CNP: AN FPGA-BASED PROCESSOR FOR CONVOLUTIONAL NETWORKS

Clément Farabet[1], *Cyril Poulet*[1], *Jefferson Y. Han*[2], *Yann LeCun*[1]

(1) Courant Institute of Mathematical Sciences, New York University,
715 Broadway, New York, NY 10003, USA
{cfarabet,yann}@cs.nyu.edu

(2) Perceptive Pixel Inc.
111 8th Avenue, New York, NY 10011, USA

ABSTRACT

Convolutional Networks (ConvNets) are biologically-inspired hierarchical architectures that can be trained to perform a variety of detection, recognition and segmentation tasks. ConvNets have a feed-forward architecture consisting of multiple linear convolution filters interspersed with point-wise non-linear squashing functions. This paper presents an efficient implementation of ConvNets on a low-end DSP-oriented Field Programmable Gate Array (FPGA). The implementation exploits the inherent parallelism of ConvNets and takes full advantage of multiple hardware multiply-accumulate units on the FPGA. The entire system uses a single FPGA with an external memory module, and no extra parts. A network compiler software was implemented, which takes a description of a trained ConvNet and compiles it into a sequence of instructions for the ConvNet Processor (CNP). A ConvNet face detection system was implemented and tested. Face detection on a 512×384 frame takes $100ms$ (10 frames per second), which corresponds to an average performance of 3.4×10^9 connections per second for this 340 million connection network. The design can be used for low-power, lightweight embedded vision systems for micro-UAVs and other small robots.

1. INTRODUCTION

Over the next decades, embedded vision systems will become part of an increasing number of applications that include autonomous or semi-autonomous vehicles, consumer electronics (cameras and mobile phones), and toys. Because of their limited payload capacity and energy capacity, toys, micro-UAVs, and small domestic robots cannot use active imaging sensors and must rely on passive cameras and vision for navigation, obstacle avoidance, and object/target recognition. The present work is a step in the direction of low power, lightweight, and low cost vision systems that are required for such applications. We describe an implementation of a complete vision/recognition system on a single low-end Field-Programmable Gate Array (FPGA). The design requires no external hardware, other than a few memory chips, and has been integrated onto a small 7×8 cm printed circuit board, that consumes less than 15W. The system is programmable, and can implement any vision system in which the bulk of the computation is spent on convolutions with small-size kernels. The design is specifically geared towards Convolutional Networks [1, 2], but can be used for many similar architectures based on local filter banks and classifiers, such as HMAX [3, 4], and HoG methods [5].

Convolutional Networks (ConvNets) are feed-forward architectures composed of multiple layers of convolutional filters, interspersed with point-wise non-linear functions [1, 2]. ConvNets are used in several commercial and experimental applications, including video surveillance, OCR [2], face/person detection [6, 7], object recognition [8], and robot navigation [9, 10]. Because they can easily be trained for a wide variety of tasks, ConvNets have many potential applications in micro-robots and other embedded vision systems that require low cost and high-speed implementations.

Pre-trained ConvNets are algorithmically simple, with low requirements for memory bandwidth and arithmetic precision. Hence, several hardware implementations have been proposed in the past. The first one was the ANNA chip, a mixed high-end, analog-digital processor that could compute 64 simultaneous 8×8 convolutions at a peak rate of 4.10^9 multiply-accumulate operations per second [11, 12]. Subsequently, Cloutier et al. proposed an FPGA implementation of ConvNets [13], but fitting it into the limited-capacity FPGAs of the time required the use of extremely low-accuracy arithmetic. Modern DSP-oriented FPGAs include large numbers of hard-wired multiply-accumulate units that can greatly speed up compute-intensive operations, such as convolutions. The system presented in this paper takes full advantage of the highly parallel nature of ConvNet operations, and the high-degree of parallelism provided by modern DSP-oriented FPGAs.

In usual hardware designs, flexibility is left aside to maximize the efficiency of the system. Such designs need to be recompiled for any small change. Instead, the system described here is a *programmable ConvNet Processor*, which can be thought of as a RISC (Reduced Instruction Set Computer) processor, with a vector instruction set that matches the elementary operations of a ConvNet. While these elementary operations are highly optimized at the hardware level, implementing a particular ConvNet simply consists in reprogramming the software layer of our processor, and does not require to reconfigure the logic circuits in the FPGA.

Section 2 describes the architecture for the CNP. Section 3 describes a particular application, based on a standard ConvNet. Finally section 4 gives results on the performance of the system.

2. ARCHITECTURE

Figure 1 shows the functional architecture of the system. The whole system fits in a single FPGA, and requires an external memory module. This design has been implemented with two different platforms: a low-end Xilinx Spartan-3A DSP 3400 FPGA coupled to a DDR-SDRAM module (development kit from Xilinx), and a high-end Xilinx Virtex-4 SX35, coupled to a pair of QDR-SRAM chips (custom design). Both FPGAs have roughly the same density (53,000 logic cells for the Spartan, and 34,000 for the Virtex), the main difference sits in the number of built-in hardware multipliers (126 for the former and 192 for the latter), and the speed at which they can operate (250MHz for the former, and 450MHz for the latter). The other major difference is the bandwidth from/to the external memory: 1GB/s for the development kit, and 7.2GB/s for our custom design. The built-in fixed-point multipliers use 18 bit inputs and accumulate on 48 bits.

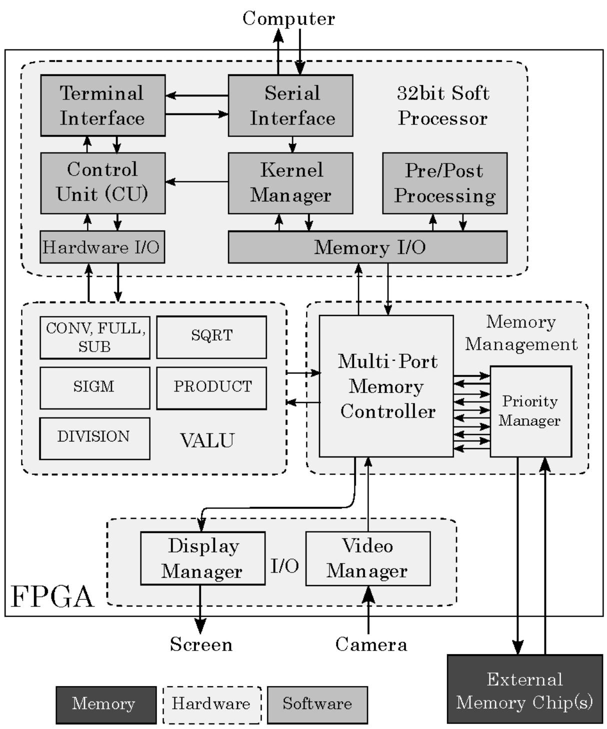

Fig. 1. CNP: architecture.

2.1. Hardware

The CNP contains a Control Unit (CU), a Parallel/Pipelined Vector Arithmetic and Logic Unit (VALU), an I/O control unit, and a memory interface. The CU is actually a full-fledged 32-bit soft CPU, based on the PowerPC architecture, which is used to sequence the operations of the VALU. The VALU implements ConvNet-specific operations including 2D convolutions, spatial pooling/subsampling, point-wise non-linear functions, and other more general vector operators (square root, division, . . .). It has direct memory access, unlike traditional Von Neuman architectures, to avoid long computations from hanging the CPU. The I/O unit comprises two hardware modules: one to acquire video data from a standard DVI[1] port (camera, or other video source), and the other to generate a video signal for display on a standard DVI monitor.

The memory interface is a key part of the system. Its main purpose is to enable parallelization by allowing multiple simultaneous access of the same memory location transparently. A dedicated hardware arbiter has been designed to multiplex/demultiplex access to the external memory chip, by providing 8 ports that can read/write from/to the memory at the same time. Its heuristic is quite simple: it connects a certain port to the external memory and estimates its bandwidth to allocate a certain time slice, then switches to the next port when this computed time slice is reached. This way, different parts of the system can access the external memory simultaneously (e.g. data flows from/to the VALU, camera, display, . . .). To ensure the continuity of data flows on each port, FIFOs are used in both directions. The depth of these FIFOs determine the maximum time slice that can be attributed per port. The output of the memory controller is about 99% when writing, and 90% when reading, and does not depend on the number of ports writing/reading simultaneously.

The second key component is the Vector ALU. All the basic operations of a ConvNet have been implemented at the hardware level, and provided as macro-instructions. These macro-instructions can be executed in any order. Their sequencing is managed at the software level by the soft CPU. This architecture combines the efficiency of hardware with the flexibility of software.

The main hard-wired macro-instructions of this system are: (1) 2D convolution with accumulation of the result, (2) 2D spatial pooling and subsampling, using a max or average filter, (3) dot product between values at identical locations in multiple 2D planes and a vector, and (4) point-wise non-linear mapping (currently an approximation of the hyperbolic tangent sigmoid function). These are higher-level instructions than those of most traditional processors, but provide an optimal framework for running ConvNets. This VALU contains other instructions (division, square root,

[1]Digital Visual Interface

product), that are needed to pre-process images. The entire instruction set is vectorial, and properly pipelined to compute any of these instructions in a linear time to the input size. Other operations required in a complete vision system can be performed on the general purpose soft CPU. We will not go into the details of implementation here, but simply describe the two crucial instructions of the system: the 2D convolution and the sigmoid.

When running a ConvNet, most of the effort goes into 2D convolutions. Therefore the efficiency of the system largely relies on the efficiency of the convolution hardware. Our 2D convolver, shown in Fig. 2, is inspired by Shoup [14], and includes a post accumulation to allow the combination of multiple convolutions. It performs the following basic operation in a single clock cycle:

$$z_{ij} = y_{ij} + \sum_{m=0}^{K-1} \sum_{n=0}^{K-1} x_{i+m,j+n} w_{mn}, \tag{1}$$

where x_{ij} is a value in the input plane, w_{mn} is a value in a $K \times K$ convolution kernel, y_{ij} is a value in a plane to be combined with the result, and z_{ij} is the output plane.

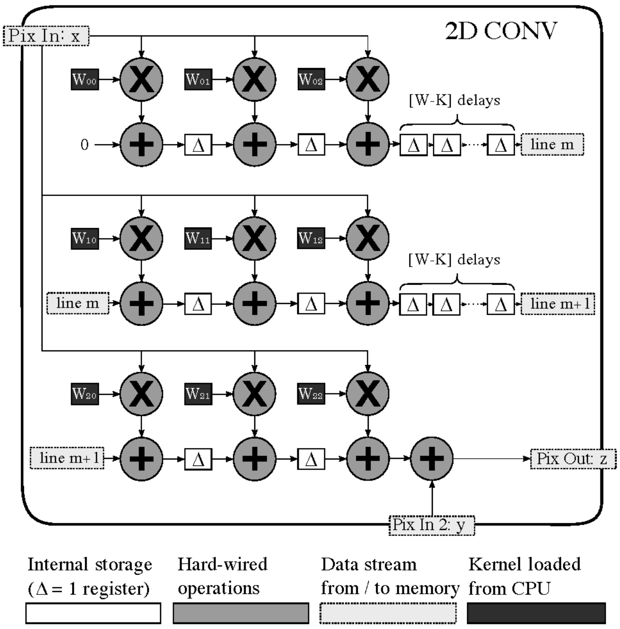

Fig. 2. 2D Convolution for $K = 3$, K = kernel width = kernel height, W = image width.

Values from the input plane are put into K on-chip FIFOs whose size is the width of the image minus the width of the kernel. Shifting values in these FIFOs corresponds to shifting the convolution window over the input plane. At each clock cycle, values are shifted by one, and the dot product between the input plane window and the kernel is computed in parallel. In other words, the convolver performs K^2 multiply-accumulate operations simultaneously (plus the accumulation of the temporary plane Y), at each clock cycle. Consequently, the number of clock cycles for a complete convolution is equal to the number of values in the output plane, plus the latency necessary to fill up the FIFOs (roughly equal to the width of the input plane times the height of the kernel). All arithmetic operations are performed with 16-bit fixed-point precision for the kernel coefficient, and 8-bit for the states. The intermediate accumulated values are stored on 48 bits in the FIFOs.

The low-end FPGA used for this implementation has 126 multiply-accumulate units, hence the maximum square kernel size is 11×11, or two simultaneous kernels of size 7×7, corresponding to a theoretical maximum rate of 24.5×10^9 operations per second at 250MHz. However, our experiments use a single 7×7 convolver because our current application does not require a larger kernel, which corresponds to a theoretical maximum of 12×10^9op/s. A second 7×7 convolver will be added in future designs.

The point-wise non-linearity is implemented as a piecewise approximation of the hyperbolic tangent function $g(x) = A.\tanh(B.x)$. Since the hard-wired multipliers are used by the convolver, the implementation was designed to avoid the use of multiplications, relying exclusively on additions and shifts. The function is approximated by a collection of linear segments for which the binary representation of the slopes a_i has few ones. This allows use to implement the multiplication using a small number of shifts and adds:

$$g(x) = a_i x + b_i \quad for \quad x \in [l_i, l_{i+1}] \tag{2}$$

$$a_i = \frac{1}{2^m} + \frac{1}{2^n} \quad m, n \in [0, 5]. \tag{3}$$

With this constraint, the sigmoid can be computed with two shifts and three adds.

The different instructions of the system have concurrent access to the external memory, allowing them to work asynchronously, given that the available bandwidth is sufficient.

2.2. Software

The soft CPU adds a layer of abstraction to the system: a program on the soft CPU acts as a micro-program for the VALU, allowing a high degree of flexibility. As shown in Fig. 1, different functions run on this processor:

- Memory I/O: a driver to interface the memory controller. It provides access to the external memory: images from the camera and feature maps produced by the ConvNet;
- Post processing operations for object detection applications include non-maximum suppression, calculation of centroids of activities, and other functions that cannot be conveniently implemented in hardware, such as formatting the results of the computation and plotting positions of objects detected on the DVI output;

- Convolution Kernel Manager: stores/retrieves convolution kernels from/to an external memory (flash, or SD Card);
- Serial Interface: provides a means of transferring data from/to an external system (e.g. a host computer);
- Terminal Interface: it provides a set of commands that a user can send from a remote machine to execute, debug or simply obtain results;
- Control Unit & Hardware I/O: the control unit of the CNP is implemented here at the software level. It is used to sequence the instruction flow, and therefore the structure of the ConvNet we want to compute. Having the CU at the software level provides the most flexible environment possible: the whole behaviour of the system can be described by a simple program (in C).

The embedded software also controls external peripherals, such as the camera (e.g. dynamic exposure adjustment), and the video monitor (resolution, color).

Prior to being run on the CNP, a ConvNet must be defined and trained on a conventional machine. Currently available software implementations of ConvNets are available in the `Lush` language (a dialect of Lisp), or as `C++` libraries, such as Torch and EBLearn. Our system is built around the ConvNet training environment distributed as part of the `Lush` system. We wrote a `Lush` compiler that takes the `Lush` description of a trained ConvNet, and automatically compiles it into the proper sequence of calls for the CNP. The result is a compact representation of the network describing the content of each layer—type of operation, matrix of connections, kernels. This representation can then be stored on a flash drive (e.g. SD Card) connected to the CNP. This way, the CNP can run different recognition tasks at the same time.

The lowest layers of abstraction—instruction set, assembly code and programming language—are provided by the FPGA. The highest layers—algorithms and data structures—already exist in classical software implementations. Our Lush-to-CNP compiler unifies these two entities. The description of a ConvNet in `Lush` is rather high-level, a direct translation of the layers and connections shown in Fig. 3 in a Lisp syntax. Kernel coefficients are automatically extracted from the trained network. This insulates users from the intricacies of the CNP architecture. Naturally, users can also program the 32-bit soft CPU directly if necessary.

3. APPLICATION TO FACE DETECTION

To demonstrate the system and to test its performance, a ConvNet face detection system was built and run on the CNP. Face detection systems based on ConvNets have been shown to outperform the popular boosted cascades of Haar wavelets method [15], both in speed and accuracy [6, 7].

3.1. Network Architecture

The ConvNet was built and trained on a conventional computer using the Lush language, and compiled to the CNP using the automatic ConvNet compiler mentioned in the previous section. The architecture of the network is quite similar to those described in [6, 7]. The training architecture of the network is given in table 1. The training images are greyscale images of size 42×42 that have been high-pass filtered by subtracting a Gaussian-filtered version of the image from the image itself. The first layer, called C1, performs 6 convolutions with 7×7 kernels on the input image, producing 6 feature maps of size 36×36. The second layer, S2 performs 2×2 spatial pooling and subsampling of each feature map using a box filter (local averaging without overlap). The third layer, C3, computes high-level features by performing 7×7 convolutions on several S2 feature maps and adding the results. Each of the sixteen C3 feature maps combines different random subsets of S2 feature maps. Layer S4 performs 2×2 pooling and subsampling similarly to S2. The C5 layer performs 6×6 convolutions, combining random subsets of S4 feature maps into 80 different C5 feature maps. Finally, F6 multiplies all feature map values at a single location by a 2×80 matrix. Each feature map in F6 represents a map of activation peaks for each category (face or background). Layer F7 is a fixed, dummy layer that simply combines the face and background outputs into a single score.

Layer	**kernels**	**layer size**
Input image		$1@42 \times 42$
C1 (Conv)	$[6@7 \times 7]$	$6@36 \times 36$
S2 (Pool)	$[6@2 \times 2]$	$6@18 \times 18$
C3 (Conv)	$[61@7 \times 7]$	$16@12 \times 12$
S4 (Pool)	$[16@2 \times 2]$	$16@6 \times 6$
C5 (Conv)	$[305@6 \times 6]$	$80@1 \times 1$
F6 (Dotp)	$[160@1 \times 1]$	$2@1 \times 1$

Table 1. Architecture of the face detector ConvNet.

3.2. Training and Running the ConvNet

The network was trained on a dataset of faces and non-faces according to the method described in [2]. The dataset contained 45,000 images from various sources, of which 30,000 were used for training, and 15,000 for testing. Each set contains 50% faces, and 50% random images (non faces). The face examples include a wide variety of angles, and slight variations of size and position within the window to improve the robustness of the detector. With a fixed detection threshold, the system reaches a roughly 3% equal error rate on this dataset after only 5 training epochs through the training set. After training, the Lush-to-CNP compiler normalizes and

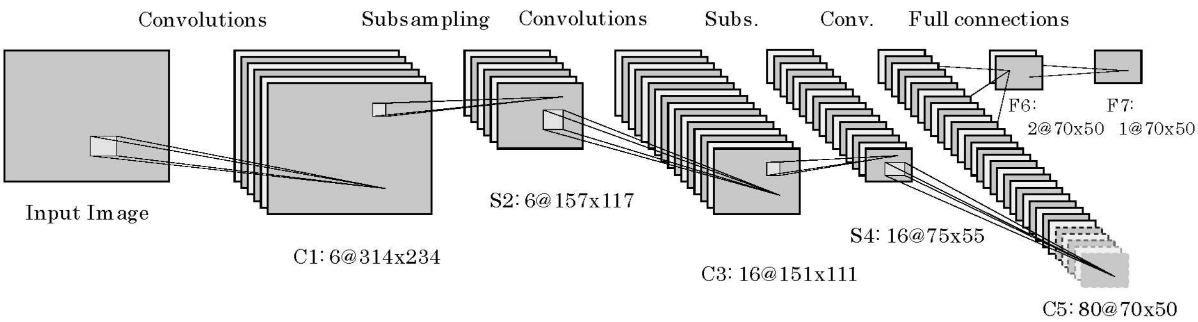

Fig. 3. Architecture of LeNet-5, an example of Convolutional Network.

quantizes the kernel coefficients to 16-bit fixed point representation for transfer to the CNP. The weight quantization did not adversely affect the accuracy of the system in any significant way.

A key advantage of ConvNets is that they can be applied to sliding windows on a large image at very low cost by simply computing convolutions at each layer over the entire image. The output layer is replicated accordingly, producing a detection score for every 42×42 window on the input, spaced every 4 pixels. The overall network is depicted in Fig. 3 for a 320×240 input image.

4. RESULTS

With the Spartan-3A DSP 3400, the design uses about 70% of the general logic (slices), and only 44% of the hardware multipliers. With the Virtex-4 SX35 it uses 90% of the logic, but only 28% of the multipliers. The multipliers are mainly used by the 2D convolver, which requires $7 \times 7 = 49$ hardware multipliers. The size of the kernel could be increased to 14×14 with the Virtex, without affecting the performance.

The system was connected to a simple greyscale camera, and the output was displayed on a monitor using the DVI interface (as shown on Figs. 4).

4.1. Speed

The current design can be run at up to 200MHz in both FPGAs. At this frequency, the peak performance is 9.8 billion connections per second, and approximately 4 billion connections per second on average, when running a realistic network. The difference between these two values is due to the time spent on pre/post processing and data fetching. With these computing resources, processing a full 512×384 greyscale image—using a standard convolutional network containing 530 million connections (as shown in Fig. 3)—takes $100ms$, or 10 frames per second. This is for a scale-invariant detection system that pre-computes a multi-resolution image pyramid as input to the ConvNet. Computing the same ConvNet on a monoscale input reduces this time by a factor of 1.5.

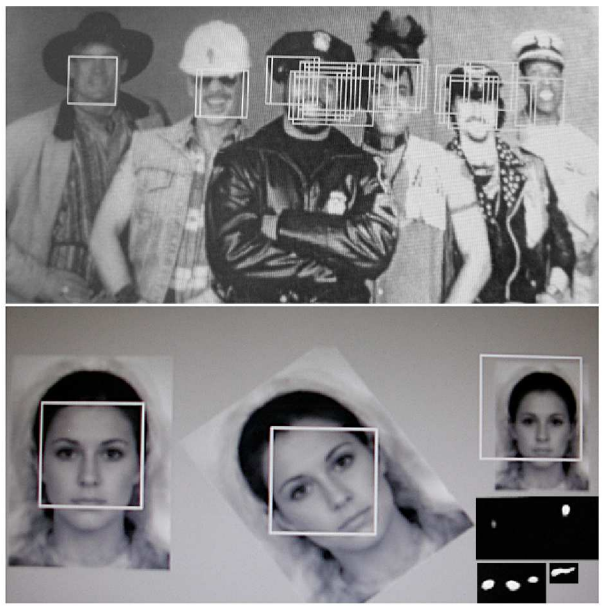

Fig. 4. up: output of the system without non maximum suppression; down: with NMS; bottom corner: output maps for three different scales.

In the current implementation on the Xilinx development board, the bandwidth of the external memory bus is limited (1GB/s). Because of this limitation, this system could only run at 125MHz, which yields a rate of 6 frames per second. Using our custom printed circuit board, the bandwidth is not an issue anymore (7.2GB/s), which allows the design to easily reach 200MHz. In fact, with further optimization, the system will soon use two simultaneous 7×7 convolvers, increasing the frame rate by 2.

5. CONCLUSIONS, FUTURE WORK

This paper presents a self-contained, high performance implementation of Convolutional Networks on a single FPGA. An application of the system to real-time face detection was demonstrated. The system opens the door to intelligent vision capabilities for low-cost robots. Given the compactness, small form factor (see Fig. 5), and low power requirement of the design (15W in peak), a particularly interesting potential application is vision-based navigation for micro-UAVs. While the present system was demonstrated for face detection, ConvNets can be trained to perform a large variety of tasks, including vision-based obstacle avoidance for mobile robots [9, 10].

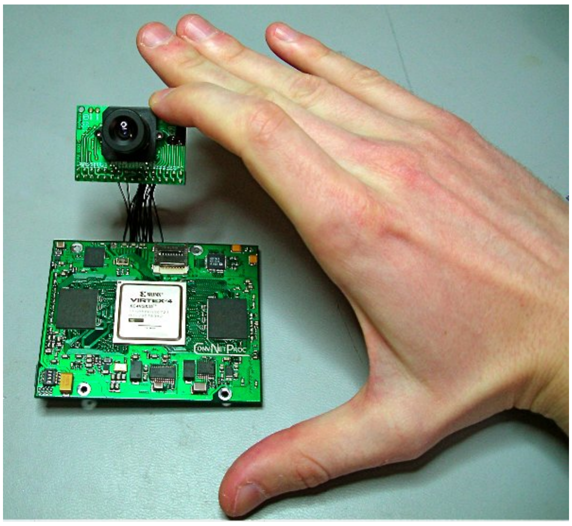

Fig. 5. A custom platform, embedding the FPGA and two QDR memory chips.

It has been demonstrated that our system can be implemented in a low-end FPGA (Xilinx Spartan-3A DSP), and could reach a very interesting performance if set up with a proper bandwidth to an external memory. Our custom board (Fig 5) allows this same design to run at full speed, thanks to the available bandwidth. The FPGA on this custom platform is a higher-end product, and the next step of this work will aim at improving the design to make use of its features: (1) by implementing a second convolver to allow two simultaneous convolutions, (2) by re-organizing the CPU interface to the VALU, to achieve a fully asynchronous system, (3) by allowing the operations in the VALU to be cascaded. All these improvements should increase the overall speed of the system by a factor of 6.

The flexibility of our system will also allow us to integrate new instructions to the VALU, to extend the CNP's capabilities.

6. REFERENCES

[1] Y. LeCun, "Generalization and network design strategies," in *Connectionism in Perspective*, R. Pfeifer, Z. Schreter, F. Fogelman, and L. Steels, Eds. Zurich, Switzerland: Elsevier, 1989, an extended version was published as a technical report of the University of Toronto.

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

[3] T. Serre, L. Wolf, and T. Poggio, "Object recognition with features inspired by visual cortex," in *CVPR*, 2005.

[4] J. Mutch and D. Lowe, "Multiclass object recognition with sparse, localized features," in *CVPR*, 2006.

[5] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *Proc. of Computer Vision and Pattern Recognition*, 2005.

[6] C. Garcia and M. Delakis, "Convolutional face finder: A neural architecture for fast and robust face detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, no. 11, pp. 1408–1423, 2004.

[7] M. Osadchy, Y. LeCun, and M. Miller, "Synergistic face detection and pose estimation with energy-based models," *Journal of Machine Learning Research*, vol. 8, pp. 1197–1215, May 2007.

[8] M. Ranzato, F. Huang, Y. Boureau, and Y. LeCun, "Unsupervised learning of invariant feature hierarchies with applications to object recognition," in *Proc. Computer Vision and Pattern Recognition Conference (CVPR'07)*. IEEE Press, 2007.

[9] Y. LeCun, U. Muller, J. Ben, E. Cosatto, and B. Flepp, "Off-road obstacle avoidance through end-to-end learning," in *Advances in Neural Information Processing Systems (NIPS 2005)*. MIT Press, 2005.

[10] R. Hadsell, A. Erkan, P. Sermanet, J. Ben, K. Kavukcuoglu, U. Muller, and Y. LeCun, "A multi-range vision strategy for autonomous offroad navigation," in *Proc. Robotics and Applications (RA'07)*, 2007.

[11] B. Boser, E. Sackinger, J. Bromley, Y. LeCun, and L. Jackel, "An analog neural network processor with programmable topology," *IEEE Journal of Solid-State Circuits*, vol. 26, no. 12, pp. 2017–2025, December 1991.

[12] E. Säckinger, B. Boser, J. Bromley, Y. LeCun, and L. D. Jackel, "Application of the ANNA neural network chip to high-speed character recognition," *IEEE Transaction on Neural Networks*, vol. 3, no. 2, pp. 498–505, March 1992.

[13] J. Cloutier, E. Cosatto, S. Pigeon, F. Boyer, and P. Y. Simard, "Vip: An fpga-based processor for image processing and neural networks," in *Fifth International Conference on Microelectronics for Neural Networks and Fuzzy Systems (MicroNeuro'96)*, Lausanne, Switzerland, 1996, pp. 330–336.

[14] R. G. Shoup, "Parameterized convolution filtering in a field programmable gate array," in *Selected papers from the Oxford 1993 international workshop on field programmable logic and applications on More FPGAs*. Oxford, United Kingdom: Abingdon EE&CS Books, 1994, pp. 274–280.

[15] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," in *CVPR*, 2001.