Invited Paper

NeuFlow: A Runtime Reconfigurable Dataflow Processor for Vision

Clément Farabet[1,2] Berin Martini[2] Benoit Corda[1] Polina Akselrod[2]
Eugenio Culurciello[2] Yann LeCun[1]
[1] Courant Institute of Mathematical Sciences, New York University, New York, NY, USA
[2] Electrical Engineering Department, Yale University, New Haven, CT, USA
http://www.neuflow.org

Abstract

In this paper we present a scalable dataflow hardware architecture optimized for the computation of general-purpose vision algorithms—neuFlow—and a dataflow compiler—luaFlow—that transforms high-level flow-graph representations of these algorithms into machine code for neuFlow. This system was designed with the goal of providing real-time detection, categorization and localization of objects in complex scenes, while consuming 10 Watts when implemented on a Xilinx Virtex 6 FPGA platform, or about ten times less than a laptop computer, and producing speedups of up to 100 times in real-world applications. We present an application of the system on street scene analysis, segmenting 20 categories on 500×375 *frames at 12 frames per second on our custom hardware neuFlow.*

1. Introduction

Computer vision is the task of extracting high-level information from raw images. Generic, or general-purpose synthetic vision systems have for ultimate goal the elaboration of a model that captures the relationships between high-dimensional data (images, videos) into a low-dimensional decision space, where arbitrary information can be retrieved easily, *e.g.* with simple linear classifiers or nearest neighbor techniques. The exploration of such models has been an active field of research for the past decades, ranging from fully trainable models—such as convolutional networks—to hand-tuned models—HMAX-type architectures, as well as systems based on dense SIFT (Scale-Invariant Feature Transform) or HoG (Histograms of Gradients).

Many successful object recognition systems use dense features extracted on regularly-spaced patches over the input image. The majority of the feature extraction systems have a common structure composed of a filter bank (generally based on oriented edge detectors or 2D gabor functions), a non-linear operation (quantization, winner-take-all, sparsification, normalization, and/or point-wise saturation) and finally a pooling operation (max, average or histogramming). For example, the scale-invariant feature transform (SIFT [23]) operator applies oriented edge filters to a small patch and determines the dominant orientation through a winner-take-all operation. Finally, the resulting sparse vectors are added (pooled) over a larger patch to form local orientation histograms. Some recognition systems use a single stage of feature extractors [19, 7, 25]. Other models like HMAX-type models [27, 24] and convolutional networks use two or more layers of successive feature extractors.

This paper presents a scalable hardware architecture for large-scale multi-layered synthetic vision systems based on large parallel filter banks, such as convolutional networks—*neuFlow*—and a dataflow compiler—*luaFlow*—that transforms a high-level flow-graph representation of an algorithm into machine code for neuFlow. This system is a dataflow vision engine that can perform real-time detection, recognition and localization in mega-pixel images processed as pipelined streams. The system was designed with the goal of providing real-time detection, categorization and localization of objects in complex scenes, while consuming ten times less than a laptop computer—on the order of 10W for an FPGA implementation—and producing speedups of up to 100 times in end-to-end applications, such as the street scene parser presented in section 3.

Graphics Processing Units (GPUs) are becoming a common alternative to custom hardware in vision applications, as demonstrated in [4]. Their advantage over custom hardware are numerous: they are inexpensive, available in most recent computers, and easily programmable with standard development kits. The main reasons for continuing developing custom hardware are twofold: performance and power consumption. By developing a custom architecture that is fully adapted to a certain range of tasks (as is shown in this paper), the product of power consumption by performance can be improved by two orders of magnitude (100x).

Other groups are currently working on custom architectures for convolutional networks or similar algorithms: NEC Labs [2], Stanford [16], Kaist [15].

Section 2 describes neuFlow's architecture. Section 3 describes a particular application, based on a standard convolutional network. Section 4 gives results on the performance of the system. Section 5 concludes.

2. Architecture

Hierarchical visual models, and more generally image processing algorithms are usually expressed as sequences, trees or graphs of transformations. They can be well described by a modular approach, in which each module processes an input image or video collection and produces a new collection. Figure 4 is a graphical illustration of this approach. Each module requires the previous bank to be fully (or at least partially) available before computing its output. This causality prevents simple parallelism to be implemented across modules. However parallelism can easily be introduced within a module, and at several levels, depending on the kind of underlying operations.

2.1. A Dataflow Grid

First dataflow architectures were introduced by [1], and quickly became an active field of research [8, 14, 18]. [3] presents one of the latest dataflow architectures that shares several similarities to the approach presented here: while both architectures rely on a grid of compute tiles, which communicate via FIFOs, the grid presented here also provides a runtime configuration bus, which allows efficient runtime reconfiguration of the hardware (as opposed to static, offline synthesis).

Figure 1 shows a dataflow architecture that we designed to process homogeneous streams of data in parallel [9]. It is defined around several key ideas:

- a 2D grid of N_{PT} Processing Tiles (PTs) that contain: 1- a bank of processing operators. An operator can be anything from a FIFO to an arithmetic operator, or even a combination of arithmetic operators. The operators are connected to local data lines, 2- a routing multiplexer (MUX). The MUX connects the local data lines to global data lines or to the 4 neighboring tiles.

- a Smart Direct Memory Access module (Smart DMA), that interfaces off-chip memory and provides asynchronous data transfers, with priority management,

- a set of N_{global} global data lines used to connect PTs to the Smart DMA, $N_{global} << N_{PT}$; and local data lines used to connect PTs with their 4 neighbors,

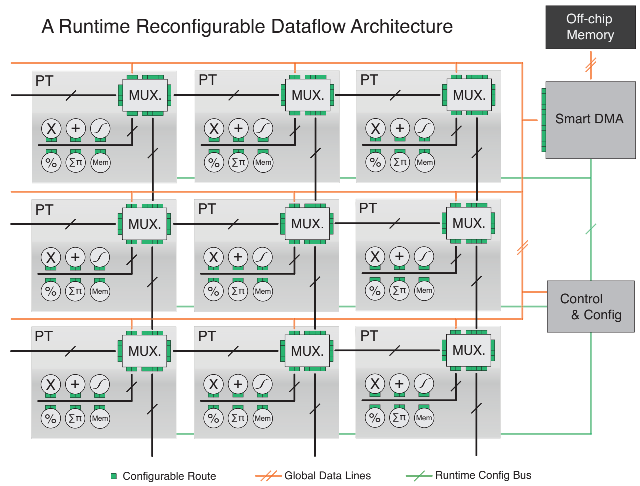

Figure 1. A dataflow computer. A set of runtime configurable processing tiles are connected on a 2D grid. They can exchange data with their 4 neighbors and with an off-chip memory via global lines. Configurable elements are depicted as squares.

- a Runtime Configuration Bus, used to reconfigure many aspects of the grid at runtime—connections, operators, Smart DMA modes…(the configurable elements are depicted as squares on Fig.1),

- a controller that can reconfigure most of the computing grid and the Smart DMA at runtime.

2.2. On Runtime Reconfiguration

One of the most interesting aspects of this grid is its configuration capabilities. Many systems have been proposed which are based on two-dimensional arrays of processing elements interconnected by a routing fabric that is reconfigurable. Field Programmable Gate Arrays (FPGAs) for instance, offer one of the most versatile grid of processing elements. Each of these processing elements—usually a simple look-up table—can be connected to any of the other elements of the grid, which provides with the most generic routing fabric one can think of. Due to the simplicity of the processing elements, the number that can be packed in a single package is in the order of 10^4 to 10^5. The drawback is the reconfiguration time, which takes in the order of milliseconds, and the synthesis time, which takes in the order of minutes to hours depending on the complexity of the circuit.

At the other end of the spectrum, recent multicore processors implement only a few powerful processing elements (in the order of 10s to 100s). For these architectures, no synthesis is involved, instead, extensions to existing programming languages are used to explicitly describe parallelism. The advantage of these architectures is the relative simplicity of use: the implementation of an algorithm rarely takes more than a few days, whereas months are required for a typical circuit synthesis for FPGAs.

The architecture presented here is in the middle of this spectrum. Building a fully generic dataflow computer is a tedious task. Reducing the range of applications to the computation of visual models, vision systems and image processing pipelines allows us to define the following constraints:

- high throughput is a top priority, low latency is not. Indeed, most of the operations performed on images are replicated over both dimensions of these images, usually bringing the amount of similar computations to a number that is much larger than the typical latencies of a pipelined processing unit

- therefore each operator has to provide with maximal throughput (*e.g*. one operation per clock cycle) to the detriment of any initial latency, and has to be stallable (*e.g*. must handle discontinuities in data streams)

- configuration time has to be low, or more precisely in the order of the system's latency. This constraint simply states that the system should be able to reconfigure itself between two kinds of operations in a time that is negligible compared to the time needed to perform one such operation. That is a crucial point to allow runtime reconfiguration

- the processing elements in the grid should be as coarse grained as permitted, to maximize the ratio between *computing logic* and *routing logic*. Creating a grid for a particular application (*e.g*. convolutional networks) allows the use of very coarse operators. On the other hand, a general purpose grid has to cover the space of standard numeric operators

The first two points of this list are crucial to create a flexible dataflow system. Several types of grids have been proposed in the past [8, 14, 17], often trying to solve the dual latency/throughput problem, and often providing a computing fabric that is too rigid.

The grid proposed here provides a flexible processing framework, due to the stallable nature of the operators. Indeed, any path can be configured on the grid, even paths that require more bandwidth that is actually feasible. Instead of breaking, each operator will stall its pipeline when required. This is achieved by the use of FIFOs at the input and output of each operators, that compensate for bubbles in the data streams, and force the operators to stall when they are full. Any sequence of operators can then be easily created, without concern for bandwidth issues.

The third point is achieved by the use of a runtime configuration bus, common to all units. Each module in the design has a set of configurable parameters, routes or settings (depicted as squares on Figure 1), and possesses a unique address on the network. Groups of similar modules also share a broadcast address, which dramatically speeds up reconfiguration of elements that need to perform similar tasks.

A typical execution of an operation on this system is the following: (1) the control unit configures each tile to be used for the computation and each connection between the tiles and their neighbors and/or the global lines, by sending a configuration command to each of them, via the Runtime Configuration Bus, (2) it configures the Smart DMA to prefetch the data to be processed, and to be ready to write results back to off-chip memory, (3) when the DMA is ready, it triggers the streaming out, (4) each tile processes its respective incoming streaming data, and passes the results to another tile, or back to the Smart DMA, (5) the control unit is notified of the end of operations when the Smart DMA has completed.

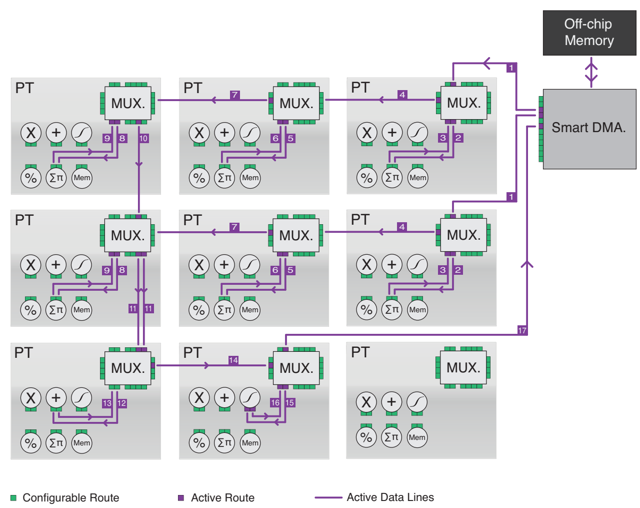

Figure 2. The grid is configured for a complex computation that involves several tiles: the 3 top tiles perform a 3×3 convolution, the 3 intermediate tiles another 3×3 convolution, the bottom left tile sums these two convolutions, and the bottom centre tile applies a function to the result.

Such a grid can be used to perform arbitrary computations on streams of data, from plain unary operations to complex nested operations. As stated above, operators can be easily cascaded and connected across tiles, independently managing their flow by the use of input/output FIFOs.

Figure 2 shows an example of configuration, where the grid is configured to compute a sum of two convolutions followed by a non-linear activation function:

$$y_{1,i,j} = Tanh(\sum_{m=0}^{K-1}\sum_{n=0}^{K-1} x_{1,i+m,j+n}w_{1,m,n} + \sum_{m=0}^{K-1}\sum_{n=0}^{K-1} x_{2,i+m,j+n}w_{2,m,n}). \quad (1)$$

The operator $\sum\prod$ performs a sum of products, or a dot-product between an incoming stream and a local set of weights (preloaded as a stream). Therefore each tile performs a 1D convolution, and 3 tiles are used to compute a 2D convolution with a 3×3 kernel. In Figure 2 all the paths are simplified, and in some cases one line represents multiple parallel streams.

2.3. Optimizing for FPGAs

Recent DSP-oriented FPGAs include a large number of hard-wired MAC units and several thousands of programmable cells (lookup tables), which allow fast prototyping and real-time simulation of circuits, but also actual implementations to be used in final products.

In this section we present a concrete implementation of the ideas presented in section 2.1, specially tailored for filter-based algorithms, and optimized for typical modern FPGAs (*e.g.* using DSP slices, large block-rams, ...). The architecture presented here has been fully coded in hardware description languages (HDL) to target both ASIC synthesis and programmable hardware like FPGAs.

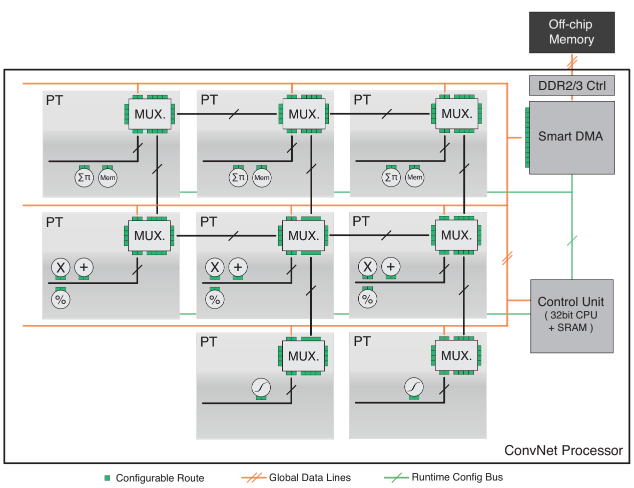

Figure 3. Optimizing the grid for filter-based systems. A grid of multiple full-custom Processing Tiles, and a fast streaming memory interface (Smart DMA).

2.3.1 Optimized Processing Tiles

The *PTs* are independent processing tiles laid out on a two-dimensional grid. As presented in section 2.1, they contain a routing multiplexer (MUX) and local operators. Compared to the general purpose architecture proposed above, this implementation is specialized for applications that rely heavily on two-dimensional convolutions (for convolutional networks, 80% to 90% of the computations are spent in filtering). Figure 3 shows the specialization:

- the top row PTs only implement Multiply and Accumulate (MAC) arrays ($\sum\prod$ operators), which can be used as 2D convolvers (implemented in the FPGA by dedicated hardwired MACs). It can also perform on-the-fly subsampling (spatial pooling), and simple dot-products (linear classifiers) [10]
- the middle row PTs contain general purpose operators, such as squaring and dividing for divisive normalization, and other mathematical operators
- the bottom row PTs implement non-linear mapping engines, used to compute all sorts of functions from $Tanh()$ to $Sqrt()$ or $Abs()$. Those can be used at various places, from normalization to non-linear activation units

The operators in the PTs are fully pipelined to produce one result per clock cycle. Image pixels are stored in off-chip memory as Q8.8 (16bit, fixed-point), transported on global lines as Q8.8 and scaled to 32bit integers within operators, to keep full precision between successive operations. The numeric precision, and hence the size of a pixel, will be noted P_{bits}.

The 2D convolver can be viewed as a dataflow grid itself, with the only difference that the connections between the operators (the MACs) are fixed. The reason for having a 2D convolver within a tile (instead of a 1D convolver per tile, or even simply one MAC per tile) is that it maximizes the ratio between computing logic and routing logic, as stated previously. This is less flexible, as the choice of the array size is a hardwired parameter, but it is a reasonable choice for an FPGA implementation, and for image processing in general. For an ASIC implementation, having a 1D dot-product operator per tile is probably the best compromise.

The pipelined implementation of this 2D convolver was previously described in [10]. Both the kernel and the image are streams loaded from the memory, and the filter kernels can be pre-loaded in local caches concurrently to another operation. Each pixel streaming in the convolver triggers $K \times K$ parallel operations, when applying $K \times K$ filters.

All the non-linearities in neural networks can be computed with the use of look-up tables or piece-wise linear decompositions. A loop-up table associates one output value for each input value, and therefore requires as much memory as the range of possible inputs. This is one of the fastest method to compute a non-linear mapping, but the time required to reload a new table is prohibitive if different mappings are to be computed with the same hardware.

A piece-wise linear decomposition is not as accurate (f is approximated by g, as in Eq. 2), but only requires a couple of coefficients a_i to represent a simple mapping such as a hyperbolic tangent, or a square root. It can be reprogrammed very quickly at runtime, allowing multiple mappings to reuse the same hardware. Moreover, if the coefficients a_i follow the constraint given by Eq. 3, the hardware can be reduced to shifters and adders only.

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

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

2.3.2 Smart DMA Implementation

A critical part of this architecture is the Direct Memory Access (DMA) module. Our *Smart DMA* module is a full custom engine that has been designed to allow N_{DMA} ports to access the external memory totally asynchronously.

A dedicated arbiter is used as hardware *Memory Interface* to multiplex and demultiplex access to the external memory with high bandwidth. Subsequent buffers on each port insure continuity of service on a port while the others are utilized.

The DMA is *smart*, because it complements the Control Unit. Each port of the DMA can be configured to read or write a particular chunk of data, with an optional stride (for 2D streams), and communicate its status to the Control Unit. Although this might seem trivial, it respects one of the foundations of dataflow computing: while the Control Unit configures the grid and the DMA ports for each operation, an operation is driven exclusively by the data, from its fetching, to its writing back to off-chip memory.

If the PTs are synchronous to the memory bus clock, the following relationship can be established between the memory bandwidth B_{EXT}, the number of possible parallel data transfers $MAX(N_{DMA})$ and the bits per pixel P_{bits}:

$$MAX(N_{DMA}) = \frac{B_{EXT}}{P_{bits}}. \tag{4}$$

For example $P_{bits} = 16$ and $B_{EXT} = 128bit/cyc$ allows $MAX(N_{DMA}) = 7$ simultaneous transfers.

2.4. LuaFlow: a Compiler for neuFlow

Prior to being run on neuFlow, a given algorithm has to be converted to a representation that can be interpreted by the Control Unit to generate controls/configurations for the system. For that purpose a compiler and dataflow API—*luaFlow* [1]—were created. LuaFlow is a full-blown compiler that takes sequential, tree-like or flow-graph descriptions of algorithms in the Torch5 [5] environment, and parses them to extract different levels of parallelism. Pattern matching is used to map known sequences of operations to low-level, pre-optimized routines. Other unknown operators are mapped in less optimized ways. Once each high-level module has been associated with a set of low-level operations, a static sequence of grid configurations, interspersed with DMA transfers is produced, and dumped as binary code for the embedded Control Unit.

Extensive research has been done on the question of how to schedule dataflow computations [22], and how to represent streams and computations on streams [18]. The problem can be formulated simply: given a particular graph-type description of an algorithm, and given a particular implementation of the dataflow grid, what is the sequence of grid configurations that yield the shortest computation time?

There are three levels at which computations can be parallelized:

- across modules: operators can be cascaded, and multiple modules can be computed on the fly (average speedup),
- across images, within a module: can be done if multiple instances of the required operator exist (poor speedup, as each independent operation requires its own input/output streams, which are limited by the bandwidth to external memory B_{EXT}),
- within an image: some operators naturally implement that (the 2D convolver, which performs all the MACs in parallel), in some cases, multiple tiles can be used to parallelize computations.

Parallelizing computations across modules can be done in special cases. For example, linear operations (convolutions) are often followed by non-linear mappings in neural networks: these two operators (each belonging to a separate module) can be easily cascaded on the grid. This simple optimization speeds up the computation by a factor of 2.

Parallelizing computations across images is straightforward, and done massively by luaFlow. Here is an example that illustrates that point: given a dataflow grid built with 4 PTs with 2D convolvers, 4 PTs with standard operators, and 4 PTs with non-linear mappers, we want to compute a fully-connected filter-bank with 4 inputs and 8 outputs, *e.g.* a filer bank where each of the 8 outputs is a sum of 4 inputs, each convolved with a different kernel:

$$y_j = \sum_{i=0}^{3} k_{ij} * x_i \quad for \quad j \in [0, 7]. \tag{5}$$

For the given hardware, the optimal mapping is: each of the four 2D convolvers is configured to convolve one of the three inputs x_i with a kernel k_{ij}, and a standard PT is configured to accumulate those 4 streams into one and produce y_j.

Parallelizing computations within images is what this grid is best at: this is the simplest form of parallelism, where locality in memory is exploited to the maximum.

3. Application to Street Scene Understanding

Several applications were implemented on neuFlow: from a simple face detector to a pixel-wise obstacle clas-

[1] http://www.neuflow.org/category/xlearn/

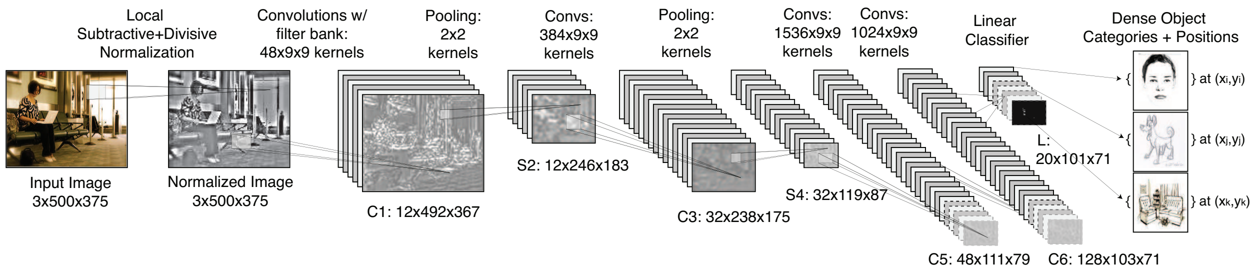

Figure 4. A convolutional network for street scene parsing.

sifier [6] and a complete street scene parser, as shown on Figure 5. Other example applications can be found at www.neuflow.org.

In this section we focus on the elaboration, training and implementation of a complete street-scene parser. This work extends and is strongly inspired by previous work from Grangier *et al*. [12]. Scene parsing aims at segmenting and recognizing the content of a scene: from objects to large structures—roads, sky, buildings, cars, etc. In other words, the goal is to map each pixel of a given input image to a unique label.

Figure 5. Street scene parsing: a convolutional network was trained on the LabelMe spanish dataset [26] with a method similar to [12]. The training set only contains photos from spanish cities; the image above is a picture taken in Edinburgh. The convolutional network is fully computed on neuFlow, achieving a speedup of about 100x (500x375 images are processed in 83ms, as opposed to 8s on a laptop).

Grangier *et al*. [12] showed that using a deep convolutional network with a greedy layer-wise learning (up to 6 convolutional layers) could yield significantly better results than simpler 2 or 3-layer systems. We followed a slightly different method, favoring larger kernels over deeper networks, but kept the idea of incrementally growing the network's capacity.

A subset of the LabelMe dataset [26], containing about 3000 images of spanish cities [2], was used to train this convolutional network. We removed 10% of the set to be used for validation (testing). The twenty most occurring classes were extracted, and the goal was set to minimize the pixel classification error on those classes.

All the images were first resized to 500×375, then 400 million patches were randomly sampled to produce a $20 \times 1e8 \times N \times N$ tensor where the first dimension indexes the classes, the second indexes patches of which the center pixel belongs to the corresponding class, and the last two dimensions are the height and width of the patch.

The training was done in 3 phases. First: we started with a simple model, CN_1 (table 1), similar to the one originally proposed in [20]. The model has small kernels (5×5) and 3 convolutional layers only. This first model was trained to optimize the pixel-wise cross entropy (negative log-likelihood) through stochastic gradient descent over the training set. Minimizing the cross entropy (rather than the mean-square error) helps promote the categories of rare appearance. Small kernels, and a few layers allowed the system to see 10 million training patches in a couple of hours, and converge to a reasonable error fairly quickly. With these parameters, the receptive field of the network is 32×32, which only represents 0.55% of the complete field of view;

Second: all the convolutional kernels were then increased to 9×9, by padding the extra weights with zeros: CN_2 (table 2). This increased the receptive field to 60×60 (about 2% of the image), with the interesting property that at time 0 of this second training phase, the network was producing the same predictions than with the smaller kernels;

Third: a fourth layer was added—a.k.a. greedy layer-wise learning—which increased the receptive field to 92×92 (5% of the image). This required dropping the previous linear classifier, and replace it with a new—randomly initialized—larger classifier.

Performances were evaluated on a separate test set, which was created using a subset (10%) of the original dataset. Results are shown on Table 4.

Once trained, the network was passed over to luaFlow, and transparently mapped to neuFlow. A key advantage of convolutional networks 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

[2] http://people.csail.mit.edu/torralba/benchmarks/

Layer	Kernels: dims [nb]	Maps: dims [nb]
Input image		32×32 [3]
N0 (Norm)		32×32 [3]
C1 (Conv)	5×5 [48]	28×28 [12]
P2 (Pool)	2×2 [1]	14×14 [12]
C3 (Conv)	5×5 [384]	10×10 [32]
P4 (Pool)	2×2 [1]	5×5 [32]
C5 (Conv)	5×5 [1536]	1×1 [48]
L (Linear)	1×1 [960]	1×1 [20]

Table 1. CN_1: base model. N: Local Normalization layer (note: only the Y channel is normalized, U and V are untouched); C: convolutional layer; P: pooling (max) layer; L: linear classifier.

Layer	Kernels: dims [nb]	Maps: dims [nb]
Input image		60×60 [3]
N0 (Norm)		60×60 [3]
C1 (Conv)	9×9 [48]	52×52 [12]
P2 (Pool)	2×2 [1]	26×26 [12]
C3 (Conv)	9×9 [384]	18×18 [32]
P4 (Pool)	2×2 [1]	9×9 [32]
C5 (Conv)	9×9 [1536]	1×1 [48]
L (Linear)	1×1 [960]	1×1 [20]

Table 2. CN_2: second model. Filters are increased, which doubles the receptive field

Layer	Kernels: dims [nb]	Maps: dims [nb]
Input image		92×92 [3]
N0 (Norm)		92×92 [3]
C1 (Conv)	9×9 [48]	84×84 [12]
P2 (Pool)	2×2 [1]	42×42 [12]
C3 (Conv)	9×9 [384]	34×34 [32]
P4 (Pool)	2×2 [1]	17×17 [32]
C5 (Conv)	9×9 [1536]	9×9 [48]
C6 (Conv)	9×9 [1024]	1×1 [128]
L (Linear)	1×1 [960]	1×1 [20]

Table 3. CN_3: a fourth convolutional layer C6 is added, which, again, increases the receptive field. Note: C6 has sparse connectivity (*e.g.* each of its 128 outputs is connected to 8 inputs only, yielding 1024 kernels instead of 6144).

output layer is replicated accordingly, producing one detection score for every 92×92 window on the input, spaced every 4 pixels. The overall network is depicted in Fig. 4 for a 500×375 input image. Producing the prediction on one image of that size takes about 8 seconds on a laptop-class Intel DuoCore 2.66GHz processor; the same prediction is produced in $83ms$ on neuFlow, with an average error of 10^{-2} (quantization noise).

Model	CN_1	CN_2	CN_3
CN Error (%)	29.75	26.13	24.26
CN+MST Error (%)	27.17	24.40	23.39

Table 4. Percentage of mislabeled pixels on validation set. CN Error is the pixelwise error obtained when using the simplest pixelwise winner, predicted by the ConvNet. CN+MST Error is the pixelwise error obtained by histogramming the ConvNet's prediction into connected components (the components are obtained by computing the minimum spanning tree of an edge-weighted graph built on the raw RGB image, and merging its nodes using a surface criterion, in the spirit of [11]).

4. Performance Comparisons

Table 5 reports a performance comparison for the computation of a typical filter bank operation on multiple platforms: 1- the CPU data was measured from compiled C code (GNU C compiler and Blas libraries) on a Core 2 Duo 2.66GHz Apple Macbook PRO laptop operating at 90W (30W for the CPU); 2- the FPGA data was measured on a Xilinx Virtex-6 VLX240T operating at 200MHz and 10W (power consumption was measured on the board) ; 3- the GPU data was obtained from a CUDA-based implementation running on a laptop-range nVidia GT335m operating at 1GHz and 30W and on a nVidia GTX480 operating at 1GHz and 220W; 4- the ASIC data is simulation data gathered from an IBM 45nm CMOS process ($5 \times 5mm$). For an ASIC-based design with a speed of 400MHz, the projected power consumption, using post-synthesis data and standard analysis tools is estimated at 5W.

	CPU	V6	mGPU	IBM	GPU
Peak GOPs	10	160	182	1280	1350
Real GOPs	1.1	147	54	1164	294
Power W	30	10	30	5	220
GOPs/W	0.04	14.7	1.8	230	1.34

Table 5. Performance comparison. 1- CPU: Intel DuoCore, 2.7GHz, optimized C code, 2- V6: neuFlow on Xilinx Virtex 6 FPGA—on board power and GOPs measurements; 3- IBM: neuFlow on IBM 45nm process: simulated results, the design was fully placed and routed; 4- mGPU/GPU: two GPU implementations, a low power GT335m and a high-end GTX480.

The current design was proven at 200MHz on a Xilinx Virtex 6 ML605 platform, using four 10×10 convolver grids. At this frequency, the peak performance is 80 billion connections per second, or 160 GMACs. Sustained performances for typical applications (such as the street scene parser) range from 60 to 120 GMACs, sustained.

5. Conclusions

We presented a dataflow architecture that was optimized for the computation of hierarchical filter-bank based visual models. Different use cases were studied, and it was seen that compiling convolutional networks was straight-forward on such an architecture, thanks to their relatively uniform design.

Because of their applicability to a wide range of tasks, convolutional networks are perfect candidates for hardware implementations, and embedded applications, as demonstrated by the increasing amount of work in this area. We expect to see many new embedded vision systems based on convolutional networks in the next few years.

A real-time (12 frames per second) street scene parser was developed, trained, and implemented on neuFlow. Satisfying results are reported on the standard LabelMe dataset.

Multiple object detection [21] or online learning for adaptive robot guidance [13] are tasks that are currently being developed around neuFlow, using and extending the convolutional network framework.

References

[1] D. A. Adams. *A computation model with data flow sequencing*. PhD thesis, Stanford, CA, USA, 1969. 110

[2] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi. A dynamically configurable coprocessor for convolutional neural networks. In *Proceedings of the 37th annual international symposium on Computer architecture*, pages 247–257. ACM, 2010. 110

[3] M. H. Cho, C. chi Cheng, M. Kinsy, G. E. Suh, and S. Devadas. Diastolic arrays: Throughput-driven reconfigurable computing, 2008. 110

[4] A. Coates, P. Baumstarck, Q. Le, and A. Ng. Scalable learning for object detection with gpu hardware. In *Proceedings of the 2009 IEEE/RSJ international conference on Intelligent robots and systems*, pages 4287–4293. Citeseer, 2009. 109

[5] R. Collobert. Torch. presented at the Workshop on Machine Learning Open Source Software, NIPS, 2008. 113

[6] B. Corda, C. Farabet, M. Scoffier, and Y. LeCun. Building heterogeneous platforms for end-to-end online learning based on dataflow computing design. Dec 2010. 114

[7] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In *CVPR*, 2005. 109

[8] J. B. Dennis and D. P. Misunas. A preliminary architecture for a basic data-flow processor. *SIGARCH Comput. Archit. News*, 3(4):126–132, 1974. 110, 111

[9] C. Farabet, B. Martini, P. Akselrod, S. Talay, Y. LeCun, and E. Culurciello. Hardware accelerated convolutional neural networks for synthetic vision systems. In *International Symposium on Circuits and Systems (ISCAS'10)*, Paris, May 2010. IEEE. 110

[10] 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 (FPL'09)*, Prague, September 2009. IEEE. 112

[11] P. Felzenszwalb and D. Huttenlocher. Efficient graph-based image segmentation. *International Journal of Computer Vision*, 59(2), Sep 2004. 115

[12] D. Grangier, L. Bottou, and R. Collobert. Deep convolutional networks for scene parsing. ICML 2009 Deep Learning Workshop, June 2009. 114

[13] R. Hadsell, P. Sermanet, M. Scoffier, A. Erkan, K. Kavackuoglu, U. Muller, and Y. LeCun. Learning long-range vision for autonomous off-road driving. *Journal of Field Robotics*, 26(2):120–144, February 2009. 116

[14] J. Hicks, D. Chiou, B. S. Ang, and Arvind. Performance studies of id on the monsoon dataflow system, 1993. 110, 111

[15] J.-Y. Kim, M. Kim, S. Lee, J. Oh, K. Kim, and H.-J. Yoo. A 201.4 GOPS 496 mW Real-Time Multi-Object Recognition Processor With Bio-Inspired Neural Perception Engine. *Solid-State Circuits, IEEE Journal of*, 45(1):32 –45, jan. 2010. 110

[16] S. Kim, L. McAfee, P. McMahon, and K. Olukotun. A highly scalable restricted boltzmann machine fpga implementation. *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 367–372, 2009. 110

[17] H. T. Kung. Why systolic architectures? pages 300–309, 1986. 111

[18] J. l. Gaudiot, L. Bic, J. Dennis, and J. B. Dennis. Stream data types for signal processing. In *In Advances in Dataflow Architecture and Multithreading*. IEEE Computer Society Press, 1994. 110, 113

[19] S. Lazebnik, C. Schmid, and J. Ponce. Beyond bags of features: Spatial pyramid matching for recognizing natural scene categories. In *Proc. of Computer Vision and Pattern Recognition*, pages 2169–2178. IEEE, June 2006. 109

[20] 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. 114

[21] Y. LeCun, F.-J. Huang, and L. Bottou. Learning methods for generic object recognition with invariance to pose and lighting. In *Proceedings of CVPR'04*. IEEE Press, 2004. 116

[22] E. A. Lee and David. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, 36:24–35, 1987. 113

[23] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 2004. 109

[24] J. Mutch and D. G. Lowe. Multiclass object recognition with sparse, localized features. In *CVPR*, 2006. 109

[25] N. Pinto, D. D. Cox, and J. J. DiCarlo. Why is real-world visual object recognition hard? *PLoS Comput Biol*, 4(1):e27, 01 2008. 109

[26] B. Russell, A. Torralba, K. Murphy, and W. T. Freeman. Labelme: a database and web-based tool for image annotation. *International Journal of Computer Vision*, 2007. 114

[27] T. Serre, L. Wolf, and T. Poggio. Object recognition with features inspired by visual cortex. In *CVPR*, 2005. 109