Towards an Embedded Biologically-Inspired Machine Vision Processor

Vinay Sriram and David Cox
Rowland Institute at Harvard
Harvard University
Cambridge, MA 02142
{sriram, cox}@rowland.harvard.edu

Kuen Hung Tsoi and Wayne Luk
Custom computing group
Imperial College London
London, UK
{khtsoi, wl}@doc.ic.ac.uk

***Abstract*—Biologically-inspired machine vision algorithms – those that attempt to capture aspects of the computational architecture of the brain – have proven to be a promising class of algorithms for performing a variety of object and face recognition tasks. However these algorithms typically require a large number of arithmetic operations per image frame evaluated. Meanwhile, the increasing ubiquity of inexpensive cameras in a wide array of embedded devices presents an enormous opportunity for the deployment of embedded machine vision systems. As a first step towards an embedded implementation, we consider the main requirements for the design of an embedded processor for biologically-inspired object recognition and demonstrate an FPGA prototype of the *V1-like* algorithm, a simple biologically-inspired system from the literature [1], [2], [3]. We present a multiple instruction, single data (MISD) pipeline implementation of *V1-like*, and show that such designs are feasible in an FPGA context, particularly for small frame sizes (e.g. 100x100). In addition, we show that such an implementation offers good performance per unit silicon area and power dissipation in comparison to traditional CPU and GPU implementations. Finally, we discuss the constraints under which such an embedded strategy would be feasible for a more general biologically inspired face recognition system, and consider paths forward towards a wider range of possible embedded targets.**

I. Introduction

Biologically-inspired object recognition algorithms have been shown to be promising candidates for the task of object recognition [4], [5], achieving state-of-the-art or near-state-of-the-art performance on a variety of object and face recognition tasks [6]. However, because the brain's visual system is itself a highly-parallel computer, with hundreds of millions of processing units (neurons), biologically-inspired algorithms that attempt to mimic even a fraction of this scale are typically highly parallel and arithmetically expensive. Thus, it is common for biologically-inspired implementations to use either large-scale compute clusters or general purpose graphics processing units (GPGPU) [6].

Meanwhile, in recent years, there has been an explosion in the deployment of embedded cameras in a wide variety of contexts (e.g. smart phones, automobiles, robots, surveillance platforms, etc.). While numerous applications could benefit from biologically-inspired algorithms for "smart" extensions of these imaging platforms (e.g. face recognition in a smart phone, or pedestrian detection in automotive platform), design size and power constrains dominate in these contexts.

While important work has been done in exploring various kinds of analog and asynchronous "neuromorphic" designs that attempt to mimic brain computations using relatively exotic hardware [7], here we instead focus on a proven family of algorithms using standard synchronous digital hardware. A variety of embedded vision processing approaches have been explored in an embedded context (e.g. [8], [9], [10], [11]) addressing a variety of different vision problems. The goal of the present work is to explore one particularly promising class of object and face recognition algorithms, and to engage in a systematic study of different parallelization techniques as they pertain to an embedded system.

Here, we consider *V1-like* [1], a vision system based on a simple first-order approximation of the processing that takes place in the first stage of cortical visual processing (area "V1") in mammals [12]. In addition to achieving state-of-the-art performance on a variety of face and object recognition test sets [1], [2], [3], the basic processes embodied in *V1-like* can also be stacked in multi-layer systems, which have been shown to yield even better performance across a range of object and face recognition tasks [6].

We first explore the design space of possible parallelization techniques for accelerating the most computationally intense sub-operation contained within our system. Our exploration spans single instruction, single data (SISD), single instruction multiple data (SIMD), and multiple instruction single data (MISD) on each of their current standard "premiere" platforms, namely CPUs, GPUs and FPGAs, respectively. This analysis allows each accelerated version of the algorithm to be compared on the basis of performance per unit price, power and silicon area used, with each version operating on hardware highly optimized for each parallelization regime.

Having identified a reasonable strategy for parallelism that achieves good performance per power and size for the computational bottleneck of the algorithm at hand, we present a more complete implementation of the system, and evaluate the resource constraints imposed by this design. Finally, we examine the challenges and opportunities presented by our particular design, discuss contexts where it could be reasonably deployed as-is, and explore contexts where substantially different strategies would be required.

II. A BIOLOGICALLY-INSPIRED VISION ALGORITHM

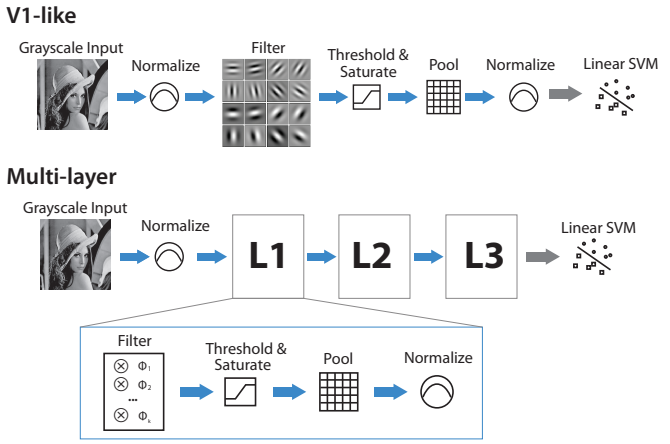


Fig. 1. **A schematic diagram of the system architecture of the family of models considered.** Models contain one or more layers of processing, with the *V1-like* being a special case of the one-layer models were filter kernels are gabor filters. Here, we consider the implementation of a single layer as a canonical unit for an embedded biologically-inspired vision processor.

A variety of different biologically-inspired machine vision systems have been proposed [13], with the work of Fukushima [14] representing one of the earliest efforts. A common theme in models of this type is a hierarchy of "layers" of simple processing units, in which each unit performs a weighted sum on the input from the preceding layer, followed by a series of nonlinear processing steps. Since the linear stage of processing can be modeled as a convolution, neural networks of this kind are often called *convolutional* neural networks. Here, we follow the basic "V1-like" algorithm described by Pinto et al. [1], [2], [3]. This algorithm also essentially comprises the single "layer" which can be cascaded into multiple layers as described in [6] to produce more powerful models. The canonical *V1-like* layer itself consists of a series of processing steps, the flow of which is shown in Figure 1. First, an input image is converted to grayscale, and it is resized to a common "retina" size by bicubic interpolation. Next, a local divisive input normalization step is applied. In this step, for each pixel in the input image, a neighborhood of pixel values within a local region around that pixel are unrolled into a vector, and the output at that pixel position is computed by subtracting the mean of the values in the vector and then dividing by the Euclidean norm of that vector. The resulting normalized image is then convolved with a bank of N filter kernels, resulting in a stack of N "feature" maps. For the *V1-like* model, the filterbank consists of a set of Gabor wavelet filters, spanning a series of orientations, spatial frequencies, and phases. As a subcomponent of more general multilayer architectures (e.g. [6]), these filters need not be Gabor wavelets and would be learned "offline" by an unsupervised learning procedure, see [6]). The results of this filtering stage are subsequently subjected to threshold and saturation functions, which clip values outside of a given range $[L, U]$ to L and U, respectively. Finally, the resulting outputs are divisively normalized again, and the resulting feature representation is unrolled into a feature vector, which becomes the input to standard machine learning tools (e.g. a support vector machine [15]). To perform object or face recognition with these features, labeled training examples are processed using the algorithm, and the resulting feature vectors are used to train a classifier. During "running" mode, feature vectors for new examples are submitted to the trained classifier, which assigns labels on the basis of experience with training data. While many different classifiers could be used for this final step, in practice, linear classifiers have proven successful [2], and they have the advantage that they only require a single dot product and threshold to classify new examples. While a variety of different numbers and sizes of filters could be used, we limit ourselves here to models with 16 filters, each 3x3 in size. These filters are significantly smaller than those described in [1], but are consistent with the filter sizes used in the multi-layer systems described in [6]. Finally, we perform all of computations in fixed point; while the original *V1-like* algorithm was implemented in floating point, it is easily converted to fixed point, without significant loss in accuracy.

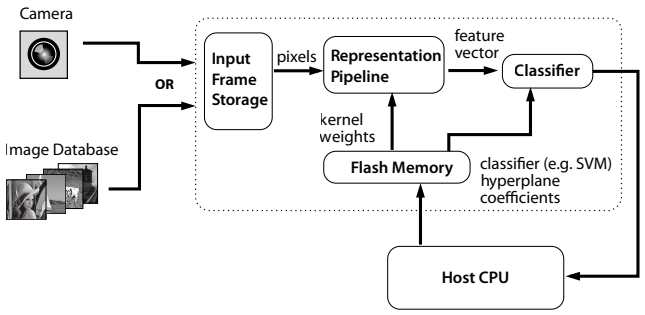


Fig. 2. **A proposed system layout for an embedded biologically inspired object recognition system.**

III. PROPOSED SYSTEM ARCHITECTURE

The overall structure of our proposed embedded vision system is shown in Figure 2. Images taken either from a camera or alternately from an image database are buffered and then fed into a representation pipeline. The pipeline here consists of 16 2D convolution cores and a normalization core. "Non-architectural" model coefficients, e.g. filter kernel weights, normalization parameters, etc., and trained classifier coefficient are stored in flash memory, and resulting classifier decisions are fed to the embedded system's host CPU. Through profiling, we established that 2D convolution accounts for 89% of total computation time. To avoid problems associated with slow intra-device data transfer speeds, we focused on implementations where all components of the representation pipeline, i.e. convolution and normalization, are performed by a single processor. In the next section we explore the use of various parallelization regimes to accelerate the most computationally complex sub-operation in the *V1-like* algorithm.

IV. Candidate implementations for 2D convolution

While there have been many published examples of GPU and FPGA acceleraton of single filter 2D convolution (e.g. [17]), acceleration of multi-filterbank 2D convolution has been less explored. Since filterbank convolution is the most compute intensive sub-operation in the *V1-like* algorithm, we first focus on understanding this sub-operation using single instruction single data (SISD), single instruction multiple data (SIMD) and multiple instruction single data (MISD) techniques. We then realize each design using a CPU (a SISD machine with SIMD capabilities), a GPU (a highly parallel SIMD machine) and an FPGA (which allows for the implementation of any parallelization regime).

A. SISD implementation on a CPU

Our reference CPU implementation of 2D convolution is written using the C programming language and compiled using GCC 4.2.4. As this is a reference design, no effort was made to take advantage of the CPU's SIMD capabilities, and it achieves 0.65 GOPS for all frame sizes.

B. SIMD implementation on a GPU

To explore SIMD implementation on GPU hardware, we developed a series of three designs, starting with a naive version, and moving to successively more optimized ones that achieve better utilization of the GPU's compute resources. Our first SIMD design was a naive implementation of 2D convolution that does not use shared memory. It achieved up to 257 GOPS performance on a 1000x1000 frame. Our second design made better utilization of the data-level parallelism and faster memory bandwidth by using shared memory to store pixels; this achieves 4.62 GOPS for a 100x100 frame, and up to 401 GOPS for a 1000 x 1000 frame. For small frame sizes we do not achieve proper utilization of SMs. In our third design, we execute multiple 2D convolution kernels in parallel. In doing so we achieve 256 GOPS for a 100x100 frame and over 473 GOPS for a 1000x1000 frame.

C. MISD implementation on an FPGA

A common paradigm for instruction-level parallelism is pipelining. In this approach, pixels are loaded into a pipeline sequentially, such that multiple operations are effectively applied to the same pixel value at once. We perform 16 parallel 3x3 2D convolutions on each input image. To achieve this in a pipeline, kernel coefficients for all 16 kernels are preloaded into registers, while pixels are loaded in from the left. Each clock cycle a new pixel is loaded in, all the pixels in the pipeline are shifted right and the rightmost pixel is discarded. Every clock cycle the pixels are multiplied with the corresponding kernel coefficients, and the results of multiplications are added to generate the new convolved pixel.

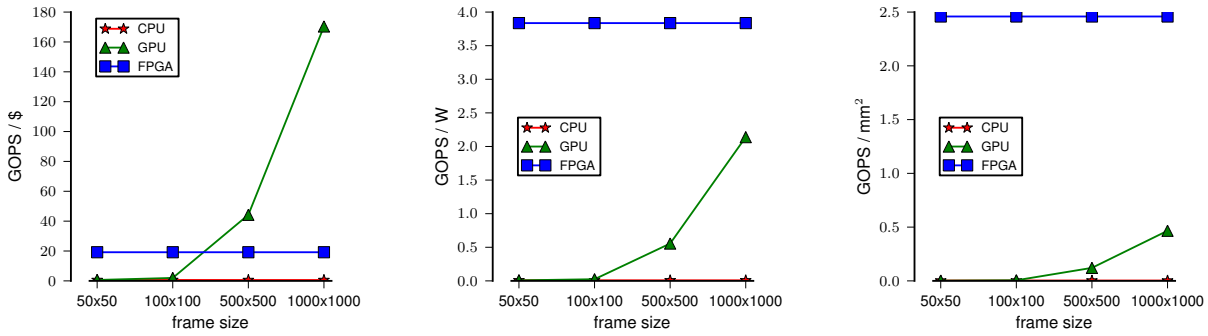
We use external memory to store each input image, using the memory controller provided by the Alpha Data API [16], explicitly avoiding BRAM in an effort to minimize the clock speed impact of data being distributed across the chip. We use the inbuilt DSP48E slices to carry out the multiplication and addition. Once the pipeline is filled, we read one pixel and output 16 pixels per clock cycle. However, we only have four banks of memory and each bank can supply 128-bit in one clock cycle. We can read/write 8 pixels from each bank and so we need at least two banks for output, while one bank is used for input. Our design is flexible in that we can change the input frame size and kernel values at runtime. Kernel size, however, must remain constant. Input frames are first stored in external memory, and address generation logic gathers pixels from external memory and feeds DSP48E slices. Our design was developed using the Xilinx ISE 11.5 tool chain and is currently clocked at 265MHz, using 24,428 Xilinx Slices, 22,281 Xilinx LUTs and 144 DSP48Es.

D. Comparisons of GOPS, power and silicon area efficiency

In Table I we present a summary of the computation time, computations per clock and the time to perform the convolution for the SIMD and MISD implementations. The MISD implementation on the FPGA performs well for the small frame sizes but fails markedly for larger frames. In our MISD implementation, pixel values are sequentially loaded into the pipeline and are read out once from cache, wasting little time in moving data to and from cache. We attribute our MISD implementation's high computation time for frames larger than 500x500 to slow clock speed, relatively fewer number of computations carried out per clock cycle and high initial pipeline latency. Slow clock speed is a limitation of the FPGA while the other two problems are limitations of the MISD parallelization technique.

Our best GPU implementation achieved over 470 GOPS, approximately half of the maximum theoretical performance of 933 GOPS [19]. Our FPGA implementation on the Virtex 5 LX330 achieved 95 GOPS. It should be noted that the FPGA used here is one generation old, and belongs to the less DSP-heavy "LX" subfamily. The newer Virtex 6 SX240 FPGA offers over 2000 DSP slices (as compared to 192 in the LX330) [20]. Given that the processing can be decomposed into multiple parallel pipelines to take advantage of more multipliers, we predict that an FPGA implementation on DSP-oriented Virtex 6 could potentially achieve a greater-than 10X speed-up relative to the Virtex 5 LX330. Interestingly, this could place the raw performance of the FPGA-based design on par with that of the Tesla C1060 GPU.

We also consider performance with respect to the various costs associated with each implementation. We acquired quotes for representative systems of each class through personal communication with the respective manufacturers, and in Figure 3(a) we plot frame size versus GOPS per unit of price. We note that for all frame sizes, the GPU achieves substantially better price / performance ratio when compared to the other options considered here. Other important considerations in embedded applications are the power requirements and silicon area used. Thus, we consider GOPS achieved with respect to processor power usage. The Intel E5405 uses a peak 80W of power [21] while the NVIDIA Tesla C1060 uses



(a) Performance per US$ (b) Performance per watt (c) Performance per mm^2

Fig. 3. **Comparison of the various implementations based on GOPS, price, power and silicon area utilization.**

TABLE I
OUR MISD-FPGA AND SIMD-GPU IMPLEMENTATION PERFORMANCE COMPARISONS

	FPGA					GPU		
Frame size	Mad /clk	Clock (GHz)	Latency (clock cycles)	Time (ms)	Time to transfer data to and from shared memory (%)	Clock (GHz)	Time (ms)	Time to transfer data to and from shared memory (%)
50 x 50	144	0.333	153	0.007	<1	1.5	0.55	57.3
100 x 100	144	0.333	303	0.03	<1	1.5	0.56	57.1
500 x 500	144	0.333	1503	0.75	<1	1.5	0.61	57.6
1000 x 1000	144	0.333	3003	3	<1	1.5	0.61	55.9

187.8W of power [19] and the Alpha Data FPGA board uses 25W of power [16]. In Figure 3(b), we plot the frame size versus GOPS per unit of power. We note that for every level frame size, the FPGA offers significantly higher GOPS per unit of power.The FPGA maintains a constant 3.8 GOPS per watt. The GPU offers a maximum of 2.5 GOPS per watt for frame size 1000x1000. The CPU on the other hand offers a constant 0.008 GOPS per watt, which is the lowest among all the platforms. Finally, in Figure 3(c), we plot GOPS per unit area of silicon used for all our implementations. The die size of Intel E5405 is 214 mm2 [21] while the die size of Tesla C1060 is 862 mm2 [19]. The silicon area of the FPGA solution was calculated as per the methodology detailed in [22]. Again, the FPGA implementation stands out where size is important.

V. SYSTEM IMPLEMENTATION

From our design space exploration above, we identified that an MISD filterbank convolution implementation on the FPGA yields promising performance while remaining consistent with the requirements of an embedded application. Thus, we set out to develop a more complete FPGA implementation. Figure 2 shows a basic schematic of this system. A camera or image database residing on a host PC sends frames to an FPGA-based recognition system, along with unchanging parameters such as filter coefficients and model parameters. The frames and parameters are stored in SDRAM, and are accessed through a DMA channel, using an API provided with the Alpha Data ADM-XRC-5T2 accelerator used to implement our prototype. Each input frame is processed by the representation pipeline and a feature vector is generated for the frame. A classifier uses this feature vector and compares it with the feature vector of the target to be identified.

The internal logic of the representation pipeline is shown in Figure 4. In a 3x3 convolution, three rows of pixels from the input frame are loaded into the pipeline before the multiplication of the pixels with the corresponding kernel values. Therefore, we first write the pixels from the three rows into two-line buffers. We do not need a line buffer for the first row as it is directly loaded into the pipeline. For line buffers, we use dual-ported FPGA BRAMs, each of which has two write ports and a single read port.

Each 2D convolution core uses 9 DSP48E slices, and we use FPGA lookup tables and flip flops to carry out the addition and square root functions needed for the normalization step. The L and U values are used to clip the output pixel values (i.e. L<pixel value<U), enabling thresholding and saturation operations to be computed efficiently. Each core is deeply pipelined, meaning that all internal operations are immediately registered. The registers used to hold the intermediate data are represented by rectangle boxes in Figure 4. The inputs from the line buffers are distributed to all 16 kernels in parallel while the kernel mask values, K[], are individually configured. Thus we generate 16 results every clock cycle. The result of each convolution cycle is a new pixel which is labeled as R in the figure. The maximum number of parallel convolutions and the size of each kernel is limited by the available DSP blocks in the FPGA.

A key feature of the "V1-like" model is the divisive normalization of the output of each simulated neuron by the output of its neighbors. Here we implement a "filter-wise" normalization

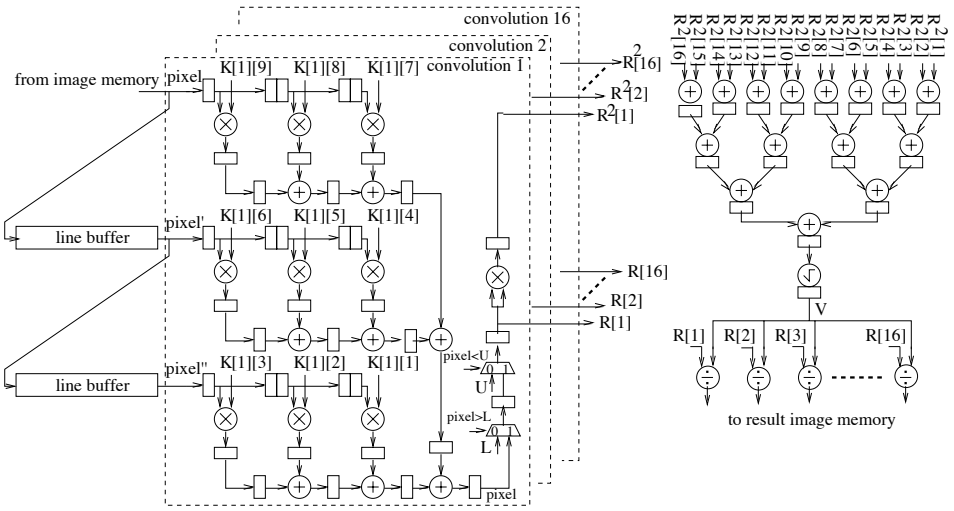


Fig. 4. **Internal logic of the representation pipeline.** The left hand side of the figure illustrates the convolution process and the right hand side illustrates the normalization process. The rectangle boxes represent registers used to hold intermediate data, L and U refers to the lower and upper normalization clipping values and R refers to the resulting pixel value after each convolution step.

TABLE II
FPGA RESOURCE UTILIZATION AND ESTIMATED ASIC GATE COUNT WHEN PROCESSING A 100X100 FRAME

FPGA resource	Resources used	Resources used (%)	Gate count
Xilinx Slices	15,455	29	77,275
DSP48Es	160	83	352,000
LUTs	26,397	12	158,382
External memory	340KB	NA	2,720,000
Total	–	–	3,307,657

operation in which the output at a given pixel location for a given filter is divided by the sum of the squares of the values for the corresponding pixels from the outputs of the other filters in the filterbank. While other normalization schemes are possible (e.g. also including neighboring pixels in the divisor), this particular normalization operation is convenient to carry out in our MISD implementation, because we generate all the pixels required for normalization each clock cycle. Figure 4 shows a schematic diagram of our FPGA implementation of normalization. The squared pixel outputs are summed using an adder tree. A modified SBTM algorithm [24] is used to approximate the square function. In this process we generate the norm value labeled as 'v' in the figure. All clipped outputs are divided by v using division cores generated by the Xilinx CoreGen tool. The 16 final outputs are grouped into two 128-bit words and are written to the two output memory banks.

In Table II we present our FPGA resource utilization and estimated equivalent ASIC gate count (calculated using the method presented in [25]). Our prototype was clocked at 200 MHz on a Xilinx LX330 FPGA chip, and used 29% of Xilinx Slices, 12% of Xilinx LUTs, 9% of BRAMs and 83% of DSP48Es. We require substantial large amount of memory to store our input and output images. In the best case scenario where we operate on the 100x100 image size, we need to store 1 input and 16 output frames where each pixel is 16 bit wide, for a total of 340KB of data. When implemented on a 0.18um CMOS 6-ML ASIC this would require over 3.3 million ASIC gates. Ideally, an embedded system would target 100K - 150K ASIC gates, so there remain substantial hurdles to be overcome. Memory is the most heavy user of ASIC gates, followed by the multipliers. Our prototype can process a 100x100 frame in just 0.05 ms to process a 100x100 frame, of which 0.03ms are required for the 2D convolution and the rest of the computation time is required for normalization.

VI. DISCUSSION AND FUTURE WORK

In this paper, we presented an analysis of where various implementations of a biologically inspired vision algorithm succeed and fail, with an eye to assessing the feasibility of an embedded visual recognition processor. We verified that a SIMD GPU implementation is ideal for applications that are price sensitive but largely power insensitive. When power and size are critical, such designs suffer from the fact that much of the size and power consumption is driven either by components that are irrelevant to the task at hand (e.g. graphics-specific hardware in a GPU), or which support functions that make the processor supremely general (e.g. the considerable cache and memory manipulation infrastructure of a CPU and GPU). Our pipelined MISD FPGA reference implementation of the *V1-like* algorithm provides a basic existence proof for the feasibility of "lean" designs, where a majority of power and area is dedicated directly to computation.

While the *V1-like* algorithm considered here can be useful in its own right, a longer-term goal of this work is to enable more powerful multi-layer systems (as in [6]). To this end, we project the resource utilization of multi-stage vision systems, based on our single-layer reference implementation. While we have so far considered a pipeline that implements only the first layer of a system such as shown in Figure 1, we can realize additional layers by iterating through this

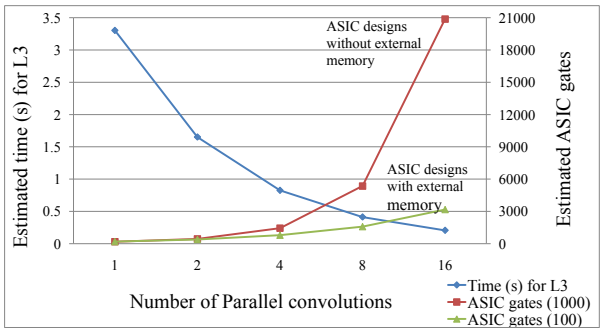


Fig. 5. **Projected area-time analysis for a hypothetical 3-layer system.** The model considered here has 16 filters in the first layer, 32 filters in the second layer and 128 filters in the third layer. The x axis shows the number of parallel convolutions implemented inside a single representation pipeline.

representation pipeline multiple times. Since the multi-layer systems described in [6] typically have variable number of filters per layer, for sake of simplicity we confine ourselves here to a model architecture with 16 filters in the first layer, 32 in the second, and 128 in the third layer, since this architecture was found to be effective in [6]. In this case, we need to iterate through the representation pipeline once in first layer, 32 times in the second layer and 128 times for each second layer output in the third layer. The estimated ASIC gate count includes the gate count for memory required to store intermediate frame data as well as the gate count for the multipliers required for realizing that level of parallelism. As noted earlier, memory and multipliers are by far the most expensive resources in our design as they require large number of ASIC gates. In Figure 5 we plot area versus time for an three-layer model implementation for ASIC designs that have external memory as well as for ASIC designs that do not have external memory. Use of external memory allows for smaller and more compact ASIC designs because all intermediate data can be stored in external memory. However, the luxury of external memory is not always feasible in consumer-oriented embedded applications, and movement in such a direction would require a different approach, relaxing memory requirements, perhaps at the expensive of optimal parallelism and data reuse. For somewhat less constrained embedded applications, however, our results suggest a potentially promising path forward.

ACKNOWLEDGMENT

The authors would like to thank the Rowland Institute at Harvard for supporting this work. The AXEL cluster was supported by an ICL Research Excellence Award, Alpha Data, NVIDIA and Xilinx.

REFERENCES

[1] Pinto, N., Cox, D. and DiCarlo, J. (2008) Why is real-world visual object recognition hard? PLoS Computational Biology. vol 4, no 1. doi:10.1371/journal.pcbi.0040027.

[2] Pinto, N., DiCarlo, J. and Cox, D. (2008) Establishing Benchmarks and Baselines for Face Recognition. Proc. ECCV Faces in Real Life Images, Marseille-France.

[3] Pinto, N., DiCarlo, J. and Cox, D. (2009) How far can you get with a modern face recognition test set using only simple features? IEEE Computer Vision and Pattern Recognition, Miami-USA.

[4] Serre, T., Wolf, L., Bileschi, S., Riesenhuber, M. and Poggio, T. (2007) Object recognition with cortex-like mechanisms. IEEE Transactions on Pattern Analysis and Machine Intelligence. vol 9, no 1, pg 411426.

[5] Arathorn, D. (2002) Map-seeking circuits in visual cognition: A computational mechanism for biological and machine vision. Stanford University Press.

[6] Pinto, N., Doukhan, D., DiCarlo, J. and Cox, D. (2009) A High-Throughput Screening Approach to Discovering Good Forms of Biologically-Inspired Visual Representation. PLoS Computational Biology. vol 5, no 11. doi: e1000579. doi:10.1371/journal.pcbi.1000579.

[7] Koch, C. and Mathur, B. (1996) Neuromorphic vision chips. IEEE Spectrum. vol 33, vol 1, pg 3846.

[8] Chung, Y. and Prasanna, VK (1997) Parallel object recognition on an FPGA-based configurable computing platform. International Workshop on Computer Architectures for Machine Perception.

[9] Malamas, E.N., Petrakis, E.G.M., Zervakis, M., Petit, L. and Legat, J.D. (2003) A survey on industrial vision systems, applications and tools. Image and Vision Computing. 21(2): 171-188.

[10] MacLean, W.J. (2005) An evaluation of the suitability of FPGAs for embedded vision systems. IEEE Conference on Computer Vision and Pattern Recognition.

[11] Farabet, C., Martini, B., Akselrod, P., Talay, S., LeCun, Y., and Culurciello, E. (2010) Hardware Accelerated Convolutional Neural Networks for Synthetic Vision Systems. Proc. of IEEE International Symposium on Circuits and Systems, Paris-France.

[12] Jones, J. and Palmer, L. (1987) An evaluation of the two-dimensional Gabor filter model of simple receptive fields in cat striate cortex. Journal of Neurophysiology. vol 58, no 6, pg 1233-1258.

[13] Hubel, D. and Wiesel, T. (1962) Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. Journal of Physiology. vol 160, no 1, pg 106-154.

[14] Fukushima, K. (1980) Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. Biological Cybernetics. vol 36, no 4, pg 193-202.

[15] Cortes, C. and Vapnik, V. (1995) Support vector networks. Journal of Machine Learning. vol 20, no 3, pg 273-297.

[16] Alpha Data ADM-XRC-5T2. Alpha Data datasheet for the PCIe Virtex 5 LX330 FPGA board. viewed on 12-12-2009, http://www.alpha-data.com/products.php?product=adm-xrc-5t2.

[17] Perria, S., Lanuzzaa, M., Corsonellob, P. and Cocorulloa, G. (2005) A high-performance fully reconfigurable FPGA-based 2D convolution processor. Microprocessors and Microsystems. vol 29, no 9, pg 381-391.

[18] Intel. Intel processors. viewed on 10 Nov 2009, http://www.intel.com/support/processors/xeon/sb/CS-020863.htm.

[19] Nvidia. Nvidia Tesla C1060 GPU, viewed on 01-04-2010, http://en.wikipedia.org/wiki/NvidiaTesla.

[20] Xilinx. Virtex-6 FPGA ML605 Evaluation Kit. viewed on 12-12-2009, http://www.xilinx.com/products/devkits/EK-V6-ML605-G.htm.

[21] Intel. Intel E5405 Chipset. viewed on 10 Nov 2009, http://processorfinder.intel.com/details.aspx?sSpec=SLAP2.

[22] Borgatti, M., Lertora, F. and Cali, L. (2003) A reconfigurable system featuring dynamically extensible embedded microprocessor, FPGA, and customizable I/O. IEEE Journal of Solid-State Circuits. vol 38, no 3, pg 521−529.

[23] Chiappetta, M. NVIDIA GeForce GTX 280 and GTX 260 Unleashed. viewed on 4/12/2009, http://hothardware.com/Articles/NVIDIA-GeForce-GTX-280-and-GTX-260-Unleashed/page=2.

[24] Schulte J. and Stine, J. (1999) Approximating Elementary Functions with Symmetric Bipartite Tables. IEEE Transactions on Computers. vol 48, no 8, pg 842-847.

[25] Wikipedia. Static random access memory. viewed on 05-03-2010, http://en.wikipedia.org/wiki/Staticrandomaccessmemory.