A High-Throughput Neural Network Accelerator

THE AUTHORS DESIGNED AN ACCELERATOR ARCHITECTURE FOR LARGE-SCALE NEURAL NETWORKS, WITH AN EMPHASIS ON THE IMPACT OF MEMORY ON ACCELERATOR DESIGN, PERFORMANCE, AND ENERGY. IN THIS ARTICLE, THEY PRESENT A CONCRETE DESIGN AT 65 NM THAT CAN PERFORM 496 16-BIT FIXED-POINT OPERATIONS IN PARALLEL EVERY 1.02 NS, THAT IS, 452 GOP/S, IN A 3.02MM^2, 485-MW FOOTPRINT (EXCLUDING MAIN MEMORY ACCESSES).

Tianshi Chen
Zidong Du
Ninghui Sun
Jia Wang
Chengyong Wu
Yunji Chen
State Key Laboratory of Computer Architecture

Olivier Temam
INRIA

As architectures evolve toward heterogeneous multicores composed of a mix of cores and accelerators, designing accelerators that realize the best possible tradeoff between flexibility and efficiency is becoming a prominent issue.

The first question is, for which category of applications should we primarily design accelerators? Together with the architecture trend toward accelerators, a second simultaneous and significant trend in high-performance and embedded applications is developing: many of the emerging high-performance and embedded applications—from image, video, and audio recognition to automatic translation, business analytics, and all forms of robotics—rely on machine-learning techniques. This trend in application comes together with a third and equally remarkable trend in machine learning, in which a small number of techniques, based on neural networks (especially deep learning algorithms such as convolutional neural networks[1] and deep neural networks[2]), have been proved in the past few years to be the state of the art across a broad range of applications. This presents a unique opportunity to design accelerators that can realize the best of both worlds: significant application scope together with high performance and efficiency due to the limited number of target algorithms.[3]

Currently, these workloads are executed mostly on multicores using single-instruction, multiple data (SIMD),[4] on GPUs,[5] or on field-programmable gate arrays (FPGAs).[6] However, the aforementioned trends have already been identified by a few researchers who have proposed accelerators implementing convolutional neural networks[6] or multilayer perceptrons[7]; accelerators focusing on other domains, such as image processing, also propose efficient implementations of some of the primitives used by machine-learning algorithms, such as convolutions.[8] Others have proposed application-specific integrated circuit (ASIC) implementations of convolutional neural networks[9] or of other custom neural network algorithms.[10] However, all these works have first, and successfully, focused on efficiently implementing the computational primitives, but they either voluntarily ignore memory transfers for the sake of simplicity,[7,8] or they directly plug their computational accelerator to memory via a more

Published by the IEEE Computer Society

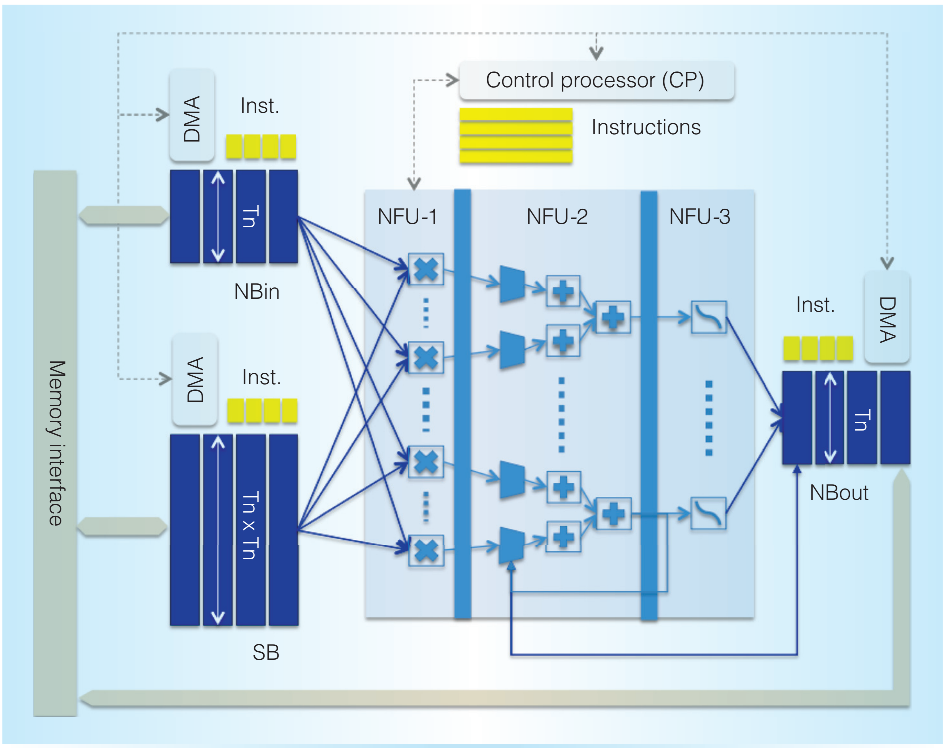

Figure 1. DianNao accelerator architecture. DianNao comprises an input buffer for input neurons, an output buffer for output neurons, and a buffer for synaptic weights. All three buffers connect to the neural functional unit and the control logic.

or less sophisticated direct memory access (DMA).[6,9,10]

Although efficient implementation of computational primitives is a first and important step with promising results, inefficient memory transfers can potentially void the throughput, energy, or cost advantages of accelerators (that is, an Amdahl's law effect) and thus they should become a first-order concern, just as in processors, rather than an element factored in accelerator design on a second step. Unlike in processors, though, we can factor in the specific nature of memory transfers in target algorithms, just as is done for accelerating computations. This is especially important in the domain of machine learning, in which we see a clear trend toward scaling up the size of neural networks in order to achieve better accuracy and more functionality.[2,11]

In this article, we present the DianNao accelerator for large-scale neural networks. We focus our study on memory usage, and we investigate the accelerator architecture to minimize memory transfers and to perform them as efficiently as possible.

Accelerator architecture

DianNao comprises an input buffer for input neurons (NBin), an output buffer for output neurons (NBout), and a buffer for synaptic weights (SB). All three buffers are connected to the *neural functional unit* (NFU), which is a computational block that performs both synapse and neuron computations, and to the control logic (see Figure 1).

NFU

The NFU implements a functional block of T_i inputs and synapses and T_n output neurons, which can be time-shared by different algorithmic blocks of neurons. Depending on the layer type, computations at the NFU can be decomposed in either two or three stages. For classifier and convolutional layers, the stages are multiplication of synapses and inputs, additions of all multiplications, and sigmoid. The nature of the last stage (sigmoid or another nonlinear function) can vary. For pooling layers, there is no multiplication (no synapse), and the pooling operations can be average or max. Note that the adders have multiple inputs; they are in fact *adder trees* (see Figure 1). The second stage also contains shifters and max operators for pooling layers. In the NFU, the sigmoid function (for classifier and convolutional layers) can be implemented efficiently using piecewise linear interpolation ($f(x) = a_x \times x + b_x, x \in [x_i : x_{i+1}]$) with negligible loss of accuracy (16 segments are sufficient).[12]

On-chip storage

DianNao's on-chip storage structures can be construed as modified buffers of scratchpads. Whereas a cache is an excellent storage structure for a general-purpose processor, it is a suboptimal way to exploit reuse because of the cache access overhead (for example, tag check, associativity, line size, and speculative read) and cache conflicts. The efficient alternative, scratchpad, is used in very long instruction word (VLIW) processors, but it's known to be difficult to compile for. However, a scratchpad in a dedicated accelerator realizes the best of both worlds: efficient storage and both efficient and easy exploitation of locality because only a few algorithms must be manually adapted.

We split on-chip storage into three structures (NBin, NBout, and SB), because there are three types of data (input neurons, output neurons, and synapses) with different characteristics (such as read width and reuse distance). Splitting structures has two benefits: first, to tailor the static RAMs (SRAMs) to the appropriate read and write widths, and second, to avoid conflicts that would occur in a cache. Moreover, we implement three DMAs to exploit the spatial locality of data, one for each buffer (two load DMAs for inputs, one store DMA for outputs).

Loop tiling

DianNao leverages loop tiling to minimize memory accesses, and thus efficiently accommodates large neural networks. For the sake of brevity, in this article we discuss only a classifier layer that has N_n output neurons, fully connected to N_i inputs. (For a detailed discussion of other layer types, see Chen et al.[13]) Figure 2 presents the classifier's original code, as well as the tiled code that maps the classifier layer to DianNao.

In the tiled code, the loops *ii* and *nn* reflect the fact that the NFU is a functional block of T_i inputs and synapses and T_n output neurons. On the other hand, input neurons are reused for each output neuron, but because the number of input neurons can range from a few tens to hundreds of thousands, they often will not fit in DianNao's NBin size. Therefore, we further tile loop *ii* (input neurons) with tile factor T_{ii}. A typical tradeoff of tiling is that improving one reference (here, *neuron*[*i*] for input neurons) increases the reuse distance of another reference (*sum*[*n*] for partial sums of output neurons), so we must tile for the second reference as well—hence, loop *nnn* and the tile factor T_{nn} for output neurons partial sums. Synapses now dominate the layer memory behavior. In a classifier layer, all synapses are usually unique, and thus there is no reuse within the layer. Overall, tiling drastically reduces the classifier layer's total memory bandwidth requirement, and we observe an approximately 50 percent reduction in our empirical study.[13]

Experiments

We empirically evaluate DianNao and compare it against SIMD and GPU baselines.

Methodology

We implemented a custom cycle-accurate, bit-accurate C++ simulator of the accelerator. We used this simulator to measure time in number of cycles. It is plugged in to a main memory model, which allows a bandwidth of up to 250 Gbytes per second. We also implemented a Verilog version of the accelerator and processed it with Synopsys tools.

We implemented a SIMD baseline using the GEM5+McPAT combination.[14] We used a four-issue superscalar x86 core with a 128-bit (8- × 16-bit) SIMD unit (SSE/SSE2), clocked at 2 GHz. The core has a 192-entry reorder buffer and a 64-entry load/store queue. The L1 data (and instruction) cache is 32 Kbytes, and the L2 cache is

```
for (int n = 0; n < Nn; n++)
       sum[n] = 0;
for (int n = 0; n < Nn; n++) // output neurons
       for (int i = 0; i < Ni; i++) // input neurons
          sum[n] += synapse [n][i] * neuron[i];
for (int n = 0; n < Nn; n++)
       neuron[n] = sigmoid(sum[n]);
(a)

for (int nnn = 0; nnn < Nn; nnn += Tnn) { // tiling for output neurons
      for (int iii = 0; iii < Ni; iii += Tii) { // tiling for input neurons
        for (int nn = nnn; nn < nnn + Tnn; nn += Tn) {
          for (int n = nn; n < nn + Tn; n++)
            sum[n] = 0;
          for (int ii = iii; ii < iii + Tii; ii += Ti)
            for (int n = nn; n < nn + Tn; n++)
              for (int i = ii; i < ii + Ti; i++)
                sum[n] += synapse[n][i] * neuron[i];

          for (int n = nn; n < nn + Tn; n++)
            neuron[n] = sigmoid(sum[n]);
}     } }
(b)
```

Figure 2. Pseudocode for a classifier layer: (a) original code; (b) tiled code. Loop tiling enables more efficient data reuse and drastically reduces the classifier layer's total memory bandwidth requirement.

2 Mbytes; both caches are eight-way associative and use a 64-byte line. These cache characteristics correspond to those of the Intel Core i7. In addition, we also employed Nvidia K20M (28-nm process, 5-Gbyte GDDR5, 3.52 Tflops peak) as the GPU baseline.

We employed several representative layer settings as benchmarks of our experimental comparisons (see Table 1). In a previous study, we had implemented tiled code of these benchmarks for DianNao, SIMD, and the GPU. For the GPU baseline, however, we no longer used our previous tiled code as in our previous study,[13] but instead employed a state-of-the-art open source neural network library called Caffe to gain reproducibility.[15] In particular, Caffe doesn't apply to the CONV5 layer because it exceeds the global memory capacity. For this specific layer, we employed our work-in-progress neural network library, libCNN.[16] libCNN will be released in the near future.

Accelerator characteristics after layout

The current version uses $T_n = 16$ (16 hardware neurons with 16 synapses each), so that the design contains 256 16-bit truncated multipliers (for classifier and convolutional layers), 16 adder trees of 15 adders each (for the same layers, plus a pooling layer if average is used), as well as a 16-input shifter and max (for pooling layers), and 16 16-bit truncated multipliers plus 16 adders (for classifier and convolutional layers, and optionally for pooling layers). For classifier and convolutional layers, multipliers and adder trees are active every cycle, achieving $256 + 16 \times 15 = 496$ fixed-point operations every cycle; at 0.98 GHz, this amounts to 452 GOP/s (giga fixed-point operations per second). We performed the synthesis and layout of the accelerator at 65 nm using Synopsys tools.

Performance

Figure 3 shows the speedups of DianNao over SIMD and the GPU. We observe that

Table 1. Benchmark layers. (Conv: convolutional, Pool: pooling, Class: classifier.)[†]

Layer	Nx	Ny	Kx	Ky	Ni	No	Description
Conv1	500	375	9	9	32	48	Street scene parsing (convolutional neural network)[9]—for example, identifying "building," "vehicle," etc.
Pool1	492	367	2	2	12	N/A	
Class1	N/A	N/A	N/A	N/A	960	20	
Conv2*	200	200	18	18	8	8	Detection of faces in YouTube videos (deep neural network)[11]
Conv3	32	32	4	4	108	200	Traffic sign identification for car navigation (CNN)[17]
Pool3	32	32	4	4	100	N/A	
Class3	N/A	N/A	N/A	N/A	200	100	
Conv4	32	32	7	7	16	512	Google Street View house numbers (CNN)[18]
Conv5*	256	256	11	11	256	384	Multiobject recognition in natural images (DNN),[2] winner of the 2012 ImageNet competition
Pool5	256	256	2	2	256	N/A	

*Private kernels.
†Nx and Ny are sizes of an input feature map, Kx and Ky are kernel sizes, and Ni and No are numbers of input and output feature maps.

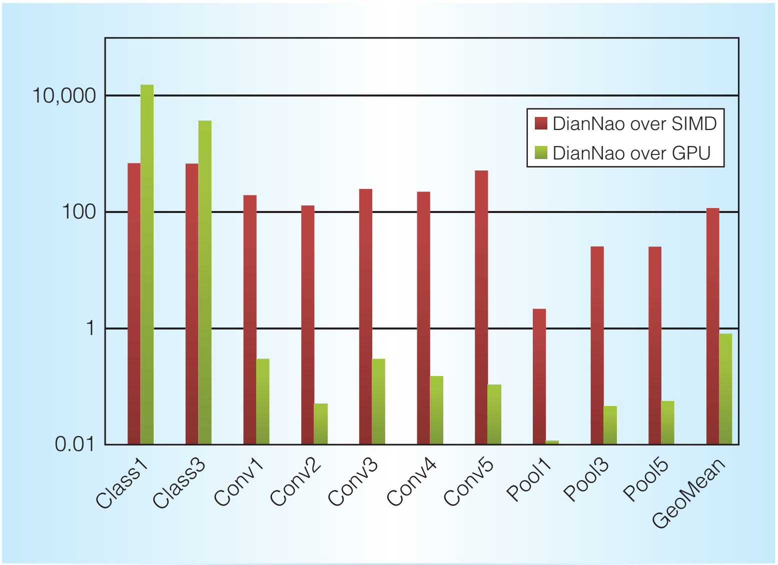

Figure 3. Speedups of DianNao over SIMD and the GPU. DianNao significantly outperforms SIMD, with an average speedup of 117.87×. However, DianNao is 0.81× slower than the GPU baseline on average.

DianNao significantly outperforms SIMD, and the average speedup is 117.87×. The main reasons are twofold. First, DianNao performs 496 16-bit operations every cycle for both classifier and convolutional layers—that is, 62× more (496/8) than the peak performance of the SIMD baseline. Second, compared with the SIMD baseline without a prefetcher, DianNao has better latency tolerance due to an appropriate combination of preloading and reuse in NBin and SB buffers.

We also observe that DianNao is 0.81× slower than the GPU baseline on average. Interestingly, DianNao is much faster than the GPU baseline on two classifier layers, because DianNao efficiently supports data reuse of NBin and SB buffers particularly common in classifier layers whose output neurons are fully connected to all inputs. In addition, DianNao implements hardware support to nonlinear activation functions that are inefficient in the GPU. On the other hand, DianNao is slower than the GPU baseline on both convolutional and pooling layers, because data reuse in such layers is less common than in classifier layers (actually, there is no data reuse in three pooling layers), and the GPU baseline employed the latest GPU libraries (Caffe[15] and libCNN[16]) optimized for convolutional layers.

Energy

Figure 4 provides DianNao's energy reductions over SIMD and the GPU. We observe that DianNao consumes 21.08× less energy than SIMD on average. This number is actually more than an order of magnitude smaller than previously reported energy ratios between processors and accelerators; for instance, Hameed et al. report an energy ratio of about 500×,[19] and 974× has been reported for a small multilayer perceptron.[7] The smaller ratio is largely due to the energy spent in memory accesses, which was voluntarily not factored in the two aforementioned studies. As in these two accelerators and others, the energy cost of computations has been considerably reduced by a combination of more efficient computational operators (especially a massive number of small 16-bit fixed-point truncated multipliers in our case), and small custom storage located close to the operators (64-entry NBin, NBout, SB, and the NFU-2 registers). As a result, there is now an Amdahl's law effect for energy, wherein any further improvement can be achieved only by bringing down the energy cost of main memory accesses. We tried to artificially set the energy cost of the main memory accesses in both the SIMD and accelerator to 0, and we observed that the average energy reduction of the accelerator increases by more than one order of magnitude, in line with previous results.

The average energy reduction of DianNao over the GPU baseline (Nvidia K20M) is 2.18×. Similar to the prior case, the small energy reduction is largely due to the energy spent in memory accesses. While K20M uses a state-of-the-art GDDR5 memory manufactured at a 55-nm process, DianNao is plugged in to a DDR3 memory at an 80-nm process. Even at the same process, GDDR5 is still 2.39× more energy efficient than DDR3.[20] Therefore, we can expect that DianNao's energy reduction over the GPU will significantly increase when using similar main memory.

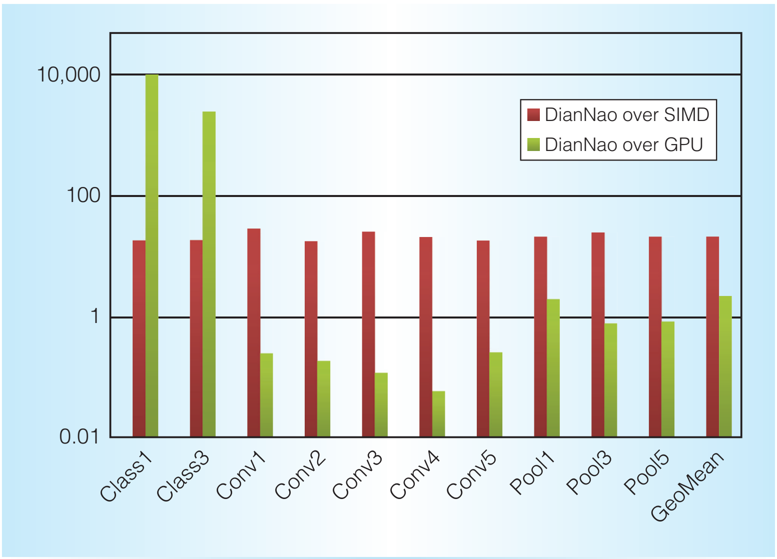

Figure 4. Energy reductions of DianNao over SIMD and the GPU. On average, DianNao consumes 21.08× less energy than SIMD and 2.18× less energy than the GPU.

Two simultaneous trends have emerged in the machine learning community: the input data size is continuously growing, and the learning model is becoming more complex. In the trends, machine learning techniques are requesting much more computational resources than ever before, which poses a great challenge for computer hardware at the bottom of the application stack. Our work takes a step toward addressing this challenge but still leaves many open topics for the community to investigate.

One such challenge is in large-scale learning models. Recent advances on deep learning suggest larger and larger neural networks, some of which even involve tens of billions of model parameters (synapse weights). To serve a large network at high performance, the collaboration of multiple CPUs and GPUs has become a necessity in industry. By replacing CPUs and GPUs with DianNao-like accelerators, a multichip system can achieve higher performance with the same power budget. Potential architecture topics could include chip architecture (functional units and memory hierarchy), interchip communications, and related algorithm-architecture co-optimizations. In our recent study, we proposed a custom multichip architecture (DaDianNao) that requires no memory access to accommodate large neural networks.[21] Each chip contains sufficiently large RAM that the RAMs of all chips can together keep the whole neural network. A 64-chip DaDianNao system is

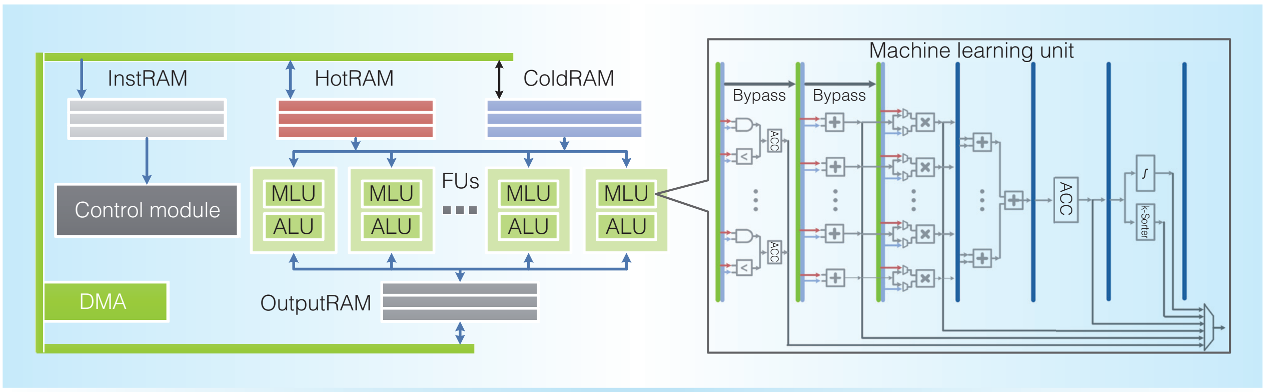

Figure 5. PuDianNao architecture. PuDianNao extends the DianNao architecture by providing functional units that are more flexible than the NFU in DianNao, and it efficiently accommodates representative machine learning techniques.

450.65× faster and consumes 150.31× less energy than an Nvidia K20M GPU.

Another topic for the research community is diverse machine learning techniques. Although neural network techniques have become state of the art for many applications, there are still scenarios under which a classical machine learning technique is better. For example, when classifying a linearly separable dataset, a nonlinear classifier can easily become overfitting, and a linear classifier has been qualified. In theory, a learning technique cannot universally beat another one over all possible datasets (that is, different techniques fit different datasets), according to the no-free-lunch theorem in machine learning.[22] When there is little prior knowledge about the data, practitioners usually find the best technique via iterative trial and error, and it would be desirable to have an accelerator supporting a basket of different machine learning techniques, not just a single technique or technique family.

In our recent study, we proposed an accelerator architecture called PuDianNao (see Figure 5) that factors in computational primitives and locality properties of different machine learning techniques.[23] So far it can accommodate seven representative machine learning techniques (*k*-means, *k*-nearest neighbors, naive bayes, support vector machine, linear regression, classification tree, and deep neural network). We are managing to support more machine learning techniques using this architecture, and our study could offer some insight on developing a general-purpose machine learning accelerator in the future.

Another topic of interest is spiking neural network algorithms. Our accelerator is designed for traditional neural network models developed by the machine learning community. Recently, researchers have become interested in another branch of neural network models called *spiking neural networks,* which use temporally dependent spike trains to encode information. They are inspired by recent advances in the neuroscience domain and are widely acknowledged to be more biologically plausible than traditional neural network models. Although it's still unclear whether spiking networks are really better than traditional networks in machine learning applications, they could be promising models for brain simulation. Future neural network accelerators might need to accommodate both traditional networks and spiking networks to balance practical usages and scientific exploitations.

One technology that's becoming more mature in industry is 3D stacking, which is particularly useful for supporting memory-intensive workloads like large-scale neural networks. 3D stacking is an intuitive yet promising idea to stack a neural network chip with memory chips. However, DianNao's architecture does not directly fit 3D chip stacking because we have not optimized it to minimize bandwidth requirements on through-silicon vias. Designing an appropriate (neural network) accelerator architecture that suits 3D stacking remains an open topic.

Finally, users still must write tedious assembly language for our accelerator, because no suitable high-level language and compiler exist. To facilitate users and enable broader applications of neural network accelerators, it is crucial to develop an easy-to-use library or programming language that effectively and concisely describes the network topology, learning algorithm, and memory behaviors of a neural network, and, in the meantime, encapsulates low-level hardware details. MICRO

Acknowledgments

Previous versions of this article appear in *Proceedings of 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (2014) and *ACM Transactions on Computer Systems* (accepted in 2014). This work is partially supported by the NSF of China (under grants 61100163, 61133004, 61222204, 61221062, 61303158, 61432016, 61472396, and 61473275), the 973 Program of China (under grant 2015CB358800), the Strategic Priority Research Program of the CAS (under grants XDA06010403 and XDB02040009), the International Collaboration Key Program of the CAS (under grant 171111KYSB201 30002), and the 10000 talent program. Olivier Temam is the corresponding author of this paper.

References

1. Y. Lecun et al., "Gradient-Based Learning Applied to Document Recognition," *Proc. IEEE*, vol. 86, no. 11, 1998, pp. 2278–2324.
2. G. Hinton and N. Srivastava, "Improving Neural Networks by Preventing Co-adaptation of Feature Detectors," *arXiv*, 2012, pp. 1–18; http://arxiv.org/abs/1207.0580.
3. T. Chen et al., "BenchNN: On the Broad Potential Application Scope of Hardware Neural Network Accelerators," *Proc. IEEE Int'l Symp. Workload Characterization*, 2012, pp. 36–45.
4. V. Vanhoucke, A. Senior, and M.Z. Mao, "Improving the Speed of Neural Networks on CPUs," *Deep Learning and Unsupervised Feature Learning Workshop*, 2011, pp. 1–8.
5. A. Coates et al., "Deep Learning with COTS HPC Systems," *Int'l Conf. Machine Learning*, 2013; http://jmlr.org/proceedings /papers/v28/coates13.html.
6. S. Chakradhar et al., "A Dynamically Configurable Coprocessor for Convolutional Neural Networks," *Proc. 37th Ann. Int'l Symp. Computer Architecture*, ACM Press, 2010, p. 247–257.
7. O. Temam, "A Defect-Tolerant Accelerator for Emerging High-Performance Applications," *Proc. 39th Ann. Int'l Symp. Computer Architecture*, 2012, pp. 356–367.
8. W. Qadeer et al., "Convolution Engine: Balancing Efficiency Flexibility in Specialized Computing," *Proc. 40th Ann. Int'l Symp. Computer Architecture*, 2013, pp. 24–35.
9. C. Farabet et al., "NeuFlow: A Runtime Reconfigurable Dataflow Processor for Vision," *Proc. IEEE Comp. Soc. Conf. Comp. Vision and Pattern Recognition Workshop*, 2011, pp. 109–116.
10. J.Y. Kim et al., "A 201.4 GOPS 496 mW Real-Time Multi-Object Recognition Processor With Bio-Inspired Neural Perception Engine," *IEEE J. Solid-State Circuits*, vol. 45, no. 1, 2010, pp. 32–45.
11. Q.V. Le et al., "Building High-Level Features using Large Scale Unsupervised Learning," *Int'l Conf. Machine Learning*, 2012; http:// icml.cc/2012/papers/73.pdf.
12. D. Larkin, A. Kinane, and N.E. O'Connor, "Towards Hardware Acceleration of Neuroevolution for Multimedia Processing Applications on Mobile Devices," *Neural Information Processing*, LNCS 4234, 2006, pp. 1178–1188.
13. T. Chen et al., "A Small-Footprint Accelerator for Large-Scale Neural Networks," to be published in *ACM Trans. Computer Systems*, 2015.
14. S. Li et al., "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," *Proc. 42nd Ann. IEEE/ACM Int'l Symp. Microarchitecture*, 2009, pp. 469–480.
15. Y. Jia et al., "Caffe: Convolutional Architecture for Fast Feature Embedding," *arXiv*, 2014; http://arxiv.org/abs/1408.5093.
16. L. Li et al., *libCNN: An Efficient GPU Library for Convolutional Neural Networks*, tech. report, Institute of Computing Technology, Chinese Academy of Sciences, 2015.
17. P. Sermanet and Y. LeCun, "Traffic Sign Recognition with Multi-scale Convolutional

Networks," *Int'l Joint Conf. Neural Networks*, 2011, pp. 2809–2813.

18. P. Sermanet, S. Chintala, and Y. LeCun, "Convolutional Neural Networks Applied to House Numbers Digit Classification," *Proc. 21st Int'l Conf. Pattern Recognition*, 2012, pp. 3288–3291.
19. R. Hameed et al., "Understanding Sources of Inefficiency in General-Purpose Chips," *Proc. 37th Ann. Int'l Symp. Computer Architecture*, ACM Press, 2010, pp. 37–47.
20. N.K. Mishra et al., "An Output Structure for a Bi-modal 6.4-gbps GDDR5 and 2.4-gbps DDR3 Compatible Memory Interface," *Proc. IEEE Custom Integrated Circuits Conf.*, 2011, pp. 1–4.
21. Y. Chen et al., "Dadiannao: A Machine-Learning Supercomputer," *Proc. 47th Ann. IEEE/ACM Int'l Symp. Microarchitecture*, 2014, pp. 609–622.
22. D.H. Wolpert, "The Lack of A Priori Distinctions between Learning Algorithms," *Neural Computation*, vol. 8, no. 7, 1996, pp. 1341–1390.
23. D. Liu et al., "PuDianNao: A Polyvalent Machine Learning Accelerator," *Proc. 20th Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, 2015, pp. 369–381.

Tianshi Chen is an associate professor at the State Key Laboratory of Computer Architecture at the Chinese Academy of Sciences. His research interests include computer architecture, parallel processing, and computational intelligence. Chen has a PhD in computer science from University of Science and Technology of China. Contact him at chentianshi@ict.ac.cn.

Zidong Du is a PhD student at the State Key Laboratory of Computer Architecture at the Chinese Academy of Sciences. His research interests include parallel processing, computer architecture, and machine learning. Du has a BSc in electronic engineering from Tsinghua University. Contact him at duzidong@ict.ac.cn.

Ninghui Sun is a professor at the State Key Laboratory of Computer Architecture at the Chinese Academy of Sciences. His research interests include supercomputing, computer architecture, and programming language. Sun has a PhD in computer science from the Institute of Computing Technology at the Chinese Academy of Sciences. Contact him at snh@ict.ac.cn.

Jia Wang is an engineer at the China Merchants Bank. His research interests include parallel processing, computer architecture, and machine learning. Wang has an MSc in computer science from the State Key Laboratory of Computer Architecture at the Chinese Academy of Sciences, where he performed the work for this article. Contact him at brownwj@163.com.

Chengyong Wu is a professor at the State Key Laboratory of Computer Architecture at the Chinese Academy of Sciences. His research interests include programming language and compiler and computer architecture. Wu has a PhD in computer science from the Institute of Computing Technology at the Chinese Academy of Sciences. Contact him at cwu@ict.ac.cn.

Yunji Chen is a professor at the State Key Laboratory of Computer Architecture at the Chinese Academy of Sciences. His research interests include parallel processing, computer architecture, and machine learning. Chen has a PhD in computer science from the Institute of Computing Technology at the Chinese Academy of Sciences. Contact him at cyj@ict.ac.cn.

Olivier Temam is a senior research scientist at INRIA. His research interests include programming language, computer architecture, and machine learning. Temam has a PhD in computer science from University of Rennes. Contact him at olivier.temam@inria.fr.