An FPGA-Based Stream Processor for Embedded Real-Time Vision with Convolutional Networks

Clément Farabet, Cyril Poulet and Yann LeCun
Courant Institute of Mathematical Sciences, New York University
{cfarabet,yann}@cs.nyu.edu
http://www.cs.nyu.edu/~yann

Abstract

Many recent visual recognition systems can be seen as being composed of multiple layers of convolutional filter banks, interspersed with various types of non-linearities. This includes Convolutional Networks, HMAX-type architectures, as well as systems based on dense SIFT features or Histogram of Gradients. This paper describes a highly-compact and low power embedded system that can run such vision systems at very high speed. A custom board built around a Xilinx Virtex-4 FPGA was built and tested. It measures 70×80 *mm, and the complete system—FPGA, camera, memory chips, flash—consumes 15 watts in peak, and is capable of more than* 4×10^9 *multiply-accumulate operations per second in real vision application. This enables real-time implementations of object detection, object recognition, and vision-based navigation algorithms in small-size robots, micro-UAVs, and hand-held devices. Real-time face detection is demonstrated, with speeds of 10 frames per second at VGA resolution.*

1. Introduction

Vision systems have progressed a lot in the past decade, but most of the modern algorithms still require an amount of computation that makes their integration to autonomous vehicles, cameras or toys impossible. 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 Field-Programmable Gate Array (FPGA). The design requires no external hardware, other than a memory chip, and has been integrated onto a small 70×80 mm printed circuit board, that consumes less than 15W, camera included. 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 [8, 9], but can be used for many similar architectures based on local filter banks and classifiers, such as HMAX [15, 11], and HoG methods [4].

Convolutional Networks (ConvNets) are feed-forward architectures composed of multiple layers of convolutional filters, interspersed with point-wise non-linear functions [8, 9]. Because they can easily be trained for a wide variety of tasks (*e.g.* OCR [9], face/person detection [6, 12], object recognition [13], and robot navigation [10, 7]), 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 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 [1, 14]. Subsequently, Cloutier et al. proposed an FPGA implementation of ConvNets [2], but fitting it into the limited-capacity FPGAs of the time required the use of extremely low-accuracy arithmetic.

Alternatively, pre-trained ConvNets and other convolution-based systems can be implemented on Digital Signal Processors (DSPs), or Graphics Processing Units (GPUs). DSPs are very simple to program, and often result in systems that consume less power than FPGAs, but have lesser capabilities in terms of parallelism. On the other hand, GPUs provide a very flexible environment for parallelism, but consume a lot of power. FPGAs have a clear advantage over these platforms, as they allow the development of custom logic, targeted for precise applications. Although they are not necessarily low-power when compared to equivalent DSPs, they can be seen as a step towards custom chips (*e.g.* ASICs) to achieve very low power.

The system presented in this paper is a *programmable ConvNet Processor (CNP)*, which can be thought of as a SIMD (Single Instruction, Multiple Data) processor, with

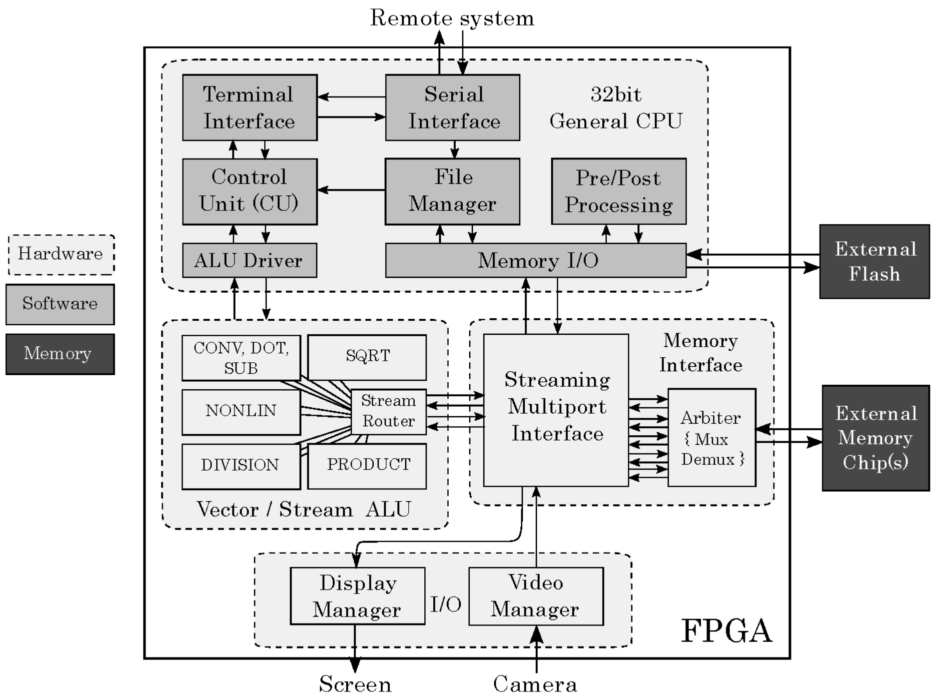

Figure 1. Architecture of the CNP.

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 ConvNet Processor (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. This design has been implemented within a single Xilinx Virtex-4 SX35, coupled to a pair of QDR-SRAM chips, on a custom board. This FPGA is rather small in terms of gates—about 34,000 logic cells—but provides 192 built-in hardware multiply and accumulate units that can operate at up to 450MHz. The bandwidth from/to the external memory is 7.2GB/s in this custom design, but only half of it is currently used. The built-in fixed-point multipliers use 18 bit inputs and accumulate on 48 bits.

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 emulated on 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 (CONV), spatial pooling/subsampling (SUB), point-wise non-linear functions (NONLIN), and other more general vector operators, such as square root (SQRT), division (DIV), ... It has direct memory access (DMA). 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.

Memory Interface

The Memory Interface is a key part of the system. Its first 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. To ensure continuity of data flows, each port is buffered with FIFOs. The depth of these FIFOs determine the maximum time slice that can be attributed per port without interrupting the streams.

The arbiter has a simple heuristic: it cycles through the ports, and connects a port to the external memory if its request queue is not empty. Then it estimates its bandwidth

[1]Digital Visual Interface

based on the quantity of data present in this queue, to allocate a certain time slice. As the arbiter constantly cycles through the ports, the amount of requests in a queue is a fairly good estimate of the bandwidth requirement for a port. Once a queue is fully processed, or if the time slice is over (whatever is shortest), it switches to the next port.

The second purpose of the Memory Interface is to provide an abstract representation of the memory. For that, it uses a streaming interface that can read/write streams from/to the memory. A stream is defined by an offset in memory, strides (to access multi-dimensional data) and sizes for each dimension. For example, a module connected to a port of the streaming interface can simply request a 2D image starting at location X, with dimensions $W \times H$, and the streaming interface will compute the offsets, generate all the addresses, start fetching data, and sets a flag when ready to stream. It will then stream out the data, until the whole chunk has been read out.

It is then easy to build up a system on top of this Memory Interface, as each module can request 2D chunks of data, process them, and simply hand them back. It also allows easy integration of other systems, such as a camera, or display, as each of these can simply read/write to a particular area of the memory. However, there is no hardware checking on the validity of data at some particular location, *e.g.* if a port is writing at some location, no other port should read from this location before the former operation is fully completed.

Vector/Stream ALU

The second key component is the Vector/Stream 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.

The main hard-wired macro-instructions of this system are: (1) 2D convolution with accumulation of the result (CONV), (2) 2D spatial pooling and subsampling (SUB), using a max or average filter, (3) dot product between values at identical locations in multiple 2D planes and a vector (DOT), and (4) point-wise non-linear mapping (NONLIN, 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, 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. More precisely, once the instruction pipeline is filled, one value is computed per clock cycle. Streams are handled by the streaming interface, therefore instructions process streams of similar elements continuously, until the stream stops.

We will not go into the details of implementation here, but simply describe the two most important instructions of the system: the 2D convolution and the sigmoid.

When computing a ConvNet, most of the effort goes into 2D convolutions. Therefore the efficiency of the system relies mainly on the efficiency of the convolution hardware (combined with the efficiency of the streaming interface). Our 2D convolver, shown in Fig. 2, is inspired by Shoup [16], 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.

Values from the input plane are put into K on-chip FIFOs of which the 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, plus a possible subsampling), 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 FPGA used for this implementation has 192 multiply-accumulate units, hence the maximum square kernel size is 13×13, or two simultaneous kernels of size 9×9, corresponding to a theoretical maximum rate of 32×10^9 operations per second at 200MHz. 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 10×10^9op/s at 200MHz. A 13×13 convolver has been successfully synthesized and routed by itself, but we could only go up to 10×10 with the rest of the design (the 32-bit CPU consumes a lot of logic/area).

As noted previously, the convolution engine is also used to perform the subsampling and the dot products.

The point-wise non-linearity is implemented as a piecewise approximation of the hyperbolic tangent function

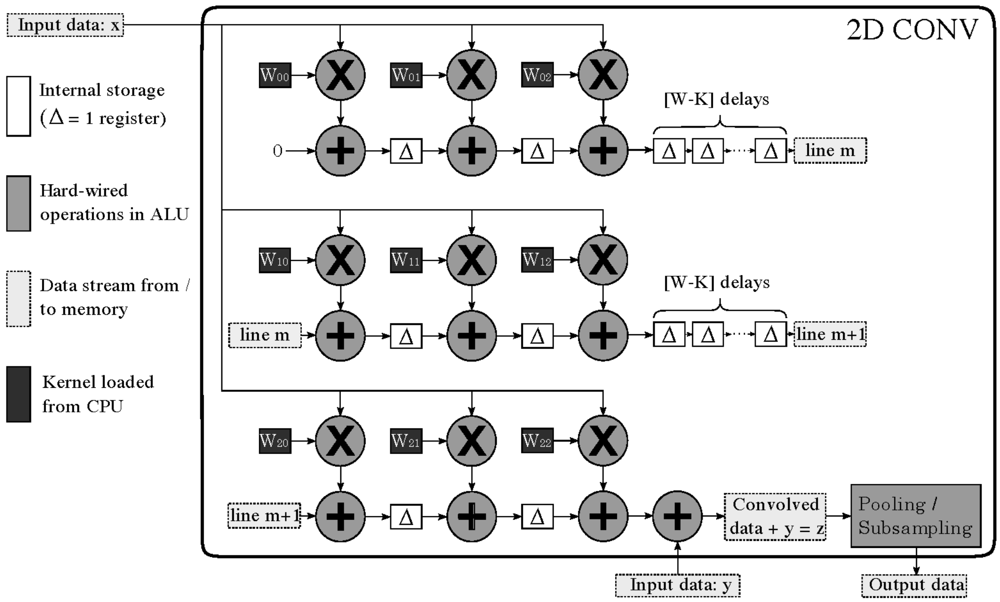

Figure 2. 2D Convolution for $K = 3$, K = kernel width = kernel height, W = image width.

$g(x) = A.\tanh(B.x)$. 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, through the stream interface, allowing them to work asynchronously, given that enough bandwidth is available.

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.

Control Unit

The advantage of using a full-blown CPU instead of a simple state machine is that standard libraries (in `C/C++`) can be easily adapted to run on this platform. In fact, our current Control Unit is a complex embedded `C++` program that reproduces most of the feed-forward part of common machine learning libraries (*e.g.*`Lush, Torch`): modularity, high-level representations of networks, de/serialization of architectures to files, dynamic/hidden memory management, ... The main difference with these libraries is that the computations are not done by the CPU, but off-loaded to the VALU that can read/write data from/to the Streaming Interface asynchronously. The model is similar to the model adopted by GPUs, for which a general purpose CPU handles high-level aspects of the code (what needs to be computed), while the actual computations are executed by the special purpose processor. This model is extremely efficient when the amount of communications between these two entities is small compared to the actual computations.

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/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 a compact representation 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.

The Control Unit running on the soft CPU is then able to decode this representation and dynamically create a representation of the network, with all its layers, tables of connections, and kernels. Once the representation is created the program computes each state of the system in a feed-forward way, from an input image to the output states, by generating the proper sequence of calls to the VALU and Streaming Interface. Memory management is totally abstracted by this program. As a result, the CNP can run different different recognition tasks at the same time and/or easily switch from an application to another at run-time.

Pre/Post Processing

Another advantage of using a 32-bit CPU is the ability of performing less common tasks that are not worth implementing in hardware. To do so, the CPU has to have full read/write access to the same memory used by the VALU/Streaming Interface. This is handled by the Memory I/O driver, which provides high-latency access to the external memory. The code can then have access to all the feature maps and images computed by the VALU, asynchronously (the CPU uses one of the Streaming Interface ports).

Post processing operations for object detection applications include blob detection, non-maximum suppression, computation of centroids of activities, and other functions such as formatting the results of the computation and plotting positions of objects detected on the DVI output, or converting pixel coordinates into real-world zenith/azimuth angles, and so on. All these operations are easily integrated in our `C++` environment.

Other Tasks

As shown in Fig. 1, other functions run on this processor. A File Manager is used to store/retrieve network configurations from/to an external memory (flash, or SD Card). These configurations contain network architectures and pre-trained convolution kernels and other trainable parameters.

A Serial Interface/Terminal Interface, which provide a means of transferring data from/to an external system (*e.g.* a host computer). Once the FPGA is programmed, this is the easiest way of uploading new network configurations.

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

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 [17], both in speed and accuracy [6, 12].

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, 12]. 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: dims [nb]	Maps: dims [nb]
Input image		42×42 [1]
C1 (Conv)	7×7 [6]	36×36 [6]
S2 (Pool)	2×2 [6]	18×18 [6]
C3 (Conv)	7×7 [61]	12×12 [16]
S4 (Pool)	2×2 [16]	6×6 [16]
C5 (Conv)	6×6 [305]	1×1 [80]
F6 (Dotp)	1×1 [160]	1×1 [2]

Table 1. Architecture of the face detector ConvNet. Each layer contains a certain number of feature maps, *e.g.* C1 contains 6 feature maps that are each 36x36. Processing the input image through C1 shrinks the size of these features, because of the convolution kernel.

3.2. Training and Running the ConvNet

The network was trained on a data set of faces and non-faces according to the method described in [9]. The data set 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 data set after only 5 training epochs through the training set. After training, the Lush-to-CNP compiler normalizes and 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

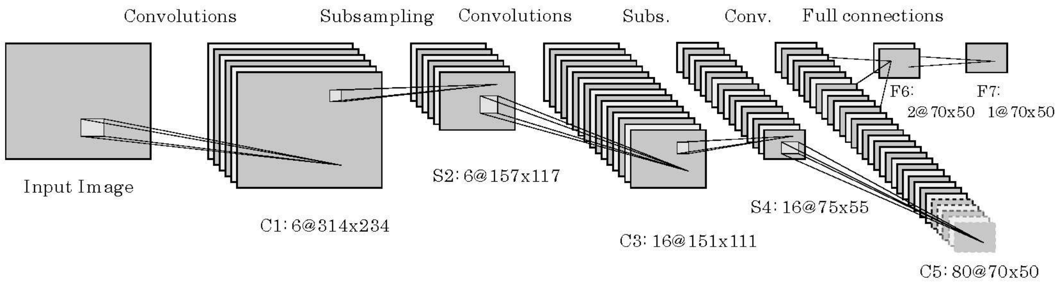

Figure 3. Architecture of LeNet-5, an example of Convolutional Network.

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

The system was connected to a simple greyscale camera, and the output was displayed on a monitor using the DVI interface, as shown on Fig. 4. Fig. 7 also shows captures for an other possible application: object recognition. Fig. 8 shows the complete system, which only requires an external power source.

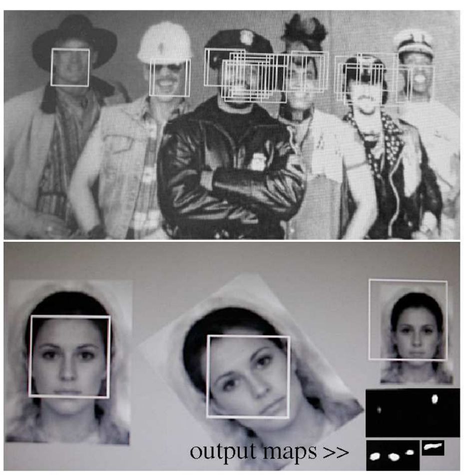

Figure 4. Face detection without (top) and with (bottom) embedded centroid detection. The output maps are shown for three different scales: on the medium scale the three faces are clearly detected. A blob detector is then run on these maps to find the centroids of these activation peaks, and squares are drawn on the input images. All these steps are performed by the soft CPU, and these images are actual captures of the DVI signal generated by the system.

4.1. Usage

The design uses 90% of the logic in the Virtex4, but only 28% of the multipliers. The multipliers are mainly used by the 2D convolver, which requires $7 \times 7 = 49$ hardware multipliers. 60% of the RAM blocks are used, mostly by the Memory Interface. The Virtex4 also provides hardware FIFOs, which are only used at 7% in this design. Table 2 shows the usage of other resources.

The size of the kernel can be easily increased up to a certain point, as the convolution engine mostly requires FIFOs and multipliers. When the router starts packing unrelated logic in the same slices, the process of optimization becomes extremely difficult, and it is hard to predict if a design will actually fit in the device.

Entity	**Number**	**Usage**
I/O Buffers	238 out of 448	53%
Clock Managers	2 out of 8	25%
DSP48s (Mult/Accs)	53 out of 192	27%
FIFO blocks	14 out of 192	7%
RAM Blocks	144 out of 192	75%
Logic Slices	13741 out of 15360	89%
Related	13741 out of 13741	100%
Unrelated	0 out of 13741	0%

Table 2. Device usage for a Virtex-4 SX35.

4.2. Speed

The current design can be run at up to 200MHz. At this frequency, the peak performance is 9.8 billion connections per second, and approximately 4 billion connections per second on average, when computing a realistic network, post-processing included. The difference between these two values is due to the time spent on pre/post processing, stream generations (initiating a stream has a certain latency), and other operations such as sigmoids.

Fig 6 compares raw performances when only computing convolutions. From an actual implementation of 2D convolution on a standard computer (in `C`, using nested for loops) to a convolution computed by our CNP, the speed-up is about 10x for small kernels ($\approx$7x7), and 30x for large kernels ($\approx$17x17).

More convincing is the speed-up attained when computing realistic networks, as shown on Fig 6. In that case, our system is approximately 100 times faster than a standard software implementation (tested), and 12 times faster than an ideal version of that code (theoretically computed, based on the Intel specifications). A realistic software implementation, optimized for a particular CPU would be somewhere in between, still giving our system the advantage of a 30x speed-up.

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 (as shown in Fig. 7). Computing the same ConvNet on a monoscale input reduces this time by a factor of 1.5, or allows full VGA frames (640×480) to be processed at 10fps.

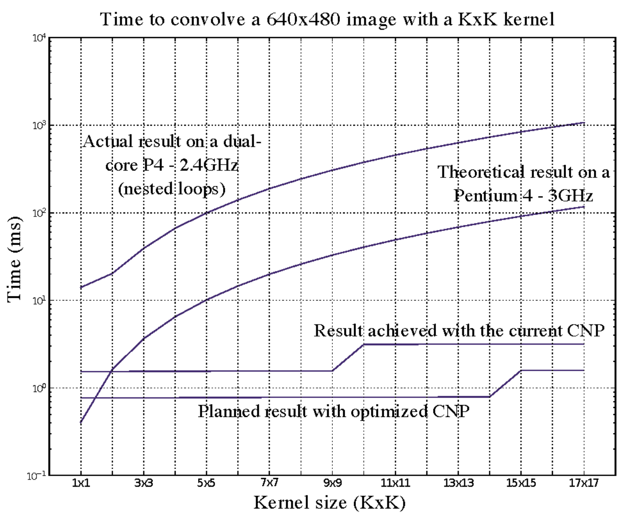

Figure 5. Time required to compute a 2D convolution, on different platforms. The current CNP uses a 7x7 convolver, whereas the optimized CNP would use a 14x14 convolver, and an optimized pre-caching system (as explained in the conclusion).

4.3. Power

An interesting survey was done by Cope [3], to compare raw performances of GPUs, FPGAs and classical CPUs for 2D convolutions.

His results for the FPGA are similar to ours, approximately 30 times faster than a Pentium-4 3.0GHz for a 7x7 kernel, which is what we would get if not using our streaming interface. The fastest GPU he used was an nVidia 6800 Ultra, and he achieved a 10x speed-up for the same kernel size, which is a bit slower than what we get *when* using our streaming interface.

It is then fair to compare our CNP to an this GPU-based system, as they offer similar performances in terms of 2D convolutions. An even better comparison would be a full

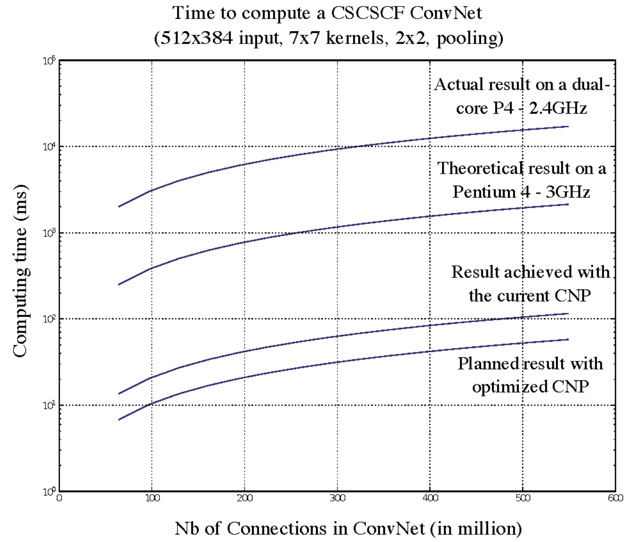

Figure 6. Time required to compute a full ConvNet, on different platforms. The 400 million connection network is the one used for face recognition.

implementation of ConvNets on a GPU, but the authors are not aware of any.

The results are in favor of our CNP. While our full system—camera included—draws 15W in peak computations, an nVidia 6800 PCI board draws more than 70W in average. To use a GPU in embedded systems, one also needs to use a general purpose CPU to interface it, and a camera

Comparisons with standard CPUs is pointless, as those have lesser performances. Low-power DSPs are an alternative, but their performances cannot match those of a GPU or FPGA when kernel sizes are larger than 3x3.

5. Conclusions, Future Work

This paper presents a self-contained, high performance implementation of Convolutional Networks on a single FPGA. The system opens the door to intelligent vision capabilities for low-cost robots. Given the compactness, small form factor (see Fig. 8), and low power requirement of the design (15W in peak, for a complete and autonomous system), 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 [10, 7], or object recognition.

Our system can also be implemented in a low-end FPGA (Xilinx Spartan-3A DSP, as first introduced in [5]) and still reach decent performances.

The next step of this work will aim at improving the design to make full use of the FPGA, and to target ASICs

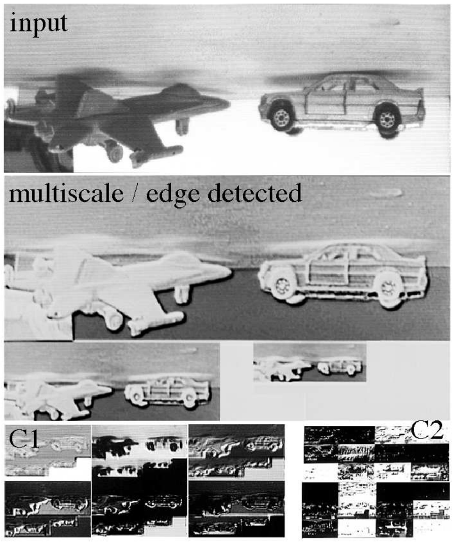

Figure 7. Captures of the video signal generated by the CNP, running an object recognition application. The input is shown first (top), then a pyramid of 3 scales, pre-processed with a Mexican Filter (middle), and a few feature maps from the two first convolutional layers (bottom). These images show that the convolutions and subsampling are applied on all the scale, to minimize the effect of latency in the system

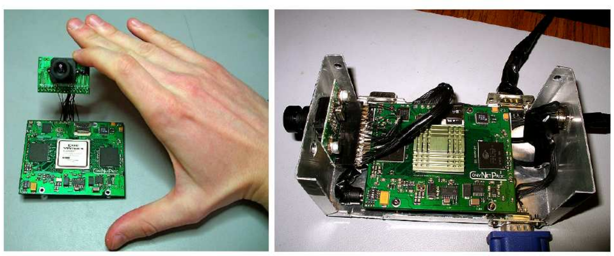

Figure 8. Our custom platform, embedding the FPGA and two QDR memory chips. The complete recognition only draws 15W in peak.

to reduce the power consumption. Our current efforts aim at: (1) transforming the convolver into a flexible and programmable grid of elementary units (*e.g.* multipliers, non-linear mappings, ...), to compute several convolutions of different sizes at the same time, (2) allowing the operations in the VALU to be cascaded (creating different paths in this grid) to reduce latencies, (3) pre-caching the kernels in the grid, and pre-fetching streams of data while processing.

References

[1] 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*, 26(12):2017–2025, December 1991. 1

[2] 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)*, pages 330–336, Lausanne, Switzerland, 1996. 1

[3] B. Cope. Implementation of 2d convolution on fpga, gpu and cpu. Technical report, Department of Electrical and Electronic Engineering, Imperial College, London, UK, 2006. 7

[4] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In *Proc. of Computer Vision and Pattern Recognition*, 2005. 1

[5] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun. Cnp: An fpga-based processor for convolutional networks. In *International Conference on Field Programmable Logic and Applications*, Prague, September 2009. IEEE. 7

[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*, 26(11):1408–1423, 2004. 1, 5

[7] 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. 1, 7

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

[9] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998. 1, 5

[10] 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. 1, 7

[11] J. Mutch and D. Lowe. Multiclass object recognition with sparse, localized features. In *CVPR*, 2006. 1

[12] M. Osadchy, Y. LeCun, and M. Miller. Synergistic face detection and pose estimation with energy-based models. *Journal of Machine Learning Research*, 8:1197–1215, May 2007. 1, 5

[13] 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. 1

[14] 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*, 3(2):498–505, March 1992. 1

[15] T. Serre, L. Wolf, and T. Poggio. Object recognition with features inspired by visual cortex. In *CVPR*, 2005. 1

[16] 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*, pages 274–280, Oxford, United Kingdom, 1994. Abingdon EE&CS Books. 3

[17] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. In *CVPR*, 2001. 5