John von Neumann Institute for Computing

Novel Brain-Derived Algorithms Scale Linearly with Number of Processing Elements

Jeff Furlong, Andrew Felch,
Jayram Moorkanikara Nageswaran, Nikil Dutt,
Alex Nicolau, Alex Veidenbaum, Ashok Chandrashekar,
Richard Granger

published in

Parallel Computing: Architectures, Algorithms and Applications ,
C. Bischof, M. Bücker, P. Gibbon, G.R. Joubert, T. Lippert, B. Mohr,
F. Peters (Eds.),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **38**, ISBN 978-3-9810843-4-4, pp. 767-776, 2007.
Reprinted in: Advances in Parallel Computing, Volume **15**,
ISSN 0927-5452, ISBN 978-1-58603-796-3 (IOS Press), 2008.

http://www.fz-juelich.de/nic-series/volume38

Novel Brain-Derived Algorithms Scale Linearly with Number of Processing Elements

Jeff Furlong[1], Andrew Felch[2], Jayram Moorkanikara Nageswaran[1], Nikil Dutt[1], Alex Nicolau[1], Alex Veidenbaum[1], Ashok Chandrashekar[2], and Richard Granger[2]

[1] University of California, Irvine
Irvine, CA 92697, USA
E-mail: {jfurlong, jmoorkan, dutt, nicolau, alexv}@ics.uci.edu

[2] Dartmouth College
Hanover, NH 03755, USA
E-mail: {andrew.felch, ashok.chandrashekar, richard.granger}@dartmouth.edu

Algorithms are often sought whose speed increases as processing elements are added, yet attempts at such parallelization typically result in little speedup, due to serial dependencies intrinsic to many algorithms. A novel class of algorithms have been developed that exhibit intrinsic parallelism, so that when processing elements are added to increase their speed, little or no diminishing returns are produced, enabling linear scaling under appropriate conditions, such as when flexible or custom hardware is added. The algorithms are derived from the brain circuitry of visual processing[10,17,8,9,7]. Given the brain's ability to outperform computers on a range of visual and auditory tasks, these algorithms have been studied in attempts to imitate the successes of real brain circuits. These algorithms are slow on serial architectures, but as might be expected of algorithms derived from highly parallel brain architectures, their lack of internal serial dependencies makes them highly suitable for efficient implementation across multiple processing elements. Here, we describe a specific instance of an algorithm derived from brain circuitry, and its implementation in FPGAs. We show that the use of FPGAs instead of general-purpose processing elements enables significant improvements in speed and power. A single high end Xilinx Virtex 4 FPGA using parallel resources attains more than a 62x performance improvement and 2500x performance-per-watt improvement. Multiple FPGAs in parallel achieve significantly higher performance improvements, and in particular these improvements exhibit desirable scaling properties, increasing linearly with the number of processing elements added. Since linear scaling renders these solutions applicable to arbitrarily large applications, the findings may provide a new class of novel approaches for many domains, such as embedded computing and robotics, that require compact, low-power, fast processing elements.

A Introduction

Brain architecture and computer architecture have adapted to two very different worlds of computation and costs. While computers use connected central processors with central memory storage, the mammalian brain evolved low-precision processing units (neurons) with local memory (synapses) and very sparse connections. From a typical engineering view, brains have a demonstrably inferior computing fabric, yet they still outperform computers in a broad range of tasks including visual and auditory recognition. We propose that these brain circuit components are designed and organized into specific brain circuit architectures, that perform atypical but quite understandable algorithms, conferring unexpectedly powerful functions to the resulting composed circuits.

In this paper we first present the components of a visual brain circuit architecture (VBCA), and an overview of visual object recognition. We then show a parallel algo-

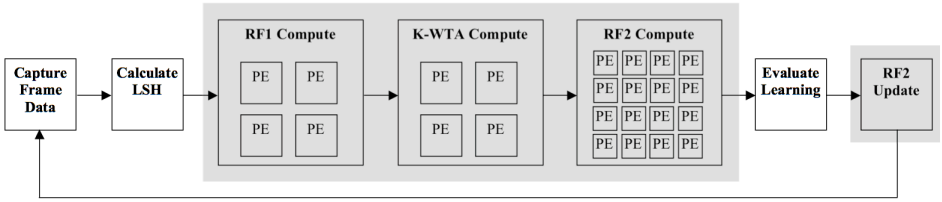

Figure 1. The components of our system are shown. The shaded blocks are targeted for parallelization on FPGAs.

rithm derived for the VBCA, and we demonstrate their application to a particular visual recognition benchmark of known difficulty (the "Wiry Object Recognition Database"[5] from CMU). We characterize the inefficiencies of mapping this intrinsically parallel algorithm onto general purpose CPUs, and then describe our implementation in FPGAs, and we analyze the resulting findings.

A.1 Background

Neurons in the eye are activated by light, and send information electrically to the thalamo-cortical system, which is the focus of our work. Thalamo-cortical circuits, constituting more than 70% of the human brain, are primarily responsible for all sensory processing as well as higher perceptual and cognitive processing.

It has long been noted that the brain operates hierarchically[20,21]: downstream regions receive input from upstream regions and in turn send feedback, forming extensive cortico-cortical loops. Proceeding downstream, neural responsiveness gains increasing complexity in types of shapes and constructs recognized, as well as becoming independent of the exact location or size of the object (translation and scale invariance)[3]. Early visual components have been shown to respond to simple constructs such as spots, lines, and corners,[13]; simulations of these capabilities have been well-studied in machine vision. Further downstream regions are selectively activated in response to assemblies of multiple features, organizing the overall system into a hierarchy[12,14,22]. Our work shows simulations of presumed intermediate stages in these cortical hierarchies, selectively responding to particular feature assemblies composed of multiple line segments. In particular we present processors of three segments ("line triples"). This highly simplified architecture is shown to be very effective on difficult visual applications such as the CMU database. It is hoped that implementations of further downstream areas will extend the work to more abstract perceptual and cognitive processing[17,9]

Key components in the implemented system are briefly described here. A given set of features activates particular regions that we refer to loosely as receptive fields (RF). Any two shapes are as similar as the number of activated neurons shared in their activation patterns. As a result of sparse population codes, most neurons are inactive; this concept is represented in a highly simplified form as sparse bit-vectors. The intrinsic random connectivity tends to select or recruit some areas to respond to some input features (RFs). Neurons train via increments to their synaptic connections, solidifying their connection-based preferences. After a simulated "developmental" phase, synapses are either present or absent and each neuron's level of activation can be represented as the bit vector.

Neurons activate local inhibitory cells which in turn de-activate their neighbors; the resulting competition among neurons is often modeled as the K best (most activated) "winners" take all or K-WTA[6,17,8,9], which our model incorporates.

These K winners activate a next set of neurons, termed RF2. As objects are viewed, these RF2 neurons are synaptically trained, becoming "recognizers" of classes of objects. RF2 activation can in turn be used in "top-down" or feedback processing, to affect the RF1 detectors based on what the RF2 cells "think" is being recognized.

The model described here uses 8192 neurons to represent the first set of input feature detectors (RF1) and 1024 neurons for RF2; other configurations exhibit comparable behaviour. Intuitively, increasing the number of neurons can be used to increase the number of classes of objects that can be recognized. The architecture modeled is depicted schematically in Fig. 1. The shaded regions are those that are targets for FPGA implementation.

A.2 Application

The Wiry Object Recognition Database (WORD)[5] from CMU was used to provide a difficult dataset dependent on shape-based recognition. The database contains a series of videos, in which a barstool is placed in several different office environments. The goal is to determine where in the videos the barstool is located. Successful identification requires a bounding box to enclose the location of the stool in the video frame, within 25% of the stool's actual area.

In top-down processing, thresholds are used to convert the activations of the second set of neurons, those for classes of objects, from data on expected shapes to data on actual shapes. This conversion builds recognition confidence, and confidence above another threshold indicates an actual guess of a recognized visual object. The guesses create bounding boxes that correspond to a particular area of the field of view.

It is important to note that the VBCA system *learns*, i.e., improves and generalizes its performance with experience. Instead of merely recognizing objects based on predetermined criteria, our system acquires information about expected and actual shape locations. The system's ability to recognize many different objects of the same type, such as variants of stools, increases as it views and learns more objects.

A.3 Related Work

Several models of the neocortex have been proposed previously. Some models contain no hierarchical structure, but are accelerated by hardware, even FPGAs[23,19]. Others do contain hierarchy, but not hardware acceleration[16,15]. Some have simulated hierarchical neocortical structures, and suggested that implementations of their models in FPGA fabrics would achieve better results, but have not actually done so and do not report object recognition results[18]. A previous implementation of the K-WTA network on a FPGA has yielded speeds similar to those of a desktop computer[11]. No similar work has shown hierarchical cortical models that operate in real-time.

B VBCA Algorithm

The entire VBCA algorithm is currently executed on two systems. We use a general purpose CPU to capture inputs and perform high-level computations, while the predominantly

```
Initialize:
   load RF1 with 8192 entries of size 120 bits (ROM)
   load RF2 with 1024 entries of size 8192 bits
(RAM)
Capture Video Frame:
   extract 400 to 600 line segments (32 bits each)
   group 20,000 to 100,000 line triples (30 bits each)
depending on scene complexity
Calculate LSH:
   let lshVector be a locality sensitive hash (LSH)
based on frame and line data
RF1 Compute:
//Computes the dot product of RF1 neurons with a
//120 bit vector based on input frame data;
//Additionally computes a threshold so that about 512
//RF1 neurons are active
   popRAM[] = array of 8192 elements of 7 bits each
   sums[] = array of 121 elements of 13 bits each
   for i=1…8192
      for j=1…20
         popCount = popCount + Max(0, 8 -
Abs(lshVector - RF1[i]))
         lshVector = lshVector >> 6
         RF1[i] = RF1[i] >> 6
      popRAM[i] = popCount
      sums[popCount] = sums[popCount] + 1
K-WTA Compute:
//Computes a vector indicating what RF1 neurons are
//receptive
   i=120
   while totalSum < 512 || i != 0
      totalSum = totalSum + sums[i--]
   threshold = i
   threshold2 = popCount / 4
   for i=1…8192
      if popRAM[i] > threshold
         midVector[i] = 1
      else
         midVector[k] = 0
RF2 Compute:
//Computes dot products between the resultant
//midVector and all RF2 neurons;  Outputs the RF2
//index if the population count of the dot prodcut is
//over threshold2
   popCountROM[] = array of 2^8192 elements of 13
bits each
   //(ROM can be distributed to reduce size)
   for i=1…1024
      popCount = popCountROM[midVector &
RF2[i]]
      if popCount > currentMax
         currentMax = popCount
      if popCount >= threshold2
         output index i
   output threshold2
   output currentMax
Evaluate Learning:
   input results from RF2 Compute
   analyze highly receptive neurons to determine if
they should learn
   modify RF2 neurons (to be similar to frame data)
based on learning
RF2 Update:
   if applicable, send updates to RF2 as a result of
learning
```

Figure 2. Detailed pseudocode for the components of the Visual Brain Circuit Architecture (VBCA) algorithm.

bottom-up or feed-forward computations, exhibiting highly parallel operation, can be sent to FGPAs. Final post-processing is also performed on the general purpose CPU. The optimized VBCA algorithm is shown in Fig. 2, which represents in more detail all of the components of Fig. 1.

As can be seen in the visual brain circuit algorithm, some sections contain a massive amount of potential parallelism, while others sections may contain sequential code or offer very little parallelism. For example, the process of capturing frame data is very well suited for general purpose CPUs. Line segment extraction has been researched extensively and well developed algorithms already exist, so we have not attempted to perform this calculation on FPGAs. While we have implemented the *Calculate LSH* code on general purpose CPUs and FPGAs, we have found it more efficient on CPUs because the computations are small, but with notable memory requirements. With optimizations to software code, it may be possible to execute these setup tasks in real-time on a general purpose CPU.

The sections termed *RF1 Compute*, *K-WTA Compute*, and *RF2 Compute* offer high degrees of parallelism, just like the human brain, and we target these operations on FPGAs. These three sections are the components of bottom-up or feed-forward computations. To run these calculations in real-time with a reaction time of one second, we need to demon-

strate a speedup of 185x over that of optimized CPU code, if we analyze 100,000 line segment triples per frame.

After processing line triples in the *RF2 Compute* section, the results can be streamed back to the CPU, where top-down learning occurs in the section *Evaluate Learning*. For simplicity, we also call this section post-processing.

Though we have a software implementation of how we believe this post processing component works, it is likely to be optimized in the future. These optimizations, some of which may include mapping the computations to more FPGAs, may offer speedups to satisfy overall real-time constraints. However, an evaluation of these post-processing calculations is only practical when the previous computations run in real-time.

C FPGA Implementation

Here we detail the parallel feed-forward computations and show that the their FPGA implementation is more computationally efficient than that on general purpose CPUs.

Consider the *RF1 Compute* section of the algorithm. This code could contain 8192 parallel processing elements, each with 20 parallel subcomponents. These subcomponents can be small, six bit functional units. However, a classic data convergence problem exists, because each of the 20 partial sums must be added together.

By building small FPGA logic elements that can be replicated, we have introduced just four parallel processing elements, each with 20 subcomponents. These small calculations are wasteful on a general purpose CPU because not all 64 bits of its datapath can be utilized.

The *K-WTA Compute* loop is a very simple comparison loop iterated 8192 times. Again, 8192 processing elements could be used in the optimal case. These elements can be small blocks of logic because the necessary comparison is only on seven bits of data. After each iteration, only one bit of output is generated. A fast 64 bit processor is unlikely to show great utilization because the datapath is simply too big. In our implementation, we have again built four small parallel functional units.

The most complex part of the algorithm is that of the *RF2 Compute* logic. The 8192 bit dot product is performed by ANDing a data value (midVector) with one of 1024 elements in the RF2 array. The number of binary ones in this resultant dot product must be counted (termed "population count"), and compared to a variable threshold. We have found that the most efficient method for the population count is to store population count values in a lookup table (ROM). Simply, the data to be counted is used as an address to the ROM, and the data stored at that address is the actual population count. Because it is not feasible to store all 2^{8192} elements in one ROM, we add parallelism by distributing the data.

In the optimal case, 1024 processing elements could compute all of the RF2 calculations in approximately one cycle. However, this is not realistic because the required logic and routing resources is far too great.

On a general purpose CPU, the simple dot product operation can take a significant amount of cycles, because the data must be partitioned to manageable 64 bit widths. However, on FPGAs, we can utilize 256 bit processing elements, and partition our population count ROM into blocks of eight bits each. More specifically, we utilize 32 parallel lookup ROMs and replicate them 16 times each. By doing so, we are able to compute 4096 bits of dot product per cycle, instead of just 64 bits on a CPU cycle. The use of parallelism on the loop and within each functional unit in that loop reduces the number of cycles from

millions on a general purpose CPU, to just thousands on an FPGA.

We have carefully chosen the level of parallelism to be used on each of the three sections of the algorithm. By doing so in the method presented, each section requires about the same number of cycles to finish its computation. Hence, we can pipeline each section, to produce a data streaming system, increasing the speed by about a factor of three.

Our implementation has been verified in post place and route simulation models. Because of the extremely limited number of Xilinx Virtex 4 FX140 devices at the time of this writing, we have been unable to implement our design in a physical FPGA. Our final performance enhancements can be seen in Table 1, which also compares the results with that of a general purpose CPU. We have demonstrated a 62.7x speedup on a single FPGA, while proving a speedup per $1000 factor that is significantly better than that of CPUs. In addition, the speedup per watt on FPGAs is a 2500x improvement over that of general-purpose CPUs.

Much of our circuit design is dedicated to the *RF2 Compute section*. The amount of logic and memory required is greatest for this part of the algorithm. Overall, our model requires approximately 9.6 Mbit of on-chip memory, limiting our design to only a few specific FPGAs that contain that much BRAM. Table 2 shows our FPGA utilization for the entire design, including a basic I/O interface.

Device	Cycles	Freq. (MHz)	Time (us)	Speedup	Cost ($)	Speedup / $1K	Power (W)	Speedup / W
Intel Core2 Duo	5423430	2930	1851	1.00x	2000	0.500	65.0	0.0154
XC4VFX140	2242	76.00	29.50	62.7x	10000	6.27	1.63	38.5

Table 1. The Xilinx Virtex 4 FPGA has a much longer cycle time, but ultimately requires less computational time. It also has better speedup per $1000 and speedup per watt ratios.

Number of RAMB16s	548	out of	552	99%
Number of Slices	38896	out of	63168	61%
Number of DSP48s	0	out of	192	0%
Number of PowerPCs	0	out of	2	0%

Table 2. The amount of BRAM required for our design is very large, while logic resources to support the BRAM are also considerable.

C.1 Parallel FPGAs

If we consider all optimal cases of the three VBCA components, where logic resources are unlimited, we could complete all required computations in as little as four clock cycles. Of these cycles, two are required for internal pipelining, and two are required for data computation. The required clock period for such massively parallel small processing elements would be great. However, by dividing our ideal unlimited logic resources between parallel FPGAs, we can address the intangibility problem and the long clock period problem.

Utilizing parallelism across multiple FPGAs, instead of purely within a single FPGA, is possible because we have built a data streaming design where our FPGA calculations

do not share data between each other. The only data that must be replicated among all FPGAs are the RF1 elements, the RF2 elements, and the ROM based lookup tables for RF2. Hence, we can create a client/server model, where the general purpose CPU sends lshVectors to each FPGA. which can then return results to the CPU for post processing.

Under ideal assumptions, by simply extrapolating our results with a single FPGA and ignoring I/O demands, we find that using 2048 parallel FPGAs is optimum. This scenario exploits all potential feed-forward parallelism, and requires just four pipelined clock cycles for a feed-forward calculation. The associated computational time is merely 52.63 ns, a speedup of over 35000x under perfect conditions.

However, creating such a client/server system requires a detailed analysis of the bandwidth requirements of multiple FGPAs, which can limit overall system feasibility.

The input bandwidth for the FPGA is quite small. Our pipelined calculations start with a 120 bit lshVector, which must be sent every 29.50 us to sustain the pipeline. That data corresponds to a rate of 4.07 Mbps.

The output bandwidth depends upon how many RF2 activations occur. Because the K-WTA algorithm has a K that is dependent upon the RF1 activations, the number of activations can be between 0 and 1024. In the case where 1024 activations occur, the FPGA must send 10 bits per activation, over a 29.50 us period of time. This data yields a rate of 347.1 Mbps.

Our targeted Xilinx Virtex 4 FX140 FPGA is, at the time of this writing, available on development boards with three distinct I/O interfaces. First, the PCI-X 133 interface is supported, which supports a maximum shared bandwidth of about 8.5 Gbps[1]. Because we must connect these PCI-X 133 interfaces indirectly to our CPU, we cannot utilize all of this bandwidth. Today's CPU motherboards only support about three PCI-X 133 interfaces, limiting the amount of FPGAs in our parallel system. Even with just three parallel FPGAs, our optimal speedup improves from 62.7x to 188.1x, just above our 185x speedup needed.

The second interface supported by the development boards is PCIe x1/x2/x4/x8[2]. The PCIe interfaces support rates of 2.0, 4.0, 8.0, and 16.0 Gbps for each direction, respectively for x1, x2, x4, and x8 lanes. Again, however, current CPU motherboards only support about four PCIe slots, limiting parallelism to four FPGAs. With this parallelism, our optimal speedup increases to 250.8x, well above our 185x requirement.

Finally, the third supported interface is Fibre Channel[2]. This interface supports up to about 2.0 Gbps on the FPGA side. However, these connections can be linked to a 4.0 Gbps Fibre Channel switch, which can connect to our server CPU via a 4.0 Gbps Fibre Channel interface. Hence, the number of interfaces is no longer our limiting factor, but the amount of bandwidth needed. Because our output bandwidth only requires about 347 Mbps, we can use up to 11 parallel FPGAs on 4.0 Gbps of system bandwidth. At this rate, our optimal speedup jumps to 689.7x, which would be fast enough to process additional frames of data and/or decrease the system's reaction time.

Because development boards with multiple Xilinx Virtex 4 FX140 FPGAs do not currently exist, we cannot utilize more than one FPGA per interface. However, in the future, it may be possible to link multiple FPGAs to one interface, allowing for more performance enhancements, and in the case of PCI interfaces, allow for greater bandwidth utilization.

We have considered the optimal speedups for parallel FPGAs. However, some efficiency loss should be expected, due to bus contention periods, data transfer bursts, etc. Despite this small loss, we do not expect the final speedup to be less than 185x, our re-

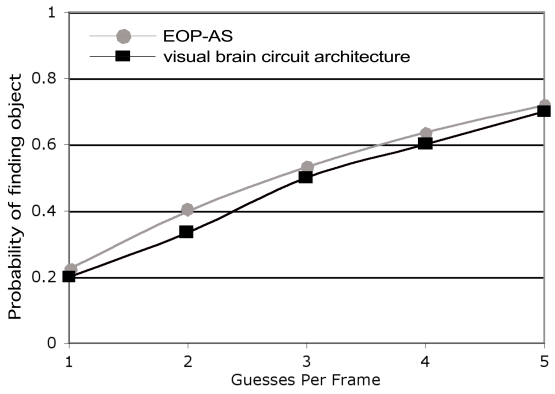

Figure 3. The results of our neocortical VBCA model compares very similarly to that of the EOP-AS model, showing accuracy between 20% and 70%. EOP-AS was estimated from the reported accuracy on "other room" tests of 22%[4].

quired speedup for real-time.

Our general purpose CPU driving the inputs to the FPGAs and collecting the results has been independently tested at a data rate of 1.1 Gbps. This rate is similar to that which would be required for three to four parallel FPGAs. The general purpose CPU may actually allow for far greater parallel FPGAs, but we have not yet tested the I/O beyond this limit.

C.2 Visual Object Recognition Accuracy

Accelerating the targeted computations is only valuable if the entire algorithm performs well in visual object recognition. At the time of this writing, the only other published results for accuracy on the Wiry Object Recognition Database, as mentioned in Section 1, is that of Ref. 4. That work uses an aggregation sensitive version of the cascade edge operator probes (EOP-AS), and differs significantly from our algorithm.

Fig. 3 compares our results with the aforementioned. As can be seen, the VBCA model achieves similar recognition accuracy to that of EOP-AS. As the systems increase guesses, from one to five, the probability of finding the sitting stool in the image increases from 20% to 70%, respectively. Both models produce very good results.

However, our model shows massive parallelism is possible; and, in our work, we have implemented part of that parallelism to produce results significantly faster. In the work of Ref. 4, they "assume access to large amounts of memory and CPU time," noting that all component computations on one 1.67 GHz Athlon CPU require "approximately one day"[4].

D Conclusion

We have presented a visual processing algorithm (VBCA) derived from brain circuitry, shown that its accuracy is comparable to the best reported algorithms on a standard com-

puter vision benchmark task, and shown that the algorithm can improve linearly with the addition of parallel processing elements. We have found our algorithm works most efficiently by using many small computational units coupled with flexible or custom hardware, which can be implemented in FPGAs. A single Xilinx Virtex 4 FX140 FPGA provided about a 62x performance improvement over general purpose CPUs in the feed-forward computations, four FPGAs could provide enough speedup to perform computations in real-time, and 11 FPGAs could allow more frames to be processed per second for a shorter reaction time. The speedup per dollar and speedup per watt ratios are orders of magnitude better on FPGAs compared to CPUs.

D.1 Future Work

The superior performance of brain circuits on many tasks may be in part due to their unusual operation, using highly parallel algorithms to achieve unusually powerful computational results. These brain circuit methods have been especially difficult to study empirically due to their slow operation on standard processors, yet they perform excellently when implemented on appropriate parallel systems. It is hoped that the methods presented here will enable further brain circuit modelling to test larger and more complex brain systems.

The work described here has been limited by several factors. While we have met all of our very high memory requirements, additional on-chip FPGA memory (BRAM) would allow us to study the effects of increased RF2 size. Also, increased logic would allow us to add even more parallelism to our design, to better mimic the massive parallelism found in the brain, which can improve overall performance. Lastly, improved interfaces, such as additional PCIe slots or 8.0 Gbps Fibre Channel ports, could double our current speedups.

With new Xilinx Virtex 5 LX330 FPGAs, we can hope to reimplement our design and achieve further improvements. That FPGA includes more on-chip memory, faster logic resources, and overall more logic resources. Such a solution addresses several of our current limitations. Speedups beyond what we have already demonstrated could allow for even shorter reaction times or additional frames per second of analyzed data.

Many systems including embedded computations, especially robotics, would benefit from extremely compact, low-power, fast processing, which has been elusive despite efforts in these fields. Further exploration of novel intrinsically parallel algorithms and their low-power parallel implementation may confer substantial benefits to these areas of research.

Acknowledgements

The research described herein was supported in part by the Office of Naval Research and the Defense Advanced Research Projects Agency.

References

1. *ADM-XRC-4FX Datasheet*. http://www.alpha-data.com/adm-xrc-4fx.html
2. *ADPe-XRC-4 Datasheet*. http://www.alpha-data.com/adpe-xrc-4.html
3. C. Bruce, R. Desimone and C. Gross, *Visual properties of neurons in a polysensory area in the superior temporal sulcus of the macaque*, Neurophysiol, **46**, 369–384, (1981).

4. O. Carmichael, *Discriminative techniques for the recognition of complex-shaped objects*, PhD Thesis, The Robotics Institute, Carnegie Mellon University, Technical Report CMU-RI-TR-03-34, (2003).
5. O. Carmichael and M. Hebert, *WORD: Wiry Object Recognition Database*, Carnegie Mellon University, (2006). `http://www.cs.cmu.edu/ owenc/word.htm`
6. R. Coultrip, R. Granger and G. Lynch, *A cortical model of winner-take-all competition via lateral inhibition*, Neural Networks, **5**, 47–54, (1992).
7. A. Felch and R. Granger, *The hypergeometric connectivity hypothesis: Divergent performance of brain circuits with different synaptic connectivity distributions*, Brain Research, (2007, in press).
8. R. Granger, *Brain circuit implementation: High-precision computation from low-precision components*, in: Replacement Parts for the Brain, T. Berger, D. Glanzman, Eds., pp. 277–294, (MIT Press, 2005).
9. R. Granger, *Engines of the brain: The computational instruction set of human cognition*, AI Magazine, **27**, 15–32, (2006).
10. R. Granger, *Neural computation: Olfactory cortex as a model for telencephalic processing*, Learning & Memory, J. Byrne, Ed., pp. 445–450, (2003).
11. C. Gao, and D. Hammerstrom, *Platform performance comparison of PALM network on Pentium 4 and FPGA*, in: International Joint Conf. on Neural Networks, (2003).
12. J. Haxby, M. I. Gobbini, M. Furey, A. Ishai, J. Schouten and P. Pietrini, *Distributed and overlapping representations of faces and objects in ventral temporal cortex*, Science, **293**, 2425–2430, (2001).
13. D. Hubel, and T. Wiesel, *Receptive fields and functional architecture in two nonstriate visual areas (18 and 19) of the cat*, J. Neurophysiol, **28**, 229–289, (1965).
14. Y. Kamitani and F. Tong, *Decoding the visual and subjective contents of the human brain*, Nat.Neurosci, **8**, 679–685, (2005).
15. D. Mumford, *On the computational architecture of the neocortex*, Biological Cybernetics, **65**, 135–145, (1991).
16. R. Rao and D. Ballard, *Predictive coding in the visual cortex: a functional interpretation of some extra-classical receptive-field effects*, Nat Neurosci, **2**, 79–87, (1999).
17. A. Rodriguez, J. Whitson and R. Granger, *Derivation and analysis of basic computational operations of thalamocortical circuits.*, J. Cog. Neurosci., **16**, 856–877, (2004).
18. B. Resko, et al., *Visual Cortex Inspired Intelligent Contouring*, in: Intelligent Engineering Systems Proceedings, September 16-19, (2005).
19. J. Starzyk, Z. Zhu and T. Liu, *Self Organizing Learning Array*, IEEE Trans. on Neural Networks, **16**, 355-363, (2005).
20. J. Szentagothai, *The module concept in cerebral cortex architecture*, Brain Research, **95**, 475–496, (1975).
21. F. Valverde, *Structure of the cerebral cortex. Intrinsic organization and comparative analysis of the neocortex*, Rev. Neurol., **34**, 758–780, (2002).
22. G. Wallis and E. Rolls, *A model of invariant object recognition in the visual system*, Prog. Neurobiol, **51**, 167–194, (1997).
23. R. K. Weinstein and R. H. Lee, *Architecture for high-performance FPGA implementations of neural models*, Journal of Neural Engineering, **3**, 21–34, (2006).