An Accelerator for High Efficient Vision Processing

Zidong Du, Shaoli Liu, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Qi Guo, Xiaobing Feng, Yunji Chen, and Olivier Temam

***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 (CNNs), 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 highly energy-efficient. We present a single-core implementation down to the layout at 65 nm, with a modest footprint of 5.94 mm^2 and consuming only 336 mW, but still about 30× faster than high-end GPUs. For visual processing with higher resolution and frame-rate requirements, we further present a multicore implementation with elevated performance.**

***Index Terms*—Accelerator architectures, convolutional neural network, vision sensor.**

I. Introduction

In the past few years, accelerators have gained increasing attention as an energy and cost effective alternative to CPUs and GPUs [1]–[6]. 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 toward recognition and mining applications [7] and the fact that a very small corpus of algorithms—i.e., neural network-based algorithms—can tackle a significant share of these applications [8]–[10]. This makes it possible to realize the best of both worlds: accelerators with high performance/efficiency and yet broad application scope. Chen *et al.* [11] 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.

This paper aimed at supporting the two main state-of-the-art neural networks: 1) convolutional neural networks (CNNs) [12] and 2) deep neural networks (DNNs) [13], [14]. 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) 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 [15] versus up to 1 billion [16] or even 10 billions [17] 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 [12], 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 [1], [18]. 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 recognition applications (followed by voice recognition). In many

Manuscript received January 25, 2016; revised May 4, 2016; accepted June 13, 2016. Date of publication June 22, 2016; date of current version January 19, 2017. This work is partially supported by the NSF of China (under Grants 61133004, 61303158, 61432016, 61472396, 61473275, 61522211, 61532016, 61521092), the 973 Program of China (under Grant 2015CB358800), 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. This paper was recommended by Associate Editor S. Hu.

Z. Du and T. Luo are with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China, and also with the University of Chinese Academy of Sciences, Beijing 100049, China (e-mail: duzidong@ict.ac.cn; luotao@ict.ac.cn).

S. Liu, T. Chen, L. Li, Q. Guo, and X. Feng are with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China (e-mail: liushaoli@ict.ac.cn; chentianshi@ict.ac.cn; liling@ict.ac.cn; guoqi@ict.ac.cn; fxb@ict.ac.cn).

R. Fasthuber and P. Ienne are with EPFL, Lausanne CH-1015, Switzerland (e-mail: robert.fasthuber@epfl.ch; paolo.ienne@epfl.ch).

Y. Chen is with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China, and also with the CAS Center for Excellence in Brain Science, Chinese Academy of Sciences, Beijing 100190, China (e-mail: cyj@ict.ac.cn).

O. Temam is with INRIA, Palaiseau 91120, France (e-mail: olivier.temam@inria.fr).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2016.2584062

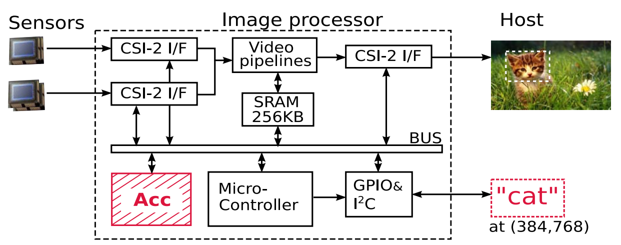


Fig. 1. Possible integration of our accelerator in a commercial image processing chip.

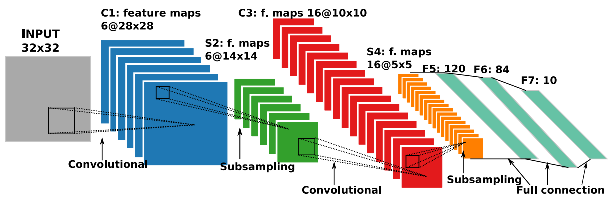


Fig. 2. Representative CNN architecture—LeNet5 [12]. C: convolutional layer; S: pooling layer; F: classifier layer.

real-world and embedded applications—e.g., smartphones, security, and 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 present an energy-efficient 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 54× more energy efficient than DianNao [11], 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 (PEs), from the sensor, and the SRAM holding the CNN model. We present a single-core implementation of the accelerator down to the layout in a 65 nm CMOS technology, and extend it to a multicore implementation to achieve higher performance. The single-core version is with a peak performance of 194 GOP/s (billions of fixed-point OPerations per second) at 5.94 mm^2, 336.51 mW, and 1 GHz, capable of processing 640×480 images at a frame rate of 11. The extended multicore version further achieves a peak performance of 896 GOP/s at 8.50 mm^2, 1440 mW, and 1 GHz with 4 cores, capable of processing 1280×720 images at a frame rate of 56. 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.

II. System Integration

Fig. 1 shows a typical integration solution for cheap cameras (closely resembling an STM chipset [19], [20]). An image processing chip is connected to cameras (in typical smartphones, two) streaming their data through standard camera serial interfaces. More advanced processors already implement rudimentary object detection and tracking functions, such as face recognition [20]. 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 [19]). 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.

III. CNN and Mapping Principles

A. Primer on Convolutional Neural Networks

1) CNN: CNNs [12] and DNNs [13] are known as two state-of-the-art machine learning algorithms. Both of them belong to the family of multilayer perceptrons (MLPs) [21] and may consist of four types of layers: 1) convolutional; 2) pooling; 3) normalization; and 4) 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 [11]. 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. Fig. 2 illustrates the architecture of LeNet-5 [12], a representative CNN widely used in document recognition. It consists of two convolutional layers (C1 and C3 in Fig. 2), two pooling layers (S2 and S4 in Fig. 2), and three classifier layers (F5, F6, and F7 in Fig. 2). Recent studies also suggest the use of normalization layers in deep learning (local response normalization [LRN] [15] and local contrast normalization [LCN] [22]). In this example, it is possible to store the synaptic weights of C1/C3 layer in a 0.29/2.93 KB size SRAM, as the total number of the synaptic weights is only 118 KB.

2) Recognition Versus 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 [11], [23]. Offline 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.

B. Mapping Principles

Roughly, a purely spatial hardware implementation of a neural network would devote a separate accumulation unit for each

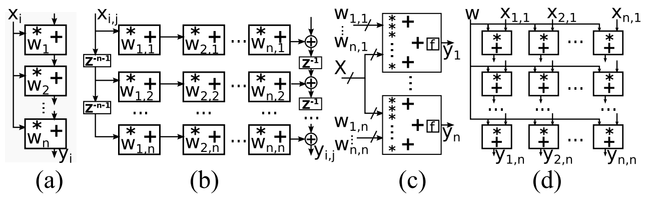


Fig. 3. Typical implementations of neural networks. (a) 1-D systolic. (b) 2-D systolic. (c) Spatial neurons. (d) ShiDianNao.

neuron and a separate multiplier for each synapse. From the early days of neural networks in the 80s and 90s, architects have imagined that concrete applications would contain too 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 [2], 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 [24] for an early taxonomy): products, prototypes, and paper designs have probably exhausted all possibilities, including attributing each neuron to a PE, each synapes to a PE, and flowing in a systolic fashion both kernels and input feature maps. Some of these principal choices are represented in Fig. 3. In this paper, we have naturally decided to rely on: 1) the 2-D nature of our processed data (images) and 2) the limited size of the convolutional kernels.

Fig. 3 presents four potential mapping strategies. The first strategy in Fig. 3(a) is 1-D systolic mapping, which fails to address the 2-D image data efficiently, as it cannot exploit all possible data reuse in 2-D convolutions. The second strategy in Fig. 3(b) is 2-D systolic mapping. Although this strategy appeared to fit 2-D images naturally, it cannot flexibly support various sizes of convolution kernels and other CNN layers such as pooling and LRN. The third strategy in Fig. 3(c) had been investigated by several recent works, where neurons are mapped to the hardware implementation in a spatially expanded manner [2] or a folded manner [11] (e.g., DianNao, neurons are computed through multiple cycles by reusing the hardware). This strategy orchestrates all computations as vector operations that can be efficiently processed by multiple vector processing units. However, it does not fully take the CNN nature, e.g., various kernel/input sizes, data reuse, and sophisticated input data requirement, into account, leading to remarkable waste of hardware resources for some CNNs. As an illustrative example, we consider the mapping strategy of the convolution layer of a popular neural network accelerator, DianNao, which contains 16 multiplication-addition vector processing units. Each unit focuses on its output neuron which has same location with other units but on different output feature maps, using shared neuron inputs and independent synaptic weights; the input neurons are from different input feature maps with same location, see Fig. 4(a) (M_o output feature maps which are in size of $N_{\text{xout}} \times N_{\text{yout}}$ are obtained by convolving M_i $N_{\text{xin}} \times N_{\text{yin}}$ input feature maps with $k_x \times k_y$ convolutional kernels). However, once the number of feature maps does not exactly match the number and vector size of the processing units, plenty of multipliers and adders in such processing units will be wasted. An extreme case is that the convolution layer contains only one input and one output feature map [see Fig. 4(b)]. In this case, only one processing

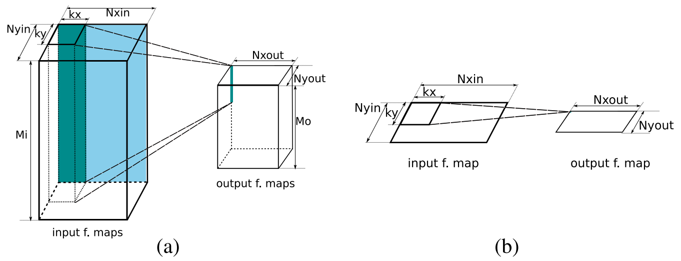


Fig. 4. Convolutional layer mapping in DianNao [11]. (a) Typical convolutional layer. (b) Convolutional layer with only one input and output feature map.

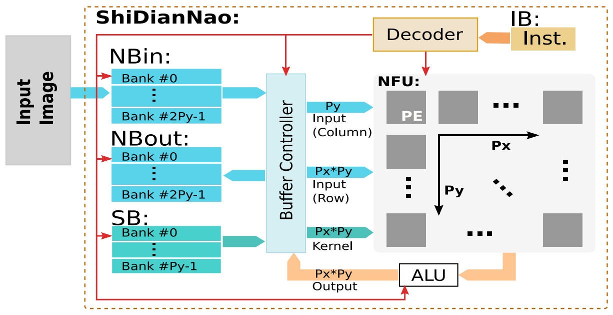


Fig. 5. ShiDianNao architecture.

unit in DianNao works on effective data while the rest ones are idle, thus the real performance is only $1/16$ of the peak performance. Although DianNao has already optimized the mapping strategy of the convolution layer, the inability to efficiently handle various kernel sizes remains. Also, without considering the 2-D nature of the convolution layer and the reuse of shared kernels, DianNao needs to repeatedly load different kernels and intermediated results into the processing units from the on-chip storage, leading to considerable SRAM power consumption. For example, in DianNao, the SRAM consumes 62% of the total on-chip power.

Overall, we have chosen the mapping in Fig. 3(d). Our PEs: 1) represent neurons; 2) are organized in a 2-D mesh; 3) receive, broadcasted, kernel elements $\omega_{i,j}$; 4) receive through right-left and up-down shifts the input feature map; and 5) accumulate locally the resulting output feature map.

Of course, the details of the mapping go well beyond the intuition of Fig. 3(d) and we will devote the complete Section VII 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.

IV. ShiDianNao Architecture: Computation

As illustrated in Fig. 5, ShiDianNao 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

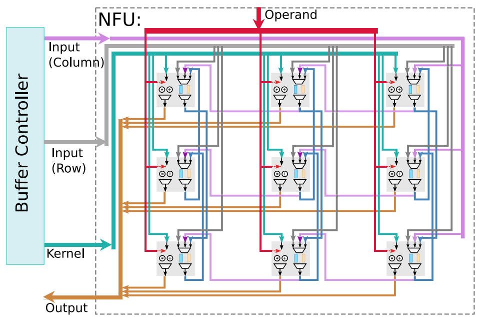


Fig. 6. NFU architecure.

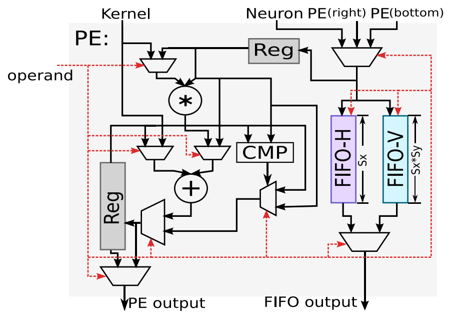


Fig. 7. PE architecture.

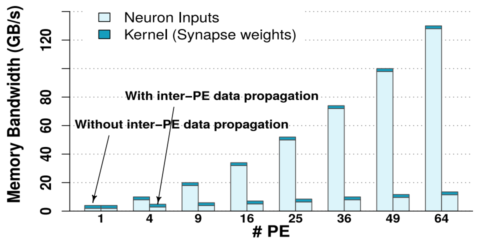


Fig. 8. Internal bandwidth from storage structures (input neurons and synapses) to NFU.

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 two-fold. First, using 16-bit fixed-point operators brings in negligible accuracy loss to neural networks, which has been validated by previous studies [2], [11], [25]. 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 [11].

A. Neural Functional Unit

Our accelerator processes 2-D feature maps (images), thus its NFU must be optimized to handle 2-D data (neuron/pixel arrays). The functional unit of DianNao [11] is inefficient for this application scenario, because it treats 2-D feature maps as a 1-D vector and cannot effectively exploit the locality of 2-D data. In contrast, our NFU is a 2-D mesh of $P_x \times P_y$ PEs, which naturally suits the topology of 2-D 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. First, 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 Fig. 6. 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 IV-A2). After performing computations, the NFU collects results from different PEs and sends them to NBout/NBin or the ALU.

1) 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 pooling layer, or a comparison for a max pooling layer, etc. (see Fig. 7). $\mathrm{PE}_{i,j}$, which is the PE at the ith row and jth 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 $\mathrm{PE}_{i+1,j}$ (right neighbor), or from $\mathrm{PE}_{i+1,j}$ (bottom neighbor), depending on the control signal. The $\mathrm{PE}_{i,j}$ 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 VII for detailed neuron-PE mappings).

2) 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 VII). 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 Fig. 8) using a representative convolutional layer (32×32 input feature map and 5×5 convolutional kernel) from LeNet-5 [12] 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 Fig. 8).

TABLE I
CNNs

CNN	Largest Layer Size (KB)	Synapses Size (KB)	Total Storage (KB)	Accuracy (%)
CNP [27]	15.19	28.17	56.38	97.00
MPCNN [28]	30.63	136.52	197.78	96.77
Face Recogn. [29]	21.33	61.14	103.8	96.20
LeNet-5 [12]	9.19	118.30	136.11	99.05
Simple conv. [30]	2.44	258.54	263.42	99.60
CFF [31]	7.00	1.72	18.49	—
NEO [32]	4.50	3.63	16.03	96.92
ConvNN [33]	45.00	4.35	87.53	96.73
Gabor [34]	2.00	0.82	5.36	87.50
Face align. [35]	15.63	29.27	56.39	—

B. Arithmetic Logic Unit

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 nonlinear 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 [11], [26]. 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.

V. 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 5.36 KB–263.11 KB storage is sufficient to simultaneously store all data required for many practical CNNs and, with only around 264 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 460 KB on-chip SRAM, which is sufficient for all ten practical CNNs listed in Table I. The cost of 460 KB SRAM is moderate: 2.95 mm^2 and 0.57 nJ per read in TSMC 65 nm process.

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 Fig. 9). 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.

VI. Accelerator Architecture: Control

A. 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 Fig. 10; the controller efficiently supports six read modes and a single write mode.

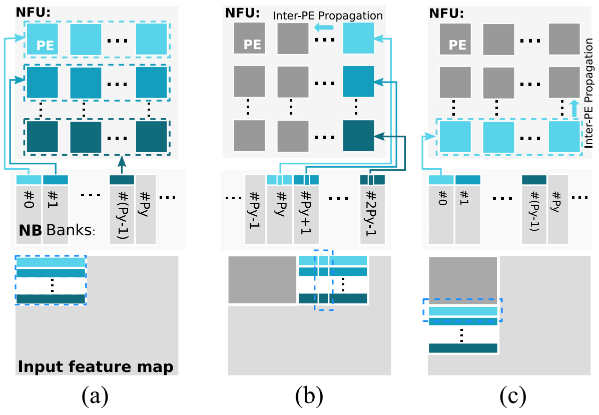


Fig. 9. Data stream in the execution of a typical convolutional layer, where we consider the most complex case: the kernel size is larger than the NFU size (#PEs), i.e., $K_x > P_x$ and $K_y > P_y$. (a) Cycle #0. (b) Cycle #1 #($K_x - 1$). (c) Cycle #K_x.

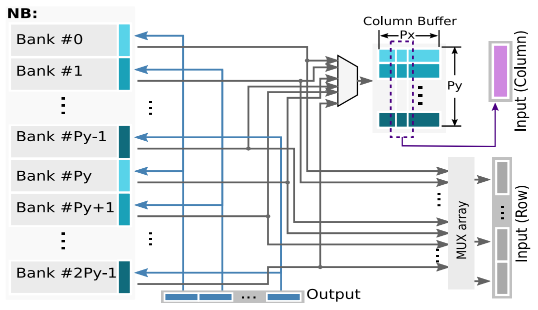


Fig. 10. NB controller architecture.

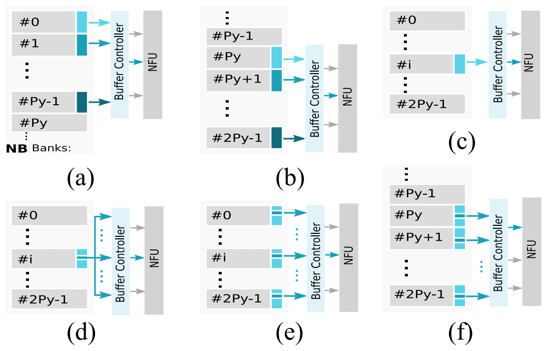


Fig. 11. Read modes of NB controller. (a) Read multiple banks (#0 to #Py-1). (b) Read multiple banks (#Py to #2Py-1). (c) Read one bank. (d) Read a single neuron. (e) Read neurons with a given step size. (f) Read a signle neuron per bank (#0 to #Py-1 or #Py to #2Py-1).

Without loss of generality, let us 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). Fig. 11 illustrates the six read modes of the NB controller.

1) Read multiple banks (#0 to #$P_y - 1$).
2) Read multiple banks (#P_y to #$2P_y - 1$).
3) Read one bank.
4) Read a single neuron.

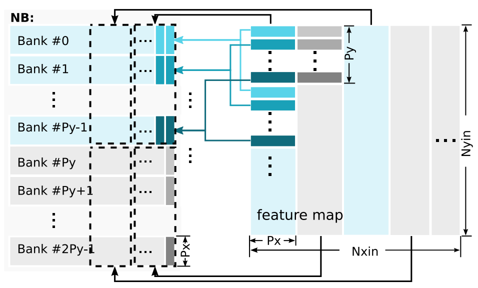


Fig. 12. Data organization of NB.

5) Read neurons with a given step size.
6) 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 1) or 2) to read $P_x \times P_y$ neurons from the NBin banks #0 to $\#P_y - 1$ or $\#P_y$ to $\#2P_y - 1$ [Fig. 9(a)], mode 5) to deal with the rare (but possible) cases in which the convolutional window is sliding with a step size larger than 1, mode 3) to read P_x neurons from an NB bank [Fig. 9(c)], and mode 6) to read P_y neurons from NB banks $\#P_y$ to $\#2P_y - 1$ or #0 to $\#P_y - 1$ [Fig. 9(b)]. For a pooling layer, we also use modes 1)–3), 5), and 6), since it has similar sliding windows (of input neurons) as a convolutional layer. For a normalization layer, we still use modes 1)–3), 5), and 6) because the layer is usually decomposed into sublayers behaving similar to convolutional and pooling layers (see Section VII). For a classifier layer, we use mode 4) 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 Fig. 10). 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 Fig. 12), the data block would be written to the first P_y banks of NB. Otherwise (when they lie in gray columns in the feature map of Fig. 12), the data block will be written to the second P_y banks of NB.

B. 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 Fig. 13). In the HFSM, first-level states describe abstract tasks processed by the accelerator (e.g., different layer types and ALU task). Associated

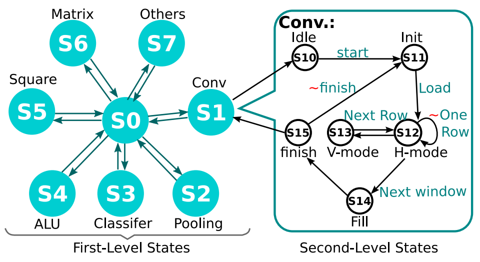


Fig. 13. Hierarchical control finite state machine.

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.

VII. CNN Mapping

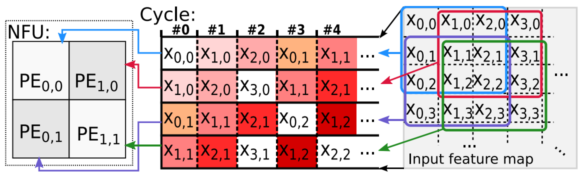
In this section, we show how different types of CNN layers are mapped to the accelerator design.

A. 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 Fig. 14 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 Fig. 14), 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 1)], 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


(a)

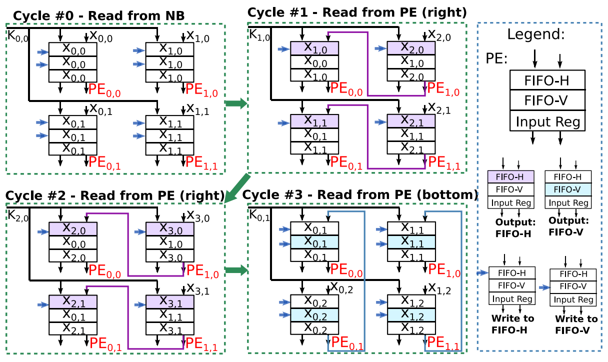

(b)

Fig. 14. Algorithm-hardware mapping between a convolutional layer (convolutional window: 3×3; step size: 1×1) and an NFU implementation (with 2×2 PEs). In (a), data marked with same colors are the same data but required by different PEs at different cycles (data with white color background do not have such reuse during the showed first five cycles). (b) Corresponding data movements in NFU.

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 6)], 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 6)]. All PEs share the kernel value $k_{2,0}$ read from SB. So far each PE has processed the first row 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 3)]. 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 5×5 and step size 1×1) [12] on a accelerator implementation having 64 PEs, inter-PE data propagations reduces by 73.88% internal bandwidth requirement between NFU and NBin (see also Fig. 8).

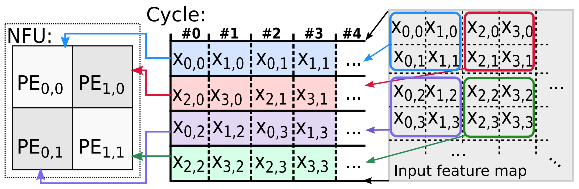


Fig. 15. Algorithm-hardware mapping between a pooling layer (pooling window: 2×2; step size: 2×2) and an NFU implementation (with 2×2 PEs).

B. 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 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 nonoverlapping, i.e., the step size of window sliding equals to the window size. We present in Fig. 15 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 Fig. 15), 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 5)]. 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.

C. 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 [12]). We present the general scheduling of a classifier layer in Fig. 16. When processing 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. In each cycle, $P_x \times P_y$ synaptic weights and a single input neuron for all

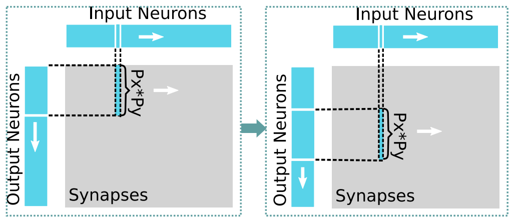


Fig. 16. Scheduling of classifier layer.

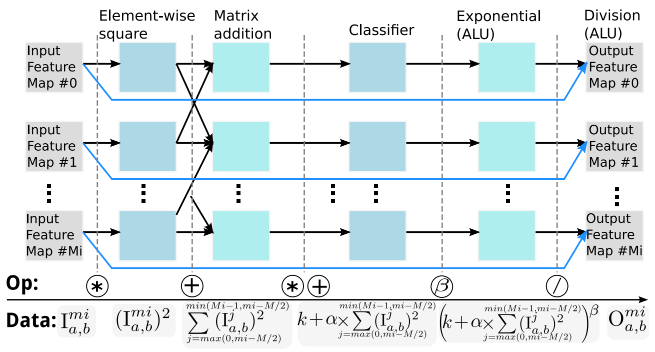


Fig. 17. Decomposition of an LRN layer.

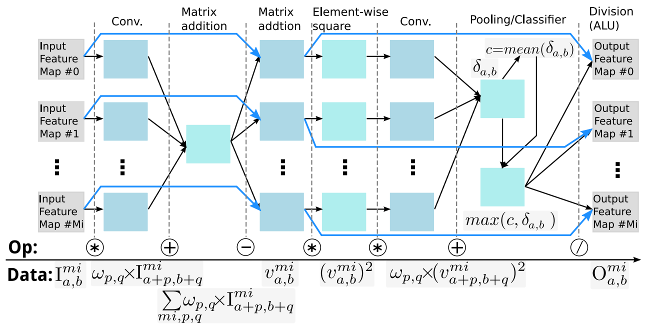


Fig. 18. Decomposition of an LCN layer.

$P_x \times P_y$ PEs are loaded to NFU (left part in Fig. 16). After that, each PE multiplies the synaptic weight and input neuron, and accumulates the result to the stored partial sum, for obtaining the dot product associated with an output neuron. The obtained result will be sent to the ALU for the computation of activation function. Then the PEs start to compute new $P_x \times P_y$ output neurons, as shown in the right part of Fig. 16.

D. Normalization Layers

Normalization layers can be composed into a number of sublayers and fundamental computational primitives in order to be executed by our accelerator. We illustrate detailed decompositions in Figs. 17 and 18, where an LRN layer is decomposed into a classifier sublayer, an element-wise square, a matrix addition, exponential functions and divisions; and an LCN layer is decomposed into two convolutional sublayers, a pooling sublayer, a classifier sublayer, two matrix additions, an element-wise square, and divisions. Convolutional, pooling, and classifier sublayers can be tackled with the rules described in former sections, 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 VI-A.

VIII. Experimental Methodology

A. 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 [36]. We compare our design with three baselines.

1) *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.
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 [37].
3) *Accelerator:* To make the comparison fair and adapted to the embedded scenario, we resized our previous work, i.e., DianNao [11] to have a comparable amount of arithmetic operators as our design—i.e., we implemented an 8×8 DianNao-NFU (eight hardware neurons, each processes eight input neurons and eight 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 reimplementation 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 reimplementation versus 3.02 mm^2 for the original DianNao [11], which tracks well the ratio in computing and storage resources.

B. Benchmarks

We collected ten CNNs from representative visual recognition applications and used them as our benchmarks (CNP [27], MPCNN [28], Face Recog. [29], LeNet-5 [12], Simple Conv [30], CFF [31], NEO [32], ConvNN [33], Gabor [34], and Face Align [35]). 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 II).

IX. Experimental Results

A. Layout Characteristics

We present in Tables II and III the parameters and layout characteristics of the current ShiDianNao version, respectively. ShiDianNao has 8×8 (64) PEs and a 64 KB NBin, a 64 KB NBout, a 300 KB SB, and a 32 KB IB. The overall SRAM capacity of ShiDianNao is 460 KB (17.7$\times$ larger than that of DianNao), in order to simultaneously store all data and instructions for a practical CNN. Yet, the total area of ShiDianNao

TABLE II
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	300 KB	16 KB
Inst. SRAM size	32 KB	8 KB

TABLE III
HARDWARE CHARACTERISTICS OF SHIDIANNAO AT 1 GHz, WHERE POWER AND ENERGY ARE AVERAGED OVER TEN BENCHMARKS

Accelerator	Area (mm^2)	Power (mW)	Energy (nJ)
Total	5.94 (100%)	336.51 (100%)	6770.13 (100%)
NFU+ALU	0.66 (11.11%)	268.82 (79.88%)	5515.87 (81.47%)
NBin	1.12 (18.86%)	38.18 (11.34%)	518.73 (7.66%)
NBout	1.12 (18.86%)	6.60 (1.96%)	86.62 (1.28%)
SB	2.96 (49.83%)	20.53 (6.10%)	266.29 (3.93%)
IB	0.31 (5.21%)	2.38 (0.71%)	36.21 (0.53%)

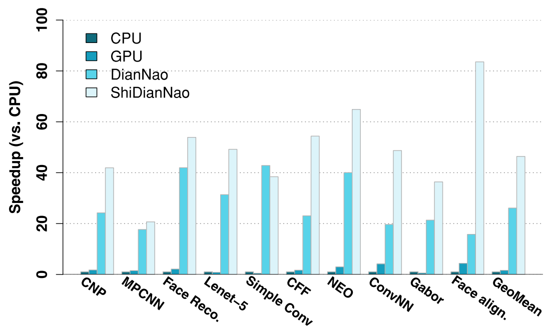


Fig. 19. Speedup of GPU, DianNao, and ShiDianNao over the CPU.

is only 4.30× larger than that of DianNao (5.94 mm^2 versus 1.38 mm^2).

B. Performance

We compare ShiDianNao against the CPU, the GPU, and DianNao on all benchmarks listed in Section VIII. The results are shown in Fig. 19. Unsurprisingly, ShiDianNao significantly outperforms the general purpose architectures and is, on average, 46.38 × faster than the CPU and 28.94× 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 I map poorly on its 2496 hardware threads.

More interestingly, ShiDianNao also outperforms our accelerator baseline on nine out of ten benchmarks (1.87× faster on average on all ten benchmarks). There are two main reasons for that: first, compared to DianNao, ShiDianNao eliminates off-chip memory accesses during execution, thanks to a sufficiently large SRAM capacity and a correspondingly slightly higher cost. Second, ShiDianNao efficiently exploits the locality of 2-D 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.

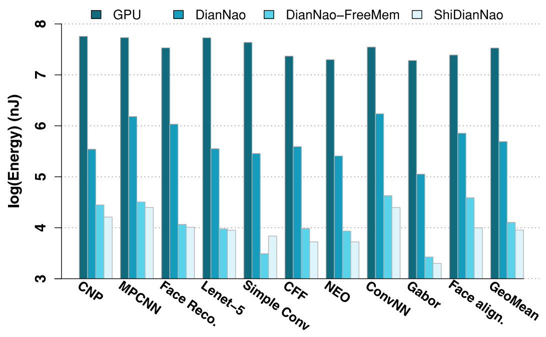


Fig. 20. Energy cost of GPU, DianNao, and ShiDianNao.

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 tradeoff 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 with a native overlapped segmentation way, the longest time to process a 640 × 480 video frame is for benchmark MPCNN which requires 0.079 ms to process a 32 × 32-pixel region. Since each frame contains $\lceil(640 - 16)/16\rceil \times \lceil(480 - 16)/16\rceil = 1131$ such regions (overlapped by 16 pixels, half of the region size), a frame takes a little more than 89 ms to process, resulting in a speed of 11 frames/s 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 × 480 resolution is in line with the fact that usually images are resized in certain range before processing [16], [23], [38], [39].

C. Energy

In Fig. 20, 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 3734.60× and 54.46× more energy efficient than GPU and DianNao, respectively. We also evaluate an ideal version of DianNao (DianNao-FreeMem, see Fig. 20), where we assume that main memory accesses incur no energy cost. Interestingly, we observe that ShiDianNao is still 1.41× 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 72.56× and 1.88× more energy efficient than DianNao and DianNao-FreeMem, respectively.

We illustrate in Table III the breakdown of the energy consumed by our design. We observe that four SRAM

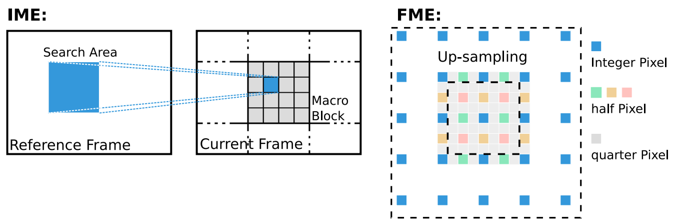


Fig. 21. IME: sub-block searching. FME: up-sampling.

TABLE IV
HARDWARE COMPARISON

	ShiDianNao	ME_ACC [42]
Area (mm^2)	5.94	0.67
Power (mW)	336.51.10	360.70
Freq. (GHz)	1.00	0.98

buffers account for only 13.61% the overall energy, and the rest is consumed by the logic (81.47%). This is significantly different from Chen *et al.*'s [11] observation made on DianNao, where more than 95% of the energy is consumed by the DRAM.

D. Applying ShiDianNao to Traditional Video Processing Tasks

By shifting the vision processing closer to the sensor, ShiDianNao eliminates most of the DRAM accesses, which significantly improves the efficiency of vision recognition tasks. In line with this philosophy, ShiDianNao can also be applied to traditional vision processing tasks, such as Demosaic, Bayer construction, motion estimation [40], and so on, because characteristics of these tasks fit ShiDianNao naturally. In the remain of this section, we take motion estimation, a time-consuming step accounting for ∼90% the computation time of video encoding [1], [41], as the driving example.

Specifically, in video standards (such as H.264), each frame is divided into blocks and further into sub-blocks. For each block/sub-block in the current frame, motion estimation finds the best-matching block/sub-block within a certain search area of the reference frame. The matching operation is performed in terms of original pixels and then in terms of fractional pixels, where fractional pixels are generated by up-sampling the original pixels. The two steps are known as integer motion estimation and fractional motion estimation (FME), see Fig. 21. The matching operation itself can be formulated as convolutional operations between current frame and reference frame, which can be efficiently accommodated with ShiDianNao. Moreover, the up-sampling operation for generating fractional pixels is formulated as pure convolutions, thus can be efficiently accommodated with ShiDianNao as well.

We report the results of ShiDianNao against ME_ACC [42], a motion estimation accelerator fabricated at 65 nm technology, on five popular 1080p (1920×1080) sequences (Basketball, Rush_Hour, Cactus, Pedestrian, Sunflower, and Riverbed), see Tables IV and V. We observed that ShiDianNao is not as efficient as ME_ACC in terms of area, power of motion estimation, which is in line with the conclusion made by Qadeer *et al.* [43]. We also notice a 0.1% PNSR loss and 1.7% BitRate increase when compared against the standard compress algorithm (with typical 32×32 search area).

TABLE V
RESULTS (COMPARED AGAINST DEFAULT SETTING OVER SIX BENCHMARKS)

Parameters	Value
Video	1920x1080
Search range	32x32
Max FPS	62
PSNR Loss (%)	0.1
BitRate Increase (%)	1.7

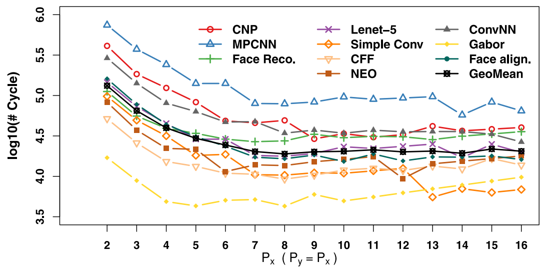


Fig. 22. Execution cycles of varying #PE implemented (#PE = $P_x \times P_y$).

Yet, this example demonstrates that the application scope of ShiDianNao is not restricted to neural networks, and can potentially be generalized to broader video processing tasks.

X. SHIDIANNAO+: MULTICORE EXTENSION OF SHIDIANNAO

The growing capability of sensor provides video streams with higher resolution and frame rate in mobile ends and wearable devices, thus it poses great challenges to real-time video processing. Although the current design of ShiDianNao can smoothly process 640×480 video streams at 11 frames/s with MPCNN, it is not sufficiently efficient for video streams, e.g., with a resolution higher as 640×480 or an frames/s higher than 11.

A. Increasing the Number of PEs: Intuitive Idea

An intuitive and immediate solution is to increase the number of PEs in ShiDianNao. We evaluated it with an empirical study, where we varied the number of PEs in ShiDianNao from 4 to 256 without modifying other parts of the accelerator architecture, and estimated it on ten benchmarks introduced in Section VIII. We reported performance results in Fig. 22, and we observed that the execution time with respect to each benchmark does not decrease proportionally to the number of PEs. Potential reasons are twofold. First, although most computations are performed concurrently, nonlinear operations are still computed sequentially as there is only a single ALU in the current design. Second, implementing more PEs in ShiDianNao may not always improve the performance. For instance, when the number of PEs is larger than the size of output feature map in convolutional layer, the execution time will not be reduced even we add more PEs in ShiDianNao.

B. Increasing the Number of ALUs

In order to elevate the performance of ShiDianNao, we add more ALUs in ShiDianNao to concurrently perform nonlinear operations. We evaluated ShiDianNao with the number of PEs varying from 2×2 to 16×16, and the number of ALUs from

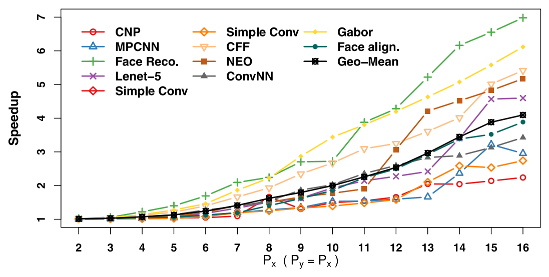


Fig. 23. Speedup of ShiDianNao with #P_y ALU versus #1 ALU.

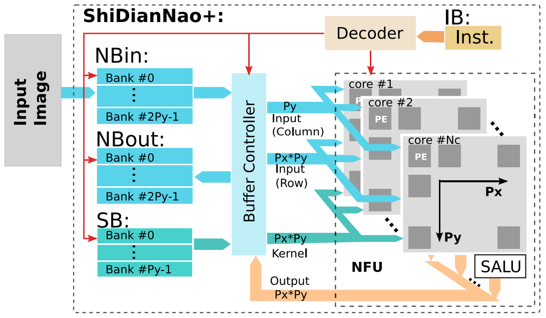


Fig. 24. ShiDianNao+ architecture.

2 to 16 on the ten benchmarks, and reported the performance results in Fig. 23. With an area of 0.015 mm^2 (2.3% of the NFU module) of an ALU module, we are able to implement P_y ALUs and enable 1.01× to 4.09× performance improvement averaging on ten benchmarks; 1.62× for ShiDianNao with 8 × 8 PEs. Yet, the performance improvement is still not sufficient to achieve, e.g., 10 frames/s for recognizing a 1280×720 video stream with MPCNN. In a nutshell, it is not a satisfactory solution to tackle high-resolution and high-frame-rate video streams via simply adding more PEs and ALUs to ShiDianNao.

C. ShiDianNao+: Architecture, Mappping, and Evaluation

In this section, we propose ShiDianNao+, a multicore extension of ShiDianNao, to perform visual recognition on video stream with high resolution and high frame rate.

1) ShiDianNao+ (Architecture): We illustrate the architecture of ShiDianNao+ in Fig. 24. ShiDianNao+ extends ShiDianNao by integrating multiple cores in the NFU, each core consists of an array of $P_x \times P_y$ PEs as well as P_y ALUs. ShiDianNao+ enables better performance on visual recognition than ShiDianNao, because: 1) the multicore settings in ShiDianNao+ allows different output feature maps to be computed in parallel and 2) the abundant ALUs integrated in each core allow nonlinear operations to be performed in parallel. We select the same data width as in ShiDianNao to avoid the costly super width data bus between storage and NFU, as well as to provide good scalability of core number. All the cores are connected to storage through shared bus lines to reduce the connection complexity. Data from NB are shared among enabled cores while data write-back are sequential among different cores through shared bus line; only SB can be accessed by different cores with different addresses but at different cycles deliberately through shared bus. We also modified the decoder module to allow ShiDianNao to select core for execution.

2) Mapping in ShiDianNao+: In a typical convolutional layer as shown in Fig. 25(a), multiple output feature maps, say N_c, can be computed simultaneously with N_c cores in ShiDianNao+, using a shared input feature map. The computations of neurons on the same output feature map, which are mapped on one of the N_c cores, are scheduled same as 1-core ShiDianNao. Each core will insist work on its output feature map and not move to next one until the current output feature map has been constructed. As illustrated in Fig. 25(b), an input feature map is fetch from NBin and broadcasts to all N_c cores while private convolutional kernels are fetched separately. In most cases, the kernel size is smaller than the feature map and kernel values are shared among a certain pair input/output feature maps. This, together with the fact that each connection between input and output feature maps are sharing a kernel thus the kernel need to be loaded only once for computing, leads to a negligible cycle costs.

Different from convolutional layer, we map pooling layer and classifier layer to only one of the cores to keep the simplicity of such less computation intensive layers. However, ShiDianNao+ can still enable performance improvement with scalability mainly from convolutional layer which consists the largest proportion of computation in a CNN. Thus, we enable only one core for such layers in our experiments in this paper.

3) Evaluating ShiDianNao+: We vary the number of NFU cores in ShiDianNao+ (from 1 to 4, each has 8×8 PEs), while fixing sizes of NBin, NBout and SB to be 64 KB, 64 KB, and 300 KB, respectively. We take ShiDianNao as the baseline, and report the results in Tables III and VI. We observed that NFU in ShiDianNao takes only 11% area but 81% energy cost, thus it leads to modest extra area cost of multicore design. The 1-core, 2-core, 3-core, and 4-core implementations of ShiDianNao+ cost consume 1.51%, 14.98%, 28.79%, and 43.10% more area than ShiDianNao, respectively.

a) Performance: We report the performance of ShiDianNao+ obtained by 1-core, 2-core, 3-core, and 4-core, and compare ShiDianNao+ against ShiDianNao in terms of the performance on ten benchmarks in Fig. 26, where ShiDianNao+ can achieve 1.62×, 2.29×, 2.87×, and 3.68× performance improvement, respectively. We note that the 1-core ShiDianNao+ which adds 8 ALUs to the original ShiDianNao performs better than 2-core ShiDianNao+ with benchmark Face Reco and ConvNN, which is mainly due to

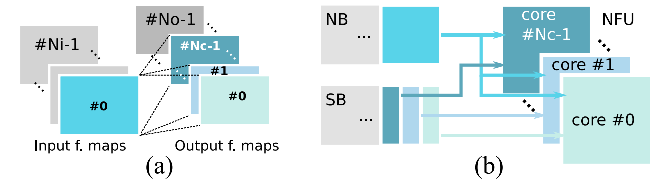


Fig. 25. ShiDianNao+ scheduling in convolutional layer. (a) ShiDianNao+ scheduling in convolutional layer. (b) Convolutional layer mapping in ShiDianNao+.

TABLE VI
HARDWARE CHARACTERISTICS OF SHIDIANNAO+ (POWER AND ENERGY ARE AVERAGED OVER TEN BENCHMARKS)

Accelerator	Area (mm^2)	Power (mW)	Energy (nJ)
ShiDianNao	5.94	336.51	6770.13
1-core	6.03	459.73	5405.89
2-core	6.83	776.29	5529.11
3-core	7.65	1102.80	5961.48
4-core	8.50	1440.04	5622.66

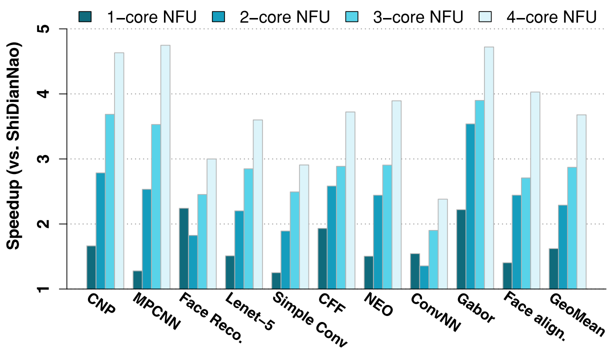


Fig. 26. Speedup of ShiDianNao+.

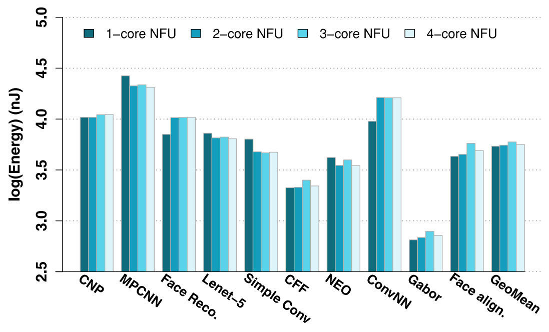


Fig. 27. Energy cost on ten benchmarks.

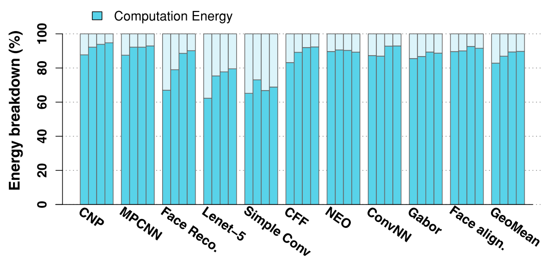


Fig. 28. Energy breakdown of ShiDianNao+ (Bar in each cluster: 1-core NFU, 2-core NFU, 3-core NFU, and 4-core NFU).

the mapping strategy we chose. Without loss of generality, we treat a convolutional layer as a fully connected layer where each input feature map will contribute to all output feature maps via convolutions; those originally nonexisting connections are now deemed as all-zero kernels. Moreover, it also leads to a mapping that is consistent for ShiDianNao+ versions with different number of cores. In particular, 1-core ShiDianNao+ can benefit from omitting the computation where kernels having zero values. Thus, 1-core ShiDianNao+ only computes the 125 connections in C3 layer of Face Reco. which has 20 input feature maps and 25 output feature maps, while 2-core ShiDianNao+ have to compute the 500 (25×20) connections. With regard to the real-time processing of video stream, the 1-core, 2-core, 3-core, and 4-core version of ShiDianNao+ are able to achieve, e.g., 25, 35, 44, and 56 frames/s for a 1280×720 video stream averaging on the ten benchmarks in the naive overlapped segmentation way. With the elevated performance, ShiDianNao+ is capable of handling more complex dealing with video stream in real-time.

b) Energy: Evaluated with the same methodology as ShiDianNao, we report the energy cost consumed by ShiDianNao+ in Fig. 27 and Table VI. We observed that increasing the number of cores does not reduce the energy consumption of ShiDianNao+, mainly because the costs of added cores can be well balanced by the speedup of performance. When integrating more cores, the energy spent on NFU increases but the energy for reading input feature maps may reduce as loading times reduces. In total, 1-core, 2-core, 3-core, and 4-core ShiDianNao+ achieve about 20.15%, 18.33%, 11.94%, and 16.95% energy savings against ShiDianNao. The average SRAM energy cost ratio over the ten benchmark reduces as the core number increases: 15.82%, 11.68%, 9.38%, and 8.84% against 13.61% of ShiDianNao, see the energy breakdown in Fig. 28. Overall, ShiDianNao+ provides an energy-efficient and scalable design for high-performance purpose.

XI. Related Work

A. 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 [44] and Samsung Gear [45]). Under emerging application scenarios such as image recognition/search [44], [46], 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 [29], [31], [34], [47] 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. This paper partially bridges this gap by shifting visual processing closer to sensors.

B. Neural Network Accelerators

Neural networks were conventionally executed on CPUs [48], [49], and GPUs [50]–[52]. 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. After a first wave of designs at the end of the last century [24], there have also been a few more modern application-specific accelerator architectures for various neural networks, with implementations on either field programmable gate arrays (FPGAs) [27], [53], [54] or ASICs [2], [11], [23]. For CNNs, Farabet *et al.* [23] proposed a systolic architecture called NeuFlow architecture, Chakradhar *et al.* [49] designed a systolic-like coprocessor. Although effective to handle 2-D convolution in signal processing [55]–[58], systolic architectures do not provide sufficient flexibility and efficiency to support different settings of CNNs [23], [49], [53], [59], 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.* [60] proposed a neural network stream processing core with an array of PEs, but the released version is still designed for MLPs. Peemen *et al.* [61] 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.* [62] 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 direct memory access. Recently, some of us [11] 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 2-D feature maps in a CNN, but instead treats them as 1-D 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 IX for experimental comparisons). Recent members of the DianNao family [63], [64] 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.

C. Irregular Neural Networks

Recent works have shown a great potential for DNNs to reduce the capacity of synapses. Han *et al* [56] proposed to obtain sparse deep neural networks by pruning the synapses and neurons. Rastegari [49] proposed a binary CNNs which can reduce the capacity of synapses in feed-forward path drastically. Although ShiDianNao is able to directly process such irregular neural networks, for example, the pruned synapses in SNN can be fed into ShiDianNao with zero values, and the outputs of each layer BNN can be treated as binary values, ShiDianNao can greatly benefit from such irregular neural networks with trivial extension. For example, an encoder/decoder module can be integrated into ShiDianNao to greatly reduce the off-chip memory access for processing the sparse neural networks.

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 [23], [49], [53], [59], 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.

XII. Conclusion

We designed a versatile accelerator for state-of-the-art visual recognition algorithms. Averaged on ten 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 [11]. ShiDianNao consumes only about 3700× and 54× less energy than the GPU and DianNao, respectively. ShiDian has an area of 5.94 mm^2 in a 65 nm process and consumes only 336.51 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. We further proposed ShiDianNao+ aiming to tackle the performance bottleneck but still with high efficiency. 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.

References

[1] R. Hameed *et al.*, "Understanding sources of inefficiency in general-purpose chips," in *Proc. Annu. Int. Symp Comput. Architect. (ISCA)*, Saint-Malo, France, 2010, pp. 37–47.

[2] O. Temam, "A defect-tolerant accelerator for emerging high-performance applications," in *Proc. 39th Annu. Int. Symp. Comput. Architect. (ISCA)*, Portland, OR, USA, 2012, pp. 356–367.

[3] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *Proc. 45th Annu. EEE/ACM Int. Symp. Microarchitect. (MICRO)*, Vancouver, BC, Canada, Dec. 2012, pp. 449–460.

[4] K. Fan, M. Kudlur, G. Dasika, and S. Mahlke, "Bridging the computation gap between programmable processors and hardwired accelerators," in *Proc. IEEE 15th Int. Symp. High Perform. Comput. Architect. (HPCA)*, Raleigh, NC, USA, 2009, pp. 313–322.

[5] G. Venkatesh *et al.*, "QSCORES: Trading dark silicon for scalable energy efficiency with quasi-specific cores categories and subject descriptors," in *Proc. 44th Annu. IEEE/ACM Int. Symp. Microarchitect. (MICRO)*, Porto Alegre, Brazil, 2011, pp. 163–174.

[6] S. Yehia, S. Girbal, H. Berry, and O. Temam, "Reconciling specialization and flexibility through compound circuits," in *Proc. IEEE 15th Int. Symp. High Perform. Comput. Architect. (HPCA)*, Raleigh, NC, USA, 2009, pp. 277–288.

[7] B. Liang and P. Dubey, "Recognition, mining and synthesis," *Intel Technol. J.*, vol. 9, no. 2, pp. 1–10, 2005.

[8] G. E. Dahl, T. N. Sainath, and G. E. Hinton, "Improving deep neural networks for LVCSR using rectified linear units and dropout," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process. (ICASSP)*, Vancouver, BC, Canada, 2013, pp. 8609–8613.

[9] V. Mnih and G. E. Hinton, "Learning to label aerial images from noisy data," in *Proc. 29th Int. Conf. Mach. Learn. (ICML)*, Edinburgh, U.K., 2012, pp. 567–574.

[10] P.-S. Huang *et al.*, "Learning deep structured semantic models for Web search using clickthrough data," in *Proc. Int. Conf. Inf. Knowl. Manag. (CIKM)*, San Francisco, CA, USA, 2013, pp. 2333–2338.

[11] T. Chen *et al.*, "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proc. 19th Int. Conf. Architect. Support Program. Lang. Oper. Syst. (ASPLOS)*, Salt Lake City, UT, USA, 2014, pp. 269–284.

[12] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.

[13] 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 *Proc. Int. Conf. Mach. Learn. (ICML)*, Corvallis, OR, USA, 2007, pp. 473–480.

[14] R. Salakhutdinov and G. Hinton, "Learning a nonlinear embedding by preserving class neighbourhood structure," in *Proc. Artif. Intell. Stat.*, vol. 3. San Juan, PR, USA, 2007, pp. 412–419.

[15] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, pp. 1106–1114.

[16] Q. V. Le *et al.*, "Building high-level features using large scale unsupervised learning," in *Proc. Int. Conf. Mach. Learn. (ICML)*, Edinburgh, U.K., 2012, pp. 8595–8598.

[17] A. Coates *et al.*, "Deep learning with COTS HPC systems," in *Proc. 30th Int. Conf. Mach. Learn. (ICML)*, Atlanta, GA, USA, 2013, pp. 1337–1345.

[18] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the future of parallel computing," *IEEE Micro*, vol. 31, no. 5, pp. 7–17, Sep./Oct. 2011.

[19] *STV0986, 5 Megapixel Mobile Imaging Processor (Data Brief)*, STMicroelectronics, Geneva, Switzerland, Jan. 2007.

[20] *STV0987, 8 Megapixel Mobile Imaging Processor (Data Brief)*, STMicroelectronics, Geneva, Switzerland, Mar. 2013.

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

[22] K. Jarrett, K. Kavukcuoglu, M. A. Ranzato, and Y. LeCun, "What is the best multi-stage architecture for object recognition?" in *Proc. IEEE 12th Int. Conf. Comput. Vis. (ICCV)*, Kyoto, Japan, pp. 2146–2153, Sep./Oct. 2009.

[23] C. Farabet *et al.*, "NeuFlow: A runtime reconfigurable dataflow processor for vision," in *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit. Workshops (CVPRW)*, Colorado Springs, CO, USA, Jun. 2011, pp. 109–116.

[24] P. Ienne, T. Cornu, and G. Kuhn, "Special-purpose digital hardware for neural networks: An architectural survey," *J. VLSI Signal Process.*, vol. 13, no. 1, pp. 5–25, 1996.

[25] Z. Du *et al.*, "Leveraging the error resilience of machine-learning applications for designing highly energy efficient accelerators," in *Proc. 19th Asia South Pac. Design Autom. Conf. (ASP-DAC)*, Singapore, Jan. 2014, pp. 201–206.

[26] 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—ISNN 2006* (LNCS 3973). Heidelberg, Germany: Springer, 2006, pp. 1319–1327.

[27] C. Farabet, C. Poulet, J. Y. Han, and Y. Lecun, "CNP: An FPGA-based processor for convolutional networks," in *Proc. Int. Conf. Field Program. Logic Appl. (FPL)*, vol. 1. Prague, Czech Republic, Aug./Sep. 2009, pp. 32–37.

[28] J. Nagi *et al.*, "Max-pooling convolutional neural networks for vision-based hand gesture recognition," in *Proc. IEEE Int. Conf. Signal Image Process. Appl. (ICSIPA)*, Kuala Lumpur, Malaysia, 2011, pp. 342–347.

[29] S. Lawrence, C. L. Giles, A. C. Tsoi, and A. D. Back, "Face recognition: A convolutional neural-network approach," *IEEE Trans. Neural Netw.*, vol. 8, no. 1, pp. 98–113, Jan. 1997.

[30] P. Y. Simard, D. Steinkraus, and J. C. Platt, "Best practices for convolutional neural networks applied to visual document analysis," in *Proc. 7th Int. Conf. Doc. Anal. Recognit. (ICDAR)*, vol. 1. Edinburgh, U.K., 2003, pp. 958–963.

[31] C. Garcia and M. Delakis, "Convolutional face finder: A neural architecture for fast and robust face detection," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 26, no. 11, pp. 1408–1423, Nov. 2004.

[32] C. Nebauer, "Evaluation of convolutional neural networks for visual recognition," *IEEE Trans. Neural Netw.*, vol. 9, no. 4, pp. 685–696, Jul. 1998.

[33] M. Delakis and C. Garcia, "Text detection with convolutional neural networks," in *Proc. Int. Conf. Comput. Vis. Theory Appl. (VISAPP)*, Funchal, Portugal, 2008, pp. 290–294.

[34] B. Kwolek, "Face detection using convolutional neural networks and Gabor filters," in *Artificial Neural Networks: Biological Inspirations—ICANN*. Heidelberg, Germany: Springer, 2005, pp. 551–556.

[35] S. Duffner and C. Garcia, "Robust face alignment using convolutional neural networks," in *Proc. Int. Conf. Comput. Vis. Theory Appl. (VISAPP)*, Funchal, Portugal, 2008, pp. 30–37.

[36] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0," in *Proc. 40th Annu. IEEE/ACM Int. Symp. Microarchitect. (MICRO)*, Chicago, IL, USA, Dec. 2007, pp. 3–14.

[37] Berkeley Vision and Learning Center. (Jul. 2015). *Caffe: A Deep Learning Framework*. [Online]. Available: http://caffe.berkeleyvision.org/

[38] M. A. Ranzato, F. J. Huang, Y.-L. Boureau, and Y. LeCun, "Unsupervised learning of invariant feature hierarchies with applications to object recognition," in *Proc. IEEE Comput. Vis. Pattern Recognit. (CVPR)*, Minneapolis, MN, USA, Jun. 2007, pp. 1–8.

[39] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors." *arXiv preprint arXiv:1207.0580 (2012).*

[40] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra, "Overview of the H.264/AVC video coding standard," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 13, no. 7, pp. 560–576, Jul. 2003.

[41] T.-C. Chen *et al.*, "Analysis and architecture design of an HDTV720p 30 frames/s H.264/AVC encoder," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 16, no. 6, pp. 673–688, Jun. 2006.

[42] D. Truong *et al.*, "A 167-processor 65 nm computational platform with per-processor dynamic supply voltage and dynamic clock frequency scaling," in *Proc. IEEE Symp. VLSI Circuits Dig. Tech. Papers*, Honolulu, HI, USA, 2008, pp. 22–23.

[43] W. Qadeer *et al.*, "Convolution engine: Balancing efficiency & flexibility in specialized computing," in *Proc. 40th Annu. Int. Symp. Comput. Architect. (ISCA)*, Tel Aviv, Israel, 2013, pp. 24–35.

[44] T. Starner, "Project glass: An extension of the self," *IEEE Pervasive Comput.*, vol. 12, no. 2, pp. 14–16, Apr./Jun. 2013.

[45] SAMSUNG, *SAMSUNG Gear2 Tech Specs*, Samsung Electron., Seoul, South Korea, 2014.

[46] Google. (Nov. 2015). *Google Image Search*. [Online]. Available: http://www.google.com/insidesearch/features/images/searchbyimage.htm

[47] S. Kamijo, Y. Matsushita, K. Ikeuchi, and M. Sakauchi, "Traffic monitoring and accident detection at intersections," *IEEE Trans. Intell. Transp. Syst.*, vol. 1, no. 2, pp. 108–118, Jun. 2000.

[48] V. Vanhoucke, A. Senior, and M. Z. Mao, "Improving the speed of neural networks on CPUs," in *Proc. Neural Inf. Process. Syst. Conf. Deep Learn. Unsupervised Feature Learn. Workshop (NIPS)*, 2011, pp. 1223–1231.

[49] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks," in *Proc. 37th Annu. Int. Symp. Comput. Architect. (ISCA)*, Saint-Malo, France, 2010, pp. 247–257.

[50] C. Farabet *et al.*, "Hardware accelerated convolutional neural networks for synthetic vision systems," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCS)*, Paris, France, May 2010, pp. 257–260.

[51] D. Scherer, H. Schulz, and S. Behnke, "Accelerating large-scale convolutional neural networks with parallel graphics multiprocessors," in *Artificial Neural Networks—ICANN 2010* (LNCS 6354), K. Diamantaras, W. Duch, and L. S. Iliadis, Eds. Heidelberg, Germany: Springer, 2010, pp. 82–91. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-15825-4_9

[52] D. C. Cireşan, U. Meier, J. Masci, L. M. Gambardella, and J. Schmidhuber, "Flexible, high performance convolutional neural networks for image classification," in *Proc. 22nd Int. Joint Conf. Artif. Intell. (IJCAI)*, Barcelona, Spain, 2011, pp. 1237–1242.

[53] M. Sankaradas *et al.*, "A massively parallel coprocessor for convolutional neural networks," in *Proc. 20th IEEE Int. Conf. Appl. Specific Syst. Architect. Process. (ASAP)*, Boston, MA, USA, Jul. 2009, pp. 53–60.

[54] P. Sermanet and Y. LeCun, "Traffic sign recognition with multi-scale convolutional networks," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, San Jose, CA, USA, Jul./Aug. 2011, pp. 2809–2813.

[55] J.-J. Lee and G.-Y. Song, "Super-systolic array for 2D convolution," in *Proc. IEEE Region 10 Conf. (TENCON)*, Hong Kong, 2006, pp. 1–4.

[56] H. T. Kung, L. M. Ruane, and D. W. L. Yen, "Two-level pipelined systolic array for multidimensional convolution," *Image Vis. Comput.*, vol. 1, no. 1, pp. 30–36, 1983.

[57] S. Y. Lee and J. K. Aggarwal, "Parallel 2D convolution on a Mesh connected array processor," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. PAMI-9, no. 4, pp. 590–594, Apr. 1987.

[58] V. Hecht and K. Ronner, "An advanced programmable 2D-convolution chip for, real time image processing," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCS)*, Singapore, 1991, pp. 1897–1900.

[59] S. A. Dawwd, "The multi 2D systolic design and implementation of convolutional neural networks," in *Proc. IEEE 20th Int. Conf. Electron. Circuits Syst. (ICECS)*, Abu Dhabi, UAE, Dec. 2013, pp. 221–224.

[60] H. Esmaeilzadeh, P. Saeedi, B. N. Araabi, C. Lucas, and S. M. Fakhraie, "Neural network stream processing core (NnSP) for embedded systems," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCS)*, 2006, pp. 2773–2776.

[61] M. Peemen, A. A. A. Setio, B. Mesman, and H. Corporaal, "Memory-centric accelerator design for convolutional neural networks," in *Proc. Int. Conf. Comput. Design (ICCD)*, Asheville, NC, USA, Oct. 2013, pp. 13–19.

[62] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello, "A 240 G-ops/s mobile coprocessor for deep neural networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. Workshops (CVPRW)*, Columbus, OH, USA, 2014, pp. 696–701.

[63] Y. Chen *et al.*, "DaDianNao: A machine-learning supercomputer," in *Proc. 47th Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, Cambridge, U.K., 2015, pp. 609–622.

[64] D. Liu *et al.*, "PuDianNao: A polyvalent machine learning accelerator," in *Proc. 20th Int. Conf. Architect. Support Program. Lang. Oper. Syst. (ASPLOS)*, Istanbul, Turkey, 2015, pp. 369–381.

Authors' photographs and biographies not available at the time of publication.