barrel correction, noise and defect filtering, autofocus control, video stabilization, and image compression. More advanced processors already implement rudimentary object detection and tracking functions, such as face recognition [56].

The figure shows also the approximate setting of our ShiDianNao accelerator, with high-level control from the embedded microcontroller and using for image input the same memory buffers. Contrary to the fairly elementary processing of the video pipelines, our accelerator is meant to achieve very significantly more advanced classification tasks. One should notice that, to contain cost and energy consumption, this type of commercial image processors go a long way to avoid full-image buffering (which, of course, for 8-megapixel images would require several megabytes of storage): input and outputs of the system are through serial streaming channels, there is no interface to external DRAM, and the local SRAM storage is very limited (e.g., 256 KB [55]). These constraints are what we are going to use to design our system: our recognition system must also avoid full-frame storage, exclude any external DRAM interface, and process sequentially a stream of partial frame sections as they flow from the sensors to the application processor. We will later show (Section 10.2) that a local storage of 256 KB appears appropriate also for our accelerator.

3. Primer on Convolutional Neural Networks

Convolutional Neural Networks (CNNs) [35] and Deep Neural Networks (DNNs) [32] are known as two state-of-the-art machine learning algorithms. Both of them belong to the family of Multi-Layer Perceptrons (MLPs) [21] and may consist of four types of layers: convolutional, pooling, normalization, and classifier layers. However, the two network types differ from each other in their convolutional layers—the type of layers dominate the execution time of both types of networks [3]. In a CNN convolutional layer, synaptic weights can be reused (shared) by certain neurons, while there is no such reuse in a DNN convolutional layer (see below for details). The data reuse in CNN naturally favors hardware accelerators, because it reduces the number of synaptic weights to store, possibly allowing them to be simultaneously kept on-chip.

General Architecture. Figure 2 illustrates the architecture of LeNet-5 [35], a representative CNN widely used in document recognition. It consists of two convolutional layers (C1 and C3 in Figure 2), two pooling layers (S2 and S4 in

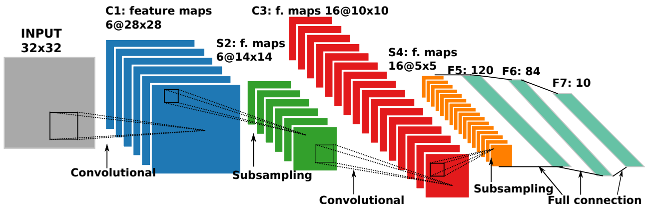

Figure 2: A representative CNN architecture—LeNet5 [35]. C: Convolutional layer; S: Pooling layer; F: Classifier layer.

Figure 2), and three classifier layers (F5, F6 and F7 in Figure 2). Recent studies also suggest the use of normalization layers in deep learning [29, 26].

Convolutional Layer. A convolutional layer can be viewed as a set of local filters designed for identifying certain characteristics of input feature maps (i.e., 2D arrays of input pixels/neurons). Each local filter has a *kernel* having $K_x \times K_y$ coefficients, and processes a *convolutional window* capturing $K_x \times K_y$ input neurons in one input feature map (or multiple same-sized windows in multiple input feature maps). A 2D array of local filters produces an output feature map, where each local filter corresponds to an output neuron, and convolutional windows of adjacent output neurons are sliding by steps of S_x (x-direction) and S_y (y-direction) in the same input feature map. Formally, the output neuron at position (a,b) of the output feature map #mo is computed with

$$O^{mo}_{a,b} = f\left(\sum_{mi \in A_{mo}} \left(\beta^{mi,mo} + \sum_{i=0}^{K_x-1}\sum_{j=0}^{K_y-1} \omega^{mi,mo}_{i,j} \times I^{mi}_{aS_x+i,bS_y+j}\right)\right), \quad (1)$$

where $\omega^{mi,mo}$ is the *kernel* between input feature map #mi and output feature map #mo, $\beta^{mi,mo}$ is the bias value for the pair of input and output feature maps, A_{mo} is the set of input feature maps connected to output feature map #mo, and $f(\cdot)$ is the non-linear activation function (e.g., *tanh* or *sigmoid*).

Pooling Layer. A pooling layer directly downsamples an input feature map by performing maximum or average operations to non-overlapping windows of input neurons (i.e., *pooling window*, each with $K_x \times K_y$ neurons) in the feature map. Formally, the output neuron at position (a,b) of the output feature map #mo is computed with

$$O^{mo}_{a,b} = \max_{0 \le i < Kx, 0 \le j < Ky}\left(I^{mi}_{a+i,b+j}\right), \quad (2)$$

where $mo = mi$ because the mapping between input and output feature maps is one-to-one. The case shown above is that of max pooling, while average pooling is similar except that the maximum operation is replaced with the average operation. Traditional CNNs also additionally perform a non-linear transformation on the above output, while recent studies no longer suggest that [36, 26].

Normalization Layers. Normalization layers introduce competition between neurons at the same position of different input feature maps, which further improves the recognition accuracy of CNN. There exist two types of normalization layers: *Local Response Normalization (LRN)* [29] and *Local Contrast Normalization (LCN)* [26]. In an LRN layer, the output neuron

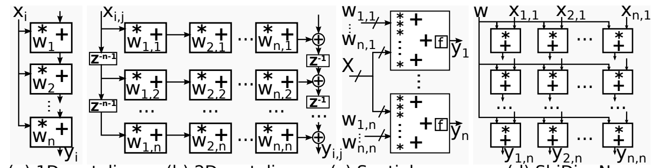

Figure 3: Typical implementations of neural networks.

at position (a,b) of output feature map #mi can be computed with

$$O^{mi}_{a,b} = I^{mi}_{a,b} / \left(k + \alpha \times \sum_{j=\max(0,mi-M/2)}^{\min(Mi-1,mi+M/2)} (I^{j}_{a,b})^2\right)^{\beta}, \quad (3)$$

where Mi is the total number of input feature maps, M is the maximum number of input feature maps connected to one output feature map, and α, β and k are constant parameters.

In an LCN layer, the output neuron at position (a,b) of output feature map #mi can be computed with

$$O^{mi}_{a,b} = v^{mi}_{a,b} / \max\left(\text{mean}(\delta_{a,b}), \delta_{a,b}\right), \quad (4)$$

where $\delta_{a,b}$ is computed with

$$\delta_{a,b} = \sqrt{\sum_{mi,a,b} (v^{mi}_{a+p,b+q})^2}, \quad (5)$$

$v^{mi}_{a,b}$ (subtractive normalization) is computed with

$$v^{mi}_{a,b} = I^{mi}_{a,b} - \sum_{j,a,b} \omega_{a,b} \times I^{j}_{a+p,b+q}, \quad (6)$$

$\omega_{a,b}$ is a normalized Gaussian weighting window satisfying $\sum_{a,b} \omega_{a,b} = 1$, and $I^{mi}_{a,b}$ is the input neuron at position (a,b) of the input feature map #mi.

Classifier Layer. After a sequence of other layers, a CNN integrates one or more classifier layers to compute the final result. In a typical classifier layer, output neurons are fully connected to input neurons with independent synapses. Formally, the output neuron #no is computed with

$$O^{no} = f\left(\beta^{no} + \sum_{ni} \omega^{ni,no} \times I^{ni}\right), \quad (7)$$

where $\omega^{ni,no}$ is the synapse between input neuron #ni and output neuron #no, β^{no} is the bias value of output neuron #no, and $f(\cdot)$ is the activation function.

Recognition vs. Training. A common misconception about neural networks is that they must be trained on-line to achieve high recognition accuracy. In fact, for visual recognition, off-line training (explicitly splitting training and recognition phases) has been proven sufficient, and this fact has been widely acknowledged by machine learning researchers [16, 3]. Off-line training by the service provider is essential for inexpensive embedded sensors, with their limited computational capacity and power budget. We will naturally focus our design on the recognition phase alone of CNNs.

4. Mapping Principles

Roughly, a purely spatial hardware implementation of a neural network would devote a separate accumulation unit for each neuron and a separate multiplier for each synapse. From the early days of neural networks in the 80's and 90's, architects have imagined that concrete applications would contain too

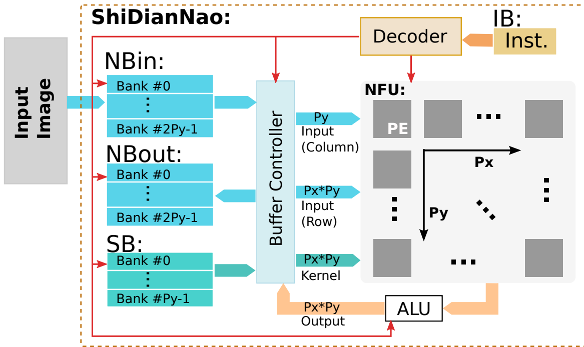

Figure 4: Accelerator architecture.

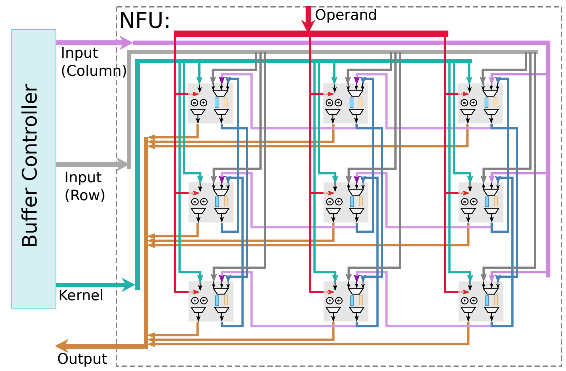

Figure 5: NFU architecure.

many neurons and synapses to be implemented as a singe deep and wide pipeline. Even though an amazing progress has since been achieved in transistor densities, current practical CNNs clearly still exceed the potentials for a pure spatial implementation [57], driving the need for some temporal partitioning of the complete CNN and sequential mapping of the partitions on the physical computational structure.

Various mapping strategies have been attempted and reviewed (see Ienne et al. for an early taxonomy [25]): products, prototypes, and paper designs have probably exhausted all possibilities, including attributing each neuron to a processing element, each synapes to a processing element, and flowing in a systolic fashion both kernels and input feature maps. Some of these principal choices are represented in Figure 3. In this work, we have naturally decided to rely on (1) the 2D nature of our processed data (images) and (2) the limited size of the convolutional kernels. Overall, we have chosen the mapping in Figure 3(d): our processing elements (i) represent neurons, (ii) are organized in a 2D mesh, (iii) receive, broadcasted, kernel elements $\omega_{i,j}$, (iv) receive through right-left and up-down shifts the input feature map, and finally (v) accumulate locally the resulting output feature map.

Of course, the details of the mapping go well beyond the intuition of Figure 3(d) and we will devote the complete Section 8 to show how all the various layers and phases of the computation can fit our architecture. Yet, for now, the figure should give the reader a sufficient broad idea of the mapping to follow the development of the architecture in the next section.

5. Accelerator Architecture: Computation

As illustrated in Figure 4, our accelerator consists of the following main components: two buffers for input and output neurons (NBin and NBout), a buffer for synapses (SB), a neural functional unit (NFU) plus an arithmetic unit (ALU) for computing output neurons, and a buffer and a decoder for instructions (IB). In the rest of this section, we introduce the computational structures, and in the next ones we describe the storage and control structures.

Our accelerator has two functional units, an NFU accommodating fundamental neuron operations (multiplications, additions and comparisons) and an ALU performing activation function computations. We use *16-bit fixed-point arithmetic operators* rather than conventional 32-bit floating-point operators in both computational structures, and the reasons are twofold. First, using 16-bit fixed-point operators brings in negligible accuracy loss to neural networks, which has been validated by previous studies [3, 10, 57]. Second, using smaller operators significantly reduces the hardware cost. For example, a 16-bit truncated fixed-point multiplier is 6.10× smaller and 7.33× more energy-efficient than a 32-bit floating-point multiplier in TSMC 65 *nm* technology [3].

5.1. Neural Functional Unit

Our accelerator processes 2D feature maps (images), thus its NFU must be optimized to handle 2D data (neuron/pixel arrays). The functional unit of DianNao [3] is inefficient for this application scenario, because it treats 2D feature maps as a 1D vector and cannot effectively exploit the locality of 2D data. In contrast, our NFU is a 2D mesh of $P_x \times P_y$ Processing Elements (PEs), which naturally suits the topology of 2D feature maps.

An intuitive way of neuron-PE mapping is to allocate a block of $K_x \times K_y$ PEs ($K_x \times K_y$ is the kernel size) to a single output neuron, computing all synapses at once. This has a couple of disadvantages: Firstly, this arrangement leads to fairly complicated logic (a large MUX mesh) to share data among different neurons. Moreover, if PEs are to be used efficiently, this complexity is compounded by the variability of the kernel size. Therefore, we adopt an efficient alternative: we map each output neuron to a single PE and we time-share each PE across input neurons (that is, synapses) connecting to the same output neuron.

We present the overall NFU structure in Figure 5. The NFU can simultaneously read synapses and input neurons from NBin/NBout and SB, and then distribute them to different PEs. In addition, the NFU contains local storage structures into each PE, and this enables local propagation of input neurons between PEs (see *Inter-PE data propagation* in Section 5.1). After performing computations, the NFU collects results from different PEs and sends them to NBout/NBin or the ALU.

Processing elements. At each cycle, each PE can perform a multiplication and an addition for a convolutional, classifier, or normalization layer, or just an addition for an average

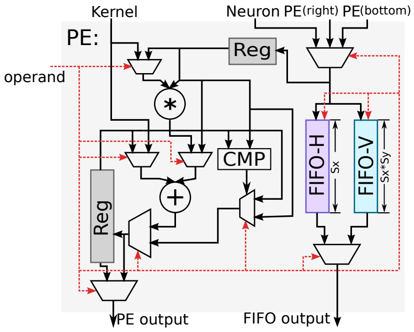

Figure 6: PE architecture.

pooling layer, or a comparison for a max pooling layer, etc. (see Figure 6). $PE_{i,j}$, which is the PE at the i-th row and j-th column of the NFU, has three inputs: one input for receiving the control signals; one input for reading synapses (e.g., kernel values of convolutional layers) from SB; and one input for reading neurons from NBin/NBout, from $PE_{i+1,j}$ (right neighbor), or from $PE_{i+1,j}$ (bottom neighbor), depending on the control signal. The $PE_{i,j}$ has two outputs: one output for writing computation results to NBout/NBin; one output for propagating locally-stored neurons to neighbor PEs (so that they can efficiently reuse the data, see below). In executing a CNN layer, each PE continuously accommodates a single output neuron, and will switch to another output neuron only when the current one has been computed (see Section 8 for detailed neuron-PE mappings).

Inter-PE data propagation. In convolutional, pooling, and normalization layers of CNNs, each output neuron requires data from a rectangular window of input neurons. Such windows are in general significantly overlapping for adjacent output neurons (see Section 8). Although all required data are available from NBin/NBout, repeatedly reading them from the buffer to different PEs requires a high bandwidth. We estimate the internal bandwidth requirement between the on-chip buffers (NBin/NBout and SB) and the NFU (see Figure 7) using a representative convolutional layer (32×32 input feature map and 5×5 convolutional kernel) from LeNet-5 [35] as workload. We observe that, for example, an NFU having only 25 PEs requires >52 GB/s bandwidth. The large bandwidth requirement may lead to large wiring overheads, or significant performance loss (if we limit the wiring overheads).

To support efficient data reuse, we allow inter-PE data propagation on the *PE mesh*, where each PE can send locally-stored input neurons to its left and lower neighbors. We enable this by having two FIFOs (horizontal and vertical: FIFO-H and FIFO-V) in each PE to temporarily store the input values it received. FIFO-H buffers data from NBin/NBout and from the right neighbor PE; such data will be propagated to the left neighbor PE for reuse. FIFO-V buffers the data from NBin/NBout and from the upper neighbor PE; such data will be propagated to the lower neighbor PE for reuse. With inter-PE data propagation, the internal bandwidth requirement can be drastically reduced (see Figure 7).

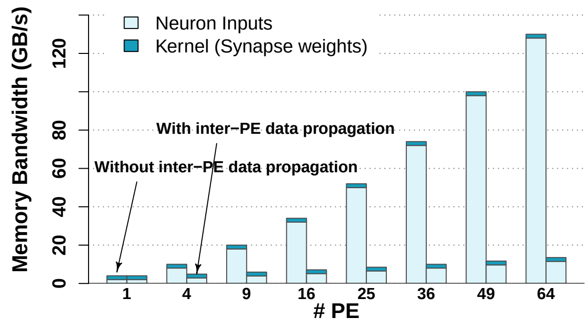

Figure 7: Internal bandwidth from storage structures (input neurons and synapses) to NFU.

5.2. Arithmetic Logic Unit (ALU)

The NFU does not cover all computational primitives in a CNN, thus we need a lightweight ALU to complement the PEs. In the ALU, we implement 16-bit fixed-point arithmetic operators, including division (for average pooling and normalization layers) and non-linear activation functions such as $tanh()$ and $sigmoid()$ (for convolutional and pooling layers). We use a piecewise linear interpolation ($f(x) = a_i x + b_i$, when $x \in [x_i, x_{i+1}]$ and where $i = 0, \ldots, 15$) to compute activation function values; this is known to bring only negligible accuracy loss to CNNs [31, 3]. Segment coefficients a_i and b_i are stored in registers in advance, so that the approximation can be efficiently computed with a multiplier and an adder.

6. Accelerator Architecture: Storage

We use on-chip SRAM to simultaneously store all data (e.g., synapses) and instructions of a CNN. While this seems surprising from both machine learning and architecture perspectives, recent studies have validated the high recognition accuracy of CNNs using a moderate number of parameters. The message is that 4.55 KB–136.11 KB storage is sufficient to simultaneously store all data required for many practical CNNs and, with only around 136 KB of on-chip SRAM, our accelerator can get rid of all off-chip memory accesses and achieve tangible energy-efficiency. In our current design, we implement a 288 KB on-chip SRAM, which is sufficient for all 10 practical CNNs listed in Table 1. The cost of 128 KB SRAM is moderate: 1.65 mm^2 and 0.44 nJ per read in TSMC 65 nm process.

Table 1: CNNs.

CNN	Largest Layer Size (KB)	Synapses Size (KB)	Total Storage (KB)	Accuracy (%)
CNP [46]	15.19	28.17	56.38	97.00
MPCNN [43]	30.63	42.77	88.89	96.77
Face Recogn. [33]	21.33	4.50	30.05	96.20
LeNet-5 [35]	9.19	118.30	136.11	99.05
Simple conv. [53]	2.44	24.17	30.12	99.60
CFF [17]	7.00	1.72	18.49	—
NEO [44]	4.50	3.63	16.03	96.92
ConvNN [9]	45.00	4.35	87.53	96.73
Gabor [30]	2.00	0.82	5.36	87.50
Face align. [11]	15.63	29.27	56.39	—

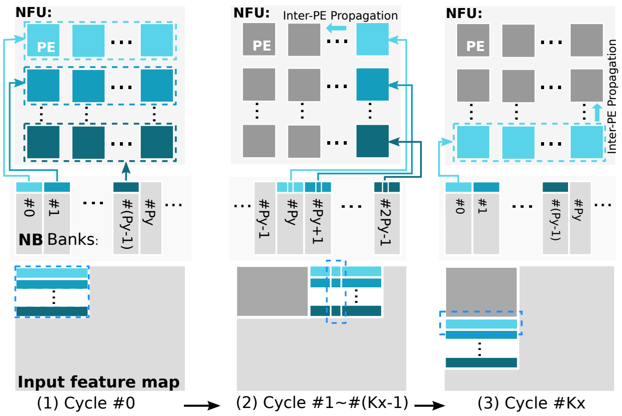

Figure 8: Data stream in the execution of a typical convolutional layer, where we consider the most complex case: the kernel size is larger than the NFU size (#PEs), i.e., $K_x > P_x$ and $K_y > P_y$.

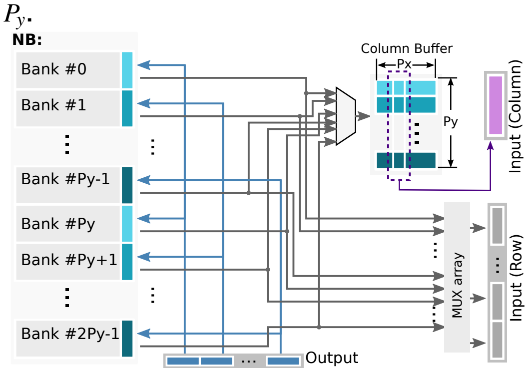

Figure 9: NB controller architecture.

We further split the on-chip SRAM into separate buffers (e.g., NBin, NBout, and SB) for different types of data. This allows us to use suitable read widths for the different types, which minimizes time and energy of each read request. Specifically, NBin and NBout respectively store input and output neurons, and exchange their functionality when all output neurons have been computed and become the input neurons of the next layer. Each of them has $2 \times P_y$ banks, in order to support SRAM-to-PE data movements, as well as inter-PE data propagations (see Figure 8). The width of each bank is $P_x \times 2$ bytes. Both NBin and NBout must be sufficiently large to store all neurons of a whole layer. SB stores all synapses of a CNN and has P_y banks.

7. Accelerator Architecture: Control

7.1. Buffer Controllers

Controllers of on-chip buffers support efficient data reuse and computation in the NFU. We detail the NB controller (used by both NBin and NBout) as an example and omit a detailed description of the other (similar and simpler) buffer controllers for the sake of brevity. The architecture of NB controller is depicted in Figure 9; the controller efficiently supports six read modes and a single write mode.

Without loss of generality, let's assume that NBin stores the input neurons of a layer and NBout is used to store the output neurons of the layer. Recall that NBin has $2 \times P_y$ banks and the width of each bank is $P_x \times 2$ bytes (i.e., P_x 16-bit neurons). Figure 10 illustrates the six read modes of the NB controller:

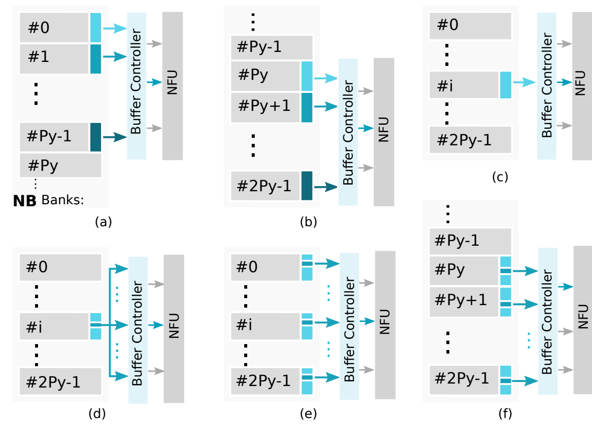

Figure 10: Read modes of NB controller.

(a) Read multiple banks (#0 to $\#P_y - 1$).
(b) Read multiple banks ($\#P_y$ to $\#2P_y - 1$).
(c) Read one bank.
(d) Read a single neuron.
(e) Read neurons with a given step size.
(f) Read a single neuron per bank (#0 to $\#P_y - 1$ or $\#P_y$ to $\#2P_y - 1$).

We select a subset of read modes to efficiently serve each type of layers. For a convolutional layer, we use modes (a) or (b) to read $P_x \times P_y$ neurons from the NBin banks #0 to $\#P_y - 1$ or $\#P_y$ to $\#2P_y - 1$ (Figure 8(1)), mode (e) to deal with the rare (but possible) cases in which the convolutional window is sliding with a step size larger than 1, mode (c) to read P_x neurons from an NB bank (Figure 8(3)), and mode (f) to read P_y neurons from NB banks $\#P_y$ to $\#2P_y - 1$ or #0 to $\#P_y - 1$ (Figure 8(2)). For a pooling layer, we also use modes (a), (b), (c), (e), and (f), since it has similar sliding windows (of input neurons) as a convolutional layer. For a normalization layer, we still use modes (a), (b), (c), (e), and (f) because the layer is usually decomposed into sub-layers behaving similar to convolutional and pooling layers (see Section 8). For a classifier layer, we use mode (d) to load the same input neuron for all output neurons.

The write mode of NB controller is relatively more straightforward. In executing a CNN layer, once a PE has performed all computations of an output neuron, the result will be temporarily stored in a register array of NB controller (see *output* register array in Figure 9). After collecting results from all $P_x \times P_y$ PEs, the NB controller will write them to NBout all at once. In line with the position of each output neuron in the output feature map, the $P_x \times P_y$ output neurons are organized as a data block with P_y rows, each is $P_x \times 2$-bit wide, and corresponds to a single bank of NB. When output neurons in the block lie in the $2kP_x, \ldots, ((2k+1)P_x - 1)$-th columns ($k = 0, 1, \ldots$) of the feature map (i.e., blue columns in the feature map of Figure 11), the data block would be written

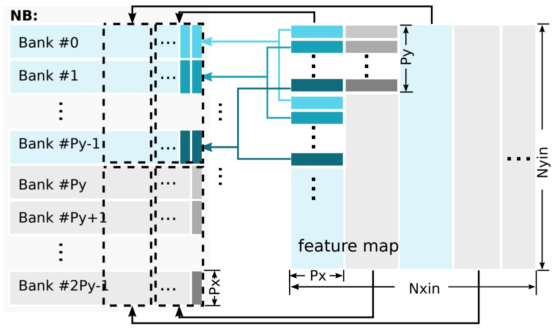

Figure 11: Data organization of NB.

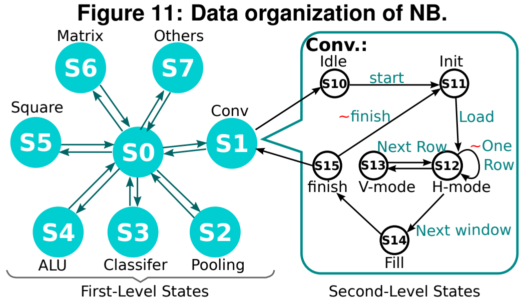

Figure 12: Hierarchical control finite state machine.

to the first P_y banks of NB. Otherwise (when they lie in grey columns in the feature map of Figure 11), the data block will be written to the second P_y banks of NB.

7.2. Control Instructions

We use control instructions to flexibly support CNNs with different settings of layers. An intuitive and straightforward approach would be to put directly cycle-by-cycle control signals for all blocks in instructions (97 bits per cycle). However, a typical CNN might need more than 50K cycles on an accelerator with 64 PEs; this would require an SRAM exceeding 600 KB (97×50K) in order to keep all instructions on-chip, which is inefficient for an embedded sensor.

We choose an efficient alternative, which leverages algorithmic features of CNN layers to provide compact and lossless representations of redundant control signals: We define a two-level Hierarchical Finite State Machine (HFSM) to describe the execution flow of the accelerator (see Figure 12). In the HFSM, *first-level states* describe abstract tasks processed by the accelerator (e.g., different layer types, ALU task). Associated with each first-level state, there are several *second-level states* characterizing the corresponding low-level execution events. For example, the second-level states associated with the first-level state *Conv* (convolutional layer) correspond to execution phases that an input-output feature map pair requires. Due to the limited space, we are not able to provide here all details of the HFSM. In a nutshell, the combination of first- and second-level states (an HFSM state as a whole) is sufficient to characterize the current task (e.g., layer type) processed by the accelerator, as well as the execution flow within a certain number of accelerator cycles. In addition, we can also partially deduce what the accelerator should do in the next few cycles, using the HFSM state and transition rules.

We use a 61-bit instruction to represent each HFSM state and related parameters (e.g., feature map size), which can be decoded into detailed control signals for a certain number of accelerator cycles. Thanks to this scheme, with virtually no loss of flexibility in practice, the aforementioned 50K-cycle CNN only requires a 1 KB instruction storage and a lightweight decoder occupying only 0.03 mm^2 (0.37% the area cost of a 600 KB SRAM) in our 65 nm process.

8. CNN Mapping

In this section, we show how different types of CNN layers are mapped to the accelerator design.

8.1. Convolutional Layer

A convolutional layer constructs multiple output feature maps with multiple input feature maps. When executing a convolutional layer, the accelerator continuously performs the computations of an output feature map, and will not move to the next output feature map until the current map has been constructed. When computing each output feature map, each PE of the accelerator continuously accommodates a single output neuron, and will not switch to another output neuron until the current neuron has been computed.

We present in Figure 13 an example to illustrate how different neurons of the same output feature map are simultaneously computed. Without losing any generality, we consider a small design having 2×2 PEs ($PE_{0,0}$, $PE_{1,0}$, $PE_{0,1}$ and $PE_{1,1}$ in Figure 13), and a convolutional layer with 3×3 kernel size (convolutional window size) and 1×1 step size. For the sake of brevity, we only depict and describe the flow at the first four cycles.

Cycle #0: All four PEs respectively read the first input neurons ($x_{0,0}$, $x_{1,0}$, $x_{0,1}$ and $x_{1,1}$) of their current kernel windows from NBin (with Read Mode (a)), and the same kernel value (synapse) $k_{0,0}$ from SB. Each PE performs a multiplication between the received input neuron and kernel value, and store the result in its local register. In addition, each PE collects its received input neuron in its FIFO-H and FIFO-V for future inter-PE data propagation.

Cycle #1: $PE_{0,0}$ and $PE_{0,1}$ respectively read their required data (input neurons $x_{1,0}$ and $x_{1,1}$) from the FIFO-Hs of $PE_{1,0}$ and $PE_{1,1}$ (i.e., inter-PE data propagation at horizon direction). $PE_{1,0}$ and $PE_{1,1}$ respectively read their required data (input neurons $x_{2,0}$, $x_{2,1}$) from NBin (with Read Mode (f)), and collect them in their FIFO-Hs for future inter-PE data propagation. All PEs share the kernel value $k_{1,0}$ read from SB.

Cycle #2: Similar to Cycle #1, $PE_{0,0}$ and $PE_{0,1}$ respectively read their required data (input neurons $x_{2,0}$ and $x_{2,1}$) from the FIFO-Hs of $PE_{1,0}$ and $PE_{1,1}$ (i.e., inter-PE data propagation at horizon direction). $PE_{1,0}$ and $PE_{1,1}$ respectively read their required data (input neurons $x_{3,0}$ and $x_{3,1}$) from NBin (with Read Mode (f)). All PEs share the kernel value $k_{2,0}$ read from SB. So far each PE has processed the first row

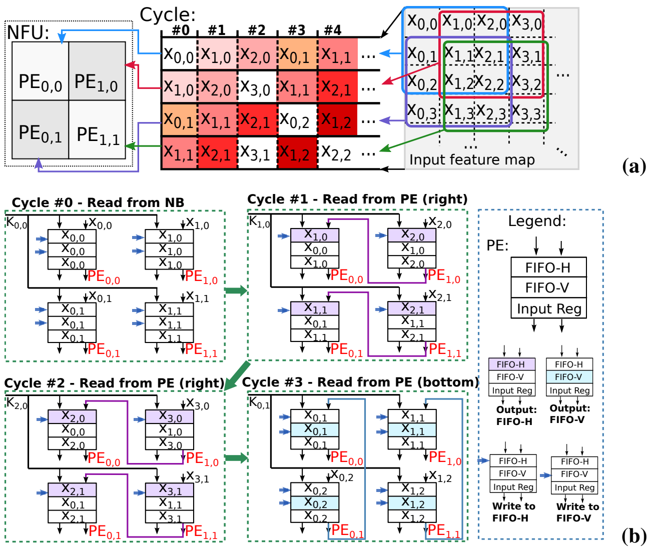

Figure 13: Algorithm-hardware mapping between a convolutional layer (convolutional window: 3×3; step size: 1×1) and an NFU implementation (with 2×2 PEs).

of the corresponding convolutional window, and will move to the second row of the convolutional window at the next cycle.

Cycle #3: $PE_{0,0}$ and $PE_{1,0}$ respectively read their required data (input neurons $x_{0,1}$ and $x_{1,1}$) from FIFO-Vs of $PE_{0,1}$ and $PE_{1,1}$ (i.e., inter-PE data propagation at vertical direction). $PE_{0,1}$ and $PE_{1,1}$ respectively read their required data (input neurons $x_{0,2}$ and $x_{1,2}$) from NBin (with Read Mode (c)). All PEs share the kernel value $k_{0,1}$ read from SB. In addition, each PE collects its received input neuron in its FIFO-H and FIFO-V for future inter-PE data propagation.

In the toy example presented above, inter-PE data propagations reduce by 44.4% the number of reads to NBin (and thus internal bandwidth requirement between NFU and NBin) for computing the four output neurons. In practice, the number of PEs and the kernel size can often be larger, and this benefit will correspondingly become more significant. For example, when executing a typical convolutional layer (C1) of LeNet-5 (kernel size 32×32 and step size 1×1) [35] on a accelerator implementation having 64 PEs, inter-PE data propagations reduces by 73.88% internal bandwidth requirement between NFU and NBin (see also Figure 7).

8.2. Pooling Layer

A pooling layer downsamples input feature maps to construct output feature maps, using *maximum* or *average* operation. Analogous to a convolutional layer, each output neuron of a pooling layer is computed with a window (i.e., pooling window) of neurons in an input feature map. When executing a pooling layer, the accelerator continuously performs the computations of an output feature map, and will not move to the next output feature map until the current map has been constructed. When computing each output feature map, each PE of the accelerator continuously accommodates a single

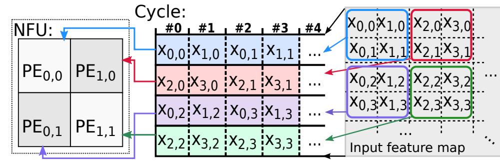

Figure 14: Algorithm-hardware mapping between a pooling layer (pooling window: 2×2; step size: 2×2) and an NFU implementation (with 2×2 PEs).

output neuron, and will not switch to another output neuron until the current neuron has been computed.

In a typical pooling layer, pooling windows of adjacent output neurons are adjacent but non-overlapping, i.e., the step size of window sliding equals to the window size. We present in Figure 14 the execution flow of one such pooling layer, where we consider a small accelerator having 2×2 PEs ($PE_{0,0}$, $PE_{1,0}$, $PE_{0,1}$ and $PE_{1,1}$ in Figure 14), a 2×2 pooling window size, and a 2×2 step size. At each cycle, each PE reads an input neuron (row-first and left-first in the pooling window) from NBin (with Read Mode (e)). PEs do not mutually propagate data because there is no data reuse between PEs.

Yet there are still rare cases in which pooling windows of adjacent neurons are overlapping, i.e., the step size of window sliding is smaller than the window size. Such cases can be treated in a way similar to a convolutional layer, except that there is no synapse in a pooling layer.

8.3. Classifier Layer

In a CNN, convolutional layers allow different input-output neuron pairs to share synaptic weights (i.e., with the kernel), and pooling layers do not have synaptic weights. In contrast, classifier layers are usually fully connected, and there is no sharing of synaptic weights among different input-output neuron pairs. As a result, classifier layers often consume the largest space in the SB (e.g., 97.28% for LeNet-5 [35]). When executing a classifier layer, each PE works on a single output neuron, and will not move to another output neuron until the current one has been computed. While each cycle of a convolutional layer reads a single synaptic weight and $P_x \times P_y$ different input neurons for all $P_x \times P_y$ PEs, each cycle of a classifier layer reads $P_x \times P_y$ different synaptic weights and a single input neuron for all $P_x \times P_y$ PEs. After that, each PE multiplies the synaptic weight and input neuron togther, and

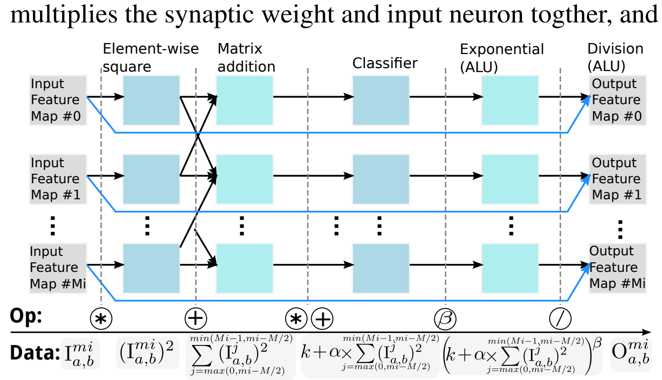

Figure 15: Decomposition of an LRN layer.

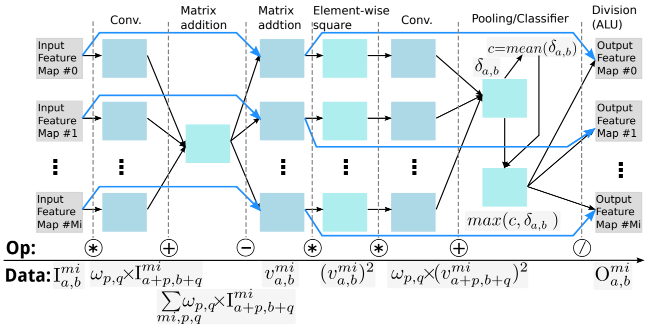

Figure 16: Decomposition of an LCN layer.

Table 2: Benchmarks (C stands for a convolutional layer, S for a pooling layer, and F for a classifier layer).

	Layer	Kernel Size #@*size*	Layer Size #@*size*		Layer	Kernel Size #@*size*	Layer Size #@*size*
CNP [46]	Input		1@42*x*42	MPCNN [43]	Input		1@32*x*32
	C1	6@7*x*7	6@36*x*36		C1	20@5*x*5	20@28*x*28
	S2	6@2*x*2	6@18*x*18		S2	20@2*x*2	20@14*x*14
	C3	61@7*x*7	16@12*x*12		C3	400@5*x*5	20@10*x*10
	S4	16@2*x*2	16@6*x*6		S4	20@2*x*2	20@5*x*5
	C5	305@6*x*6	80@1*x*1		C5	400@3*x*3	20@3*x*3
	F6	160@1*x*1	2@1*x*1		F6	6000@1*x*1	300@1*x*1
					F7	1800@1*x*1	6@1*x*1
	Layer	Kernel Size #@*size*	Layer Size #@*size*		Layer	Kernel Size #@*size*	Layer Size #@*size*
Face Recog. [33]	Input		1@23*x*28	LeNet-5 [35]	Input		1@32*x*32
	C1	20@3*x*3	20@21*x*26		C1	6@5*x*5	6@28*x*28
	S2	20@2*x*2	20@11*x*13		S2	6@2*x*2	6@14*x*14
	C3	125@3*x*3	25@9*x*11		C3	60@5*x*5	16@10*x*10
	S4	25@2*x*2	25@5*x*6		S4	16@2*x*2	16@5*x*5
	F5	1000@1*x*1	40@1*x*1		F5	1920@5*x*5	120@1*x*1
					F6	10080@1*x*1	84@1*x*1
					F7	840@1*x*1	10@1*x*1
	Layer	Kernel Size #@*size*	Layer Size #@*size*		Layer	Kernel Size #@*size*	Layer Size #@*size*
Simple Conv [53]	Input		1@29*x*29	CFF [17]	Input		1@32*x*36
	C1	5@5*x*5	5@13*x*13		C1	4@5*x*5	4@28*x*32
	C2	250@5*x*5	50@5*x*5		S2	4@2*x*2	4@14*x*16
	F3	5000@1*x*1	100@1*x*1		C3	20@3*x*3	14@12*x*14
	F4	1000@1*x*1	10@1*x*1		S4	14@2*x*2	14@6*x*7
					F5	14@6*x*7	14@1*x*1
					F6	14@1*x*1	1@1*x*1
	Layer	Kernel Size #@*size*	Layer Size #@*size*		Layer	Kernel Size #@*size*	Layer Size #@*size*
NEO [44]	Input		1@24*x*24	ConvNN [9]	Input		3@64*x*36
	C1	4@5*x*5	1@24*x*24		C1	12@5*x*5	12@60*x*32
	S2	6@3*x*3	4@12*x*12		S2	12@2*x*2	12@30*x*16
	C3	14@5*x*5	4@12*x*12		C3	60@3*x*3	14@28*x*14
	S4	60@3*x*3	16@6*x*6		S4	14@2*x*2	14@14*x*7
	F5	160@6*x*7	10@1*x*1		F5	14@14*x*7	14@1*x*1
					F6	14@1*x*1	1@1*x*1
	Layer	Kernel Size #@*size*	Layer Size #@*size*		Layer	Kernel Size #@*size*	Layer Size #@*size*
Gabor [30]	Input		1@20*x*20	Face Align. [11]	Input		1@46*x*56
	C1	4@5*x*5	4@16*x*16		C1	4@7*x*7	4@40*x*50
	S2	4@2*x*2	4@8*x*8		S2	4@2*x*2	4@20*x*25
	C3	20@3*x*3	14@6*x*6		C3	6@5*x*5	3@16*x*21
	S4	14@2*x*2	14@3*x*3		S4	3@2*x*2	3@8*x*10
	F5	14@1*x*1	14@1*x*1		F5	180@8*x*10	60@1*x*1
	F6	14@1*x*1	1@1*x*1		F6	240@1*x*1	4@1*x*1

accumulates the result to the partial sum stored at its local register. After a number of cycles, when the dot product (between input neurons and synapses) associated with an output neuron has been computed, the result will be sent to the ALU for the computation of activation function.

8.4. Normalization Layers

Normalization layers can be composed into a number of sub-layers and fundamental computational primitives in order to be executed by our accelerator. We illustrate detailed decompositions in Figures 15 and 16, where an LRN layer is decomposed into a classifier sub-layer, an element-wise square, a matrix addition, exponential functions and divisions; and an LCN layer is decomposed into two convolutional sub-layers, a pooling sub-layer, a classifier sub-layer, two matrix additions, an element-wise square, and divisions. Convolutional, pooling, and classifier sub-layers can be tackled with the rules described in former subsections, and exponential functions and divisions are accommodated by the ALU. The rest computational primitives, including element-wise square and matrix addition, are accommodated by the NFU. In supporting the two primitives, at each cycle, each PE works on an matrix element output with its multiplier or adder, and results of all $P_x \times P_y$ PEs are then written to NBout, following the flow presented in Section 7.1.

9. Experimental Methodology

Measurements. We implemented our design in Verilog, synthesized it with Synopsys Design Compiler, and placed and routed it with Synopsys IC Compiler using the TSMC 65 *nm* Gplus High VT library. We used CACTI 6.0 to estimate the energy cost of DRAM accesses [42]. We compare our design with three baselines:

CPU. The CPU baseline is a 256-bit SIMD (Intel Xeon E7-8830, 2.13 GHz, 1 TB memory). We compile all benchmarks with GCC 4.4.7 with options "-O3 -lm -march=native", enabling the use of SIMD instructions such as MMX, SSE, SSE2, SSE4.1 and SSE4.2.

GPU. The GPU baseline is a modern GPU card (NVIDIA K20M, 5 GB GDDR5, 3.52 TFlops peak in 28 *nm* technology); we use the Caffe library, since it is widely regarded as the fastest CNN library for GPU [1].

Accelerator. To make the comparison fair and adapted to the embedded scenario, we resized our previous work, i.e., DianNao [3] to have a comparable amount of arithmetic operators as our design—i.e., we implemented an 8×8 DianNao-NFU (8 hardware neurons, each processes 8 input neurons and 8 synapses per cycle) with a 62.5 GB/s bandwidth memory model instead of the original 16×16 DianNao-NFU with 250 GB/s bandwidth memory model (unrealistic in a vision sensor). We correspondingly shrank the sizes of on-chip buffers by half in our re-implementation of DianNao: 1 KB NBin/NBout and 16 KB SB. We have verified that our implementation is roughly fitting to the original design. For instance, we obtained an area of 1.38 mm^2 for our re-implementation versus 3.02 mm^2 for the original DianNao [3], which tracks well the ratio in computing and storage resources.

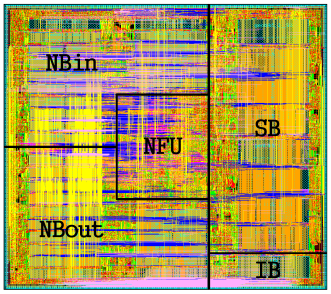

Figure 17: Layout of ShiDianNao (65 nm).

Benchmarks. We collected 10 CNNs from representative visual recognition applications and used them as our benchmarks (Table 2). Among all layers of all benchmarks, input neurons consume at most 45 KB, and synapses consume at most 118 KB, which do not exceed the SRAM capacities of our design (Table 3).

10. Experimental Results

10.1. Layout Characteristics

We present in Tables 3 and 4 the parameters and layout characteristics of the current ShiDianNao version (see Figure 17), respectively. ShiDianNao has 8×8 (64) PEs and a 64 KB NBin, a 64 KB NBout, a 128 KB SB, and a 32 KB IB. The overall SRAM capacity of ShiDianNao is 288 KB (11.1$\times$ larger than that of DianNao), in order to simultaneously store all data and instructions for a practical CNN. Yet, the total area of ShiDianNao is only 3.52$\times$ larger than that of DianNao (4.86 mm^2 vs. 1.38 mm^2).

10.2. Performance

We compare ShiDianNao against the CPU, the GPU, and DianNao on all benchmarks listed in Section 9. The results are shown in Figure 18. Unsurprisingly, ShiDianNao significantly outperforms the general purpose architectures and is, on average, 46.38 $\times$ faster than the CPU and 28.94$\times$ faster than the GPU. In particular, the GPU cannot take full advantage of its high computational power because the small computational kernels of the visual recognition tasks listed in Table 1 map poorly on its 2,496 hardware threads.

More interestingly, ShiDianNao also outperforms our accelerator baseline on 9 out of 10 benchmarks (1.87$\times$ faster on average on all 10 benchmarks). There are two main reasons for that: Firstly, compared to DianNao, ShiDianNao eliminates off-chip memory accesses during execution, thanks to a sufficiently large SRAM capacity and a correspondingly slightly higher cost. Secondly, ShiDianNao efficiently exploits the locality of 2D feature maps with its dedicated SRAM controllers and its inter-PE data reuse mechanism; DianNao, on the other hand, cannot make good use of that locality.

ShiDianNao performs slightly worse than the accelerator baseline on benchmark *Simple Conv*. The issue is that ShiDianNao works on a single output feature map at a time and each PE works on a single output neuron of the feature map.

Table 3: Parameter settings of ShiDianNao and DianNao.

	ShiDianNao	DianNao
Data width	16-bit	16-bit
# multipliers	64	64
NBin SRAM size	64 KB	1 KB
NBout SRAM size	64 KB	1 KB
SB SRAM size	128 KB	16 KB
Inst. SRAM size	32 KB	8 KB

Table 4: Hardware characteristics of ShiDianNao at 1GHz, where power and energy are averaged over 10 benchmarks.

Accelerator	Area (mm^2)	Power (mW)	Energy (nJ)
Total	4.86 (100%)	320.10 (100%)	6048.70 (100%)
NFU	0.66 (13.58%)	268.82 (83.98%)	5281.09 (87.29%)
NBin	1.12 (23.05%)	35.53 (11.10%)	475.01 (7.85%)
NBout	1.12 (23.05%)	6.60 (2.06%)	86.61 (1.43%)
SB	1.65 (33.95%)	6.77 (2.11%)	94.08 (1.56%)
IB	0.31 (6.38%)	2.38 (0.74%)	35.84 (0.59%)

Therefore, when most of an application consists of uncommonly small output feature maps with fewer output neurons than implemented PEs (e.g., 5×5 in the C2 layer of benchmark *Simple Conv* for 8×8 PEs in the current accelerator design), some PEs will be idle. Although we played with the idea of alleviating this issue by adding complicated control logic to each PE and allowing different PEs to simultaneously work on different feature maps, we ultimately decided against this option as it appeared a poor trade-off with a detrimental impact on the programming model.

Concerning the ability of ShiDianNao to process in real time a stream of frames from a sensor, the longest time to process a 640x480 video frame is for benchmark *ConvNN* which requires 0.047 ms to process a 64×36-pixel region. Since each frame contains $\lceil (640-64)/16+1 \rceil \times \lceil (480-36)/16+1 \rceil = 1073$ such regions (overlapped by 16 pixels), a frame takes a little more than 50 ms to process, resulting in a speed of 20 frames per second for the most demanding benchmark. Since typical commercial sensors can stream data at a desired rate and since streaming speed can thus be matched to the processing rate, the partial frame buffer must store only the parts of the image reused across overlapping regions. This is of the order of a few tens of pixel rows and fits well the 256 KB of commercial image processors. Although apparently low, the 640$\times$480 resolution is in line with the fact that usually images are resized in certain range before processing [47, 34, 23, 16].

10.3. Energy

In Figure 19, we report the energy consumed by GPU, DianNao and ShiDianNao, inclusive of main memory accesses to obtain the input data. Even if ShiDianNao is not meant to access DRAM, we have conservatively included main memory accesses for the sake of a fair comparison. ShiDianNao is on average 4688.13$\times$ and 63.48$\times$ more energy efficient than GPU and DianNao, respectively. We also evaluate an ideal version of DianNao (DianNao-FreeMem, see Figure 19), where we assume that main memory accesses incur no energy cost. Interestingly, we observe that ShiDianNao is still

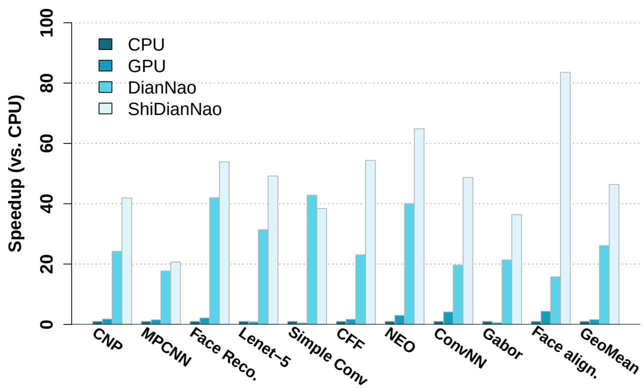

Figure 18: Speedup of GPU, DianNao, and ShiDianNao over the CPU.

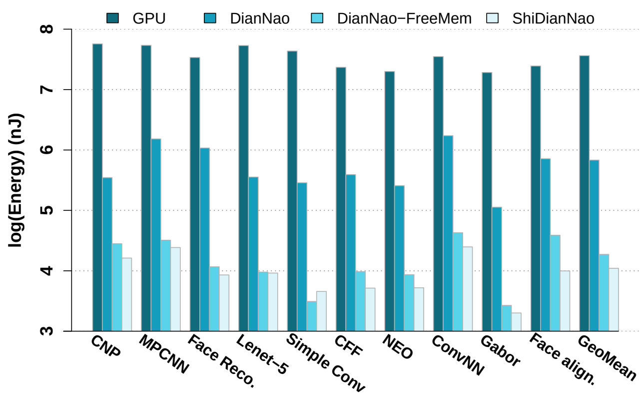

Figure 19: Energy cost of GPU, DianNao, and ShiDianNao.

1.66× more energy efficient than DianNao-FreeMem. Moreover, when ShiDianNao is integrated in an embedded vision sensor and frames are stored directly into its NBin, the superiority is even more significant: in this setting, ShiDianNao is 87.39× and 2.37× more energy efficient than DianNao and DianNao-FreeMem, respectively.

We illustrate in Table 4 the breakdown of the energy consumed by our design. We observe that four SRAM buffers account for only 11.43% the overall energy, and the rest is consumed by the logic (87.29%). This is significantly different from Chen et al.'s observation made on DianNao [3], where more than 95% of the energy is consumed by the DRAM.

11. Related Work

Visual sensor and processing. Due to the rapid development of integrated circuits and sensor technologies, the size and cost of a vision sensor quickly scales down, which offers a great opportunity to integrate higher-resolution sensors in mobile ends and wearable devices (e.g., Google Glass [54] and Samsung Gear [49]). Under emerging application scenarios such as image recognition/search [19, 54], however, end devices do not locally perform intensive visual processing on images captured by sensors, due to the limited computational capacity and power budget. Instead, computation-intensive visual processing algorithms like CNNs [27, 17, 33, 30] are performed at the sever end, leading to considerable workloads to the server, which greatly limits the QoS and, ultimately, the growth of end users. Our study partially bridges this gap by shifting visual processing closer to sensors.

Neural network accelerators. Neural networks were conventionally executed on CPUs [59, 2], and GPUs [15, 51, 5]. These platforms can flexibly adapt to various workloads, but the flexibility is achieved at a large fraction of transistors, significantly affecting the energy-efficiency of executing specific workloads such as CNNs (see Section 3). After a first wave of designs at the end of the last century [25], there have also been a few more modern application-specific accelerator architectures for various neural networks, with implementations on either FPGAs [50, 46, 52] or ASICs [3, 16, 57]. For CNNs, Farabet et al. proposed a systolic architecture called NeuFlow architecture [16], Chakradhar et al. designed a systolic-like coprocessor [2]. Although effective to handle 2D convolution in signal processing [37, 58, 38, 22], systolic architectures do not provide sufficient flexibility and efficiency to support different settings of CNNs [8, 50, 16, 2], which is exemplified by their strict restrictions on CNN parameters (e.g., size of convolutional window, step size of window sliding, etc), as well as their high memory bandwidth requirements. There have been some neural network accelerators adopting SIMD-like architectures. Esmaeilzadeh et al. proposed a neural network stream processing core (NnSP) with an array of PEs, but the released version is still designed for Multi-Layer Perceptrons (MLPs) [12]. Peemen et al. [45] proposed to accelerate CNNs with an FPGA accelerator controlled by a host processor. Although this accelerator is equipped with a memory subsystem customized for CNNs, the requirement of a host processor limits the overall energy efficiency. Gokhale et al. [18] designed a mobile coprocessor for visual processing at mobile devices, which supports both CNNs and DNNs. The above studies did not treat main memory accesses as the first-order concern, or directly linked the computational block to the main memory via a DMA. Recently, some of us [3] designed dedicated on-chip SRAM buffers to reduce main memory accesses, and the proposed DianNao accelerator cover a broad range of neural networks including CNNs. However, in order to flexibly support different neural networks, DianNao does not implement specialized hardware to exploit data locality of 2D feature maps in a CNN, but instead treats them as 1D data vectors in common MLPs. Therefore, DianNao still needs frequent memory accesses to execute a CNN, which is less energy efficient than our design (see Section 10 for experimental comparisons). Recent members of the DianNao family [4, 40] have been optimized for large-scale neural networks and classic machine learning techniques respectively. However, they are not designed for embedded applications, and their architectures are significantly different from the ShiDianNao architecture.

Our design is substantially different from previous studies in two aspects. First, unlike previous designs requiring memory accesses to get data, our design does not access to the main memory when executing a CNN. Second, unlike previous systolic designs supporting a single CNN with a fixed parameter and layer setting [2, 8, 16, 50], or a single convolutional layer with fixed parameters, our design flexibly accommodates different CNNs with different parameter and layer settings. Due to these attractive features, our design is more energy-efficient than previous designs on CNNs, thus particularly suits visual recognition in embedded systems.

12. Conclusions

We designed a versatile accelerator for state-of-the-art visual recognition algorithms. Averaged on 10 representative benchmarks, our design is, respectively, about 50×, 30×, and 1.87× faster than a mainstream CPU, a GPU, and our own reimplementation of the DianNao neural network accelerator [3]. ShiDianNao consumes only about 4700x and 60x less energy than the GPU and DianNao, respectively. Our design has an area of 4.86 mm^2 in a 65 nm process and consumes only 320.10 mW at 1 GHz. Thanks to its high performance, its low power consumption, as well as its small area, ShiDianNao particularly suits visual applications at mobile ends and wearable devices. Our accelerator is suitable for integration in such devices, on the streaming path from sensors to hosts. This would significantly reduce workloads at servers, greatly enhance the QoS of emerging visual applications, and eventually contribute to the ubiquitous success of visual processing.

Acknowledgments

This work is partially supported by the NSF of China (under Grants 61100163, 61133004, 61222204, 61221062, 61303158, 61432016, 61472396, 61473275, 60921002), the 973 Program of China (under Grant 2015CB358800, 2011CB302504), the Strategic Priority Research Program of the CAS (under Grants XDA06010403, XDB02040009), the International Collaboration Key Program of the CAS (under Grant 171111KYSB20130002), and the 10000 talent program.

References

[1] Berkeley Vision and Learning Center, “Caffe: a deep learning framework.” Available: http://caffe.berkeleyvision.org/

[2] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, “A dynamically configurable coprocessor for convolutional neural networks,” in *Proceedings of the 37th annual international symposium on Computer architecture (ISCA)*. New York, USA: ACM Press, 2010, pp. 247–257.

[3] T. Chen, Z. Du, N. Sun, J. Wang, and C. Wu, “DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning,” in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Salt Lake City, UT, USA, 2014, pp. 269–284.

[4] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, “DaDianNao: A Machine-Learning Supercomputer,” in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015, pp. 609–622.

[5] D. C. Cires, U. Meier, J. Masci, and L. M. Gambardella, “Flexible , High Performance Convolutional Neural Networks for Image Classification,” in *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence (IJCNN)*, 2003, pp. 1237–1242.

[6] A. Coates, B. Huval, T. Wang, D. J. Wu, and A. Y. Ng, “Deep learning with COTS HPC systems,” in *Proceedings of the 30th International Conference on Machine Learning (ICML)*, 2013, pp. 1337–1345.

[7] G. Dahl, T. Sainath, and G. Hinton, “Improving deep neural networks for LVCSR using rectified linear units and dropout,” in *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP’13)*, 2013, pp. 8609–8613.

[8] S. A. Dawwd, “The multi 2D systolic design and implementation of Convolutional Neural Networks,” in *2013 IEEE 20th International Conference on Electronics, Circuits, and Systems (ICECS)*. IEEE, Dec. 2013, pp. 221–224.

[9] M. Delakis and C. Garcia, “Text Detection with Convolutional Neural Networks,” in *International Conference on Computer Vision Theory and Applications (VISAPP)*, 2008, pp. 290–294.

[10] Z. Du, A. Lingamneni, Y. Chen, K. Palem, O. Temam, and C. Wu, “Leveraging the error resilience of machine-learning applications for designing highly energy efficient accelerators,” *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 201–206, Jan. 2014.

[11] S. Duffner and C. Garcia, “Robust Face Alignment Using Convolutional Neural Networks,” in *International Conference on Computer Vision Theory and Applications (VISAPP)*, 2008, pp. 30–37.

[12] H. Esmaeilzadeh, P. Saeedi, B. Araabi, C. Lucas, and S. Fakhraie, “Neural Network Stream Processing Core (NnSP) for Embedded Systems,” *2006 IEEE International Symposium on Circuits and Systems (ISCS)*, pp. 2773–2776, 2006.

[13] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, “Neural Acceleration for General-Purpose Approximate Programs,” *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 449–460, Dec. 2012.

[14] K. Fan, S. Mahlke, and A. Arbor, “Bridging the Computation Gap Between Programmable Processors and Hardwired Accelerators,” in *IEEE 15th International Symposium on High Performance Computer Architecture (HPCA)*, 2009, pp. 313–322.

[15] C. Farabet, B. Martini, P. Akselrod, S. Talay, Y. LeCun, and E. Culurciello, “Hardware accelerated convolutional neural networks for synthetic vision systems,” in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems (ISCS)*. IEEE, May 2010, pp. 257–260.

[16] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, “NeuFlow: A runtime reconfigurable dataflow processor for vision,” in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. IEEE, Jun. 2011, pp. 109–116.

[17] C. Garcia and M. Delakis, “Convolutional face finder: a neural architecture for fast and robust face detection.” *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, vol. 26, no. 11, pp. 1408–23, Nov. 2004.

[18] V. Gokhale, J. Jin, and A. Dundar, “A 240 G-ops/s mobile coprocessor for deep neural networks,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2014, pp. 682–687.

[19] Google, “Google image search.” Available: http://www.google.com/insidesearch/features/images/searchbyimage.htm

[20] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, “Understanding sources of inefficiency in general-purpose chips,” in *Proceedings of Annual International Symposium on Computer Architecture (ISCA)*. New York, USA: ACM Press, 2010, p. 37.

[21] S. Haykin, *Neural Networks: A Comprehensive Foundation*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1998.

[22] V. Hecht, K. Ronner, and P. Pirsch, “An Advanced Programmable 2D-Convolution Chip for Real Time Image Processing,” in *IEEE International Sympoisum on Circuits and Systems (ISCS)*, 1991, pp. 1897 – 1900.

[23] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, “Improving neural networks by preventing co-adaptation of feature detectors,” in *arXiv: 1207.0580 (2012)*, pp. 1–18.

[24] P. S. Huang, X. He, J. Gao, and L. Deng, “Learning deep structured semantic models for web search using clickthrough data,” in *International Conference on Information and Knowledge Management (CIKM)*, 2013, pp. 2333–2338.

[25] P. Ienne, T. Cornu, and G. Kuhn, “Special-purpose digital hardware for neural networks: An architectural survey,” *Journal of VLSI Signal Processing*, vol. 13, no. 1, pp. 5–25, 1996.

[26] K. Jarrett, K. Kavukcuoglu, M. A. Ranzato, and Y. LeCun, "What is the best multi-stage architecture for object recognition?" *2009 IEEE 12th International Conference on Computer Vision (ICCV)*, pp. 2146–2153, Sep. 2009.

[27] S. Kamijo, Y. Matsushita, K. Ikeuchi, and M. Sakauchi, "Traffic monitoring and accident detection at intersections," *IEEE Transactions on Intelligent Transportation Systems*, vol. 1, no. 2, pp. 108–118, Jun. 2000.

[28] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the future of parallel computing," *IEEE Micro*, pp. 7–17, 2011.

[29] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," *Advances In Neural Information Processing Systems*, pp. 1–9, 2012.

[30] B. Kwolek, "Face detection using convolutional neural networks and Gabor filters," *Artificial Neural Networks: Biological Inspirations (ICANN)*, pp. 551–556, 2005.

[31] D. Larkin, A. Kinane, V. Muresan, and N. E. O'Connor, "An Efficient Hardware Architecture for a Neural Network Activation Function Generator," in *Advances in Neural Networks*, ser. Lecture Notes in Computer Science, vol. 3973. Springer, 2006, pp. 1319–1327.

[32] H. Larochelle, D. Erhan, A. Courville, J. Bergstra, and Y. Bengio, "An empirical evaluation of deep architectures on problems with many factors of variation," in *International Conference on Machine Learning (ICML)*. New York, New York, USA: ACM Press, 2007, pp. 473–480.

[33] S. Lawrence, C. L. Giles, a. C. Tsoi, and a. D. Back, "Face recognition: a convolutional neural-network approach." *IEEE Transactions on Neural Networks*, vol. 8, no. 1, pp. 98–113, Jan. 1997.

[34] Q. V. Le, M. A. Ranzato, M. Devin, G. S. Corrado, and A. Y. Ng, "Building High-level Features Using Large Scale Unsupervised Learning," *International Conference on Machine Learning (ICML)*, pp. 8595 – 8598, 2012.

[35] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-Based Learning Applied to Document Recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278– 2324, 1998.

[36] Y. LeCun, K. Kavukcuoglu, and C. Farabet, "Convolutional networks and applications in vision," *Proceedings of 2010 IEEE International Symposium on Circuits and Systems (ISCS)*, pp. 253–256, May 2010.

[37] J.-J. Lee and G.-Y. Song, "Super-Systolic Array for 2D Convolution," *2006 IEEE Region 10 Conference (TENCON)*, pp. 1–4, 2006.

[38] S. Y. Lee and J. K. Aggarwal, "Parallel 2D convolution on a Mesh Connected Array Processor," *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, vol. PAMI-9, no. 4, pp. 590–594, 1987.

[39] B. Liang and P. Dubey, "Recognition, Mining and Synthesis," *Intel Technology Journal*, vol. 09, no. 02, 2005.

[40] D. Liu, T. Chen, S. Liu, J. Zhou, S. Zhou, O. Temam, X. Feng, X. Zhou, and Y. Chen, "Pudiannao: A polyvalent machine learning accelerator," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015, pp. 369–381.

[41] V. Mnih and G. Hinton, "Learning to Label Aerial Images from Noisy Data," in *Proceedings of the 29th International Conference on Machine Learning (ICML)*, 2012, pp. 567–574.

[42] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0," *The 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 3–14, Dec. 2007.

[43] J. Nagi, F. Ducatelle, G. A. D. Caro, D. Cires, U. Meier, A. Giusti, and L. M. Gambardella, "Max-Pooling Convolutional Neural Networks for Vision-based Hand Gesture Recognition," in *IEEE International Conference on Signal and Image Processing Applications (ICSIPA)*, 2011, pp. 342–347.

[44] C. Nebauer, "Evaluation of convolutional neural networks for visual recognition." *IEEE Transactions on Neural Networks*, vol. 9, no. 4, pp. 685–96, Jan. 1998.

[45] M. Peemen, A. a. a. Setio, B. Mesman, and H. Corporaal, "Memory-centric accelerator design for Convolutional Neural Networks," in *International Conference on Computer Design (ICCD)*. IEEE, Oct. 2013, pp. 13–19.

[46] C. Poulet, J. Y. Han, and Y. Lecun, "CNP: An FPGA-based processor for Convolutional Networks," in *International Conference on Field Programmable Logic and Applications (FPL)*, vol. 1, no. 1, 2009, pp. 32–37.

[47] M. Ranzato, F. J. Huang, Y. L. Boureau, and Y. LeCun, "Unsupervised learning of invariant feature hierarchies with applications to object recognition," in *Computer Vision and Pattern Recognition (CVPR)*. IEEE, Jun. 2007, pp. 1–8.

[48] R. Salakhutdinov and G. Hinton, "Learning a nonlinear embedding by preserving class neighbourhood structure," in *AI and Statistics*, ser. JMLR Workshop and Conference Proceedings, vol. 3, no. 5. Citeseer, 2007, pp. 412–419.

[49] SAMSUNG, "SAMSUNG Gear2 Tech Specs," Samsung Electronics, 2014.

[50] M. Sankaradas, V. Jakkula, S. Cadambi, S. Chakradhar, I. Durdanovic, E. Cosatto, and H. P. Graf, "A Massively Parallel Coprocessor for Convolutional Neural Networks," *2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 53–60, Jul. 2009.

[51] D. Scherer, H. Schulz, and S. Behnke, "Accelerating large-scale convolutional neural networks with parallel graphics multiprocessors," in *Artificial Neural Networks (ICANN)*, ser. Lecture Notes in Computer Science, K. Diamantaras, W. Duch, and L. Iliadis, Eds. Springer Berlin Heidelberg, 2010, vol. 6354, pp. 82–91.

[52] P. Sermanet and Y. LeCun, "Traffic sign recognition with multi-scale Convolutional Networks," in *International Joint Conference on Neural Networks (IJCNN)*. Ieee, Jul. 2011, pp. 2809–2813.

[53] P. Y. Simard, D. Steinkraus, and J. C. Platt, "Best practices for convolutional neural networks applied to visual document analysis," in *Seventh International Conference on Document Analysis and Recognition 2003 Proceedings (ICDAR)*, vol. 1. IEEE Comput. Soc, 2003, pp. 958–963.

[54] T. Starner, "Project Glass: An Extension of the Self," *IEEE Pervasive Computing*, vol. 12, no. 2, pp. 14–16, Apr. 2013.

[55] *STV0986, 5 Megapixel mobile imaging processor (Data Brief)*, STMicroelectronics, Jan. 2007.

[56] *STV0987, 8 Megapixel mobile imaging processor (Data Brief)*, STMicroelectronics, Mar. 2013.

[57] O. Temam, "A defect-tolerant accelerator for emerging high-performance applications," *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, vol. 00, no. c, pp. 356–367, 2012.

[58] H. Tkung and D. Wl, "Two-level pipelined systolic array for multidimensional convolution," *Image and Vision Computing*, vol. 1, no. 1, pp. 30–36, 1983.

[59] V. Vanhoucke, A. Senior, and M. Z. Mao, "Improving the speed of neural networks on CPUs," in *Deep Learning and Unsupervised Feature Learning Workshop, Neural Information Processing Systems Conference (NIPS)*, 2011.

[60] G. Venkatesh, J. Sampson, N. Goulding-hotta, S. K. Venkata, M. B. Taylor, and S. Swanson, "QSCORES : Trading Dark Silicon for Scalable Energy Efficiency with Quasi-Specific Cores Categories and Subject Descriptors," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011, pp. 163–174.

[61] S. Yehia, S. Girbal, H. Berry, and O. Temam, "Reconciling specialization and flexibility through compound circuits," in *IEEE 15th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2009, pp. 277–288.