Fast Neuromimetic Object Recognition Using FPGA Outperforms GPU Implementations

Garrick Orchard, Jacob G. Martin, R. Jacob Vogelstein, and Ralph Etienne-Cummings, *Fellow, IEEE*

***Abstract*— Recognition of objects in still images has traditionally been regarded as a difficult computational problem. Although modern automated methods for visual object recognition have achieved steadily increasing recognition accuracy, even the most advanced computational vision approaches are unable to obtain performance equal to that of humans. This has led to the creation of many biologically inspired models of visual object recognition, among them the hierarchical model and X (HMAX) model. HMAX is traditionally known to achieve high accuracy in visual object recognition tasks at the expense of significant computational complexity. Increasing complexity, in turn, increases computation time, reducing the number of images that can be processed per unit time. In this paper we describe how the computationally intensive and biologically inspired HMAX model for visual object recognition can be modified for implementation on a commercial field-programmable aate Array, specifically the Xilinx Virtex 6 ML605 evaluation board with XC6VLX240T FPGA. We show that with minor modifications to the traditional HMAX model we can perform recognition on images of size 128 × 128 pixels at a rate of 190 images per second with a less than 1% loss in recognition accuracy in both binary and multiclass visual object recognition tasks.**

***Index Terms*— Field programmable gate arrays, object recognition, HMAX.**

I. Introduction

OBJECT recognition has received a lot of attention in recent years and is an important step toward building machines that can understand and interact meaningfully with their environment. In this context, both a high recognition accuracy and a short recognition time are desirable. By shortening the recognition time even further, we foresee applications that include rapidly searching and categorizing images on the Internet based on features extracted from their pixel content on the fly. Many currently available image search and characterization platforms rely on image metadata and watermarks rather than the images' actual pixel values, while those platforms that do make use of actual pixel values typically rely on previously extracted image features rather than creating and extracting new features on the fly.

The challenge of consistently recognizing an object is complicated by the fact that the appearance of the object can vary significantly depending on its location, orientation, and scale within an image. Reliable object recognition must therefore be invariant to translation, scale, and orientation. Some methods of object recognition incorporate these invariances, such as the scale-invariant feature transformation (SIFT) [1] or speeded-up robust features (SURF) [2]. These models achieve good recognition rates, but still fall far short of the recognition rates achieved by humans. There is evidence suggesting that, after viewing an object for the first time, a biological system is capable of recognizing that object again at a novel position and scale [3]. The object can also be recognized if it is slightly rotated, but the recognition accuracy decreases when the object is rotated too far from a familiar view [3]. A biologically inspired model that shares this property of scale and translation invariance but also achieves only limited rotation invariance is the hierarchical model and X (HMAX) [4] in which the "X" represents a nonlinearity.

Jarrett *et al.* [5] investigated which architecture is best for object recognition. They found that nonlinearities are the most important feature in such models. Their results show that rectification and local normalization significantly improve recognition accuracy. Their results also indicate that a multistage method of feature extraction outperforms single-stage feature extraction. The HMAX model is a multistage model that mixes Gabor filters in the first stage with learned filters in the second. HMAX is intended to model the first 100–200 ms of object recognition due to purely feed-forward mechanisms in the ventral visual pathway [4]. HMAX is biologically inspired and incorporates rectification and local normalization nonlinearities, both of which were later recommended by Jarret *et al.* [5] as important properties for object recognition models.

In this paper, we focus specifically on the version of HMAX described in [6]. The recognition accuracy of HMAX is well below that of the biological counterparts it attempts to mimic for real-world tasks because it only mimics the first stages of the feed-forward pathways. However, HMAX performs comparably to its biological counterparts on rapid characterization tasks in which a stimulus is presented long enough for feed-forward recognition to take place, but short enough to prevent top-down feedback from having an effect [7], [8]. HMAX provides a valuable step toward achieving higher recognition accuracy and better understanding the operation of the ventral

Manuscript received June 10, 2012; revised March 10, 2013; accepted March 13, 2013. Date of publication April 18, 2013; date of current version June 28, 2013. This work was supported by the Defense Advanced Research Projects Agency NeoVision2 Program under Government Contract HR0011-10-C-0033 and the Research Program in Applied Neuroscience.

G. Orchard is with the Department of Electrical and Computer Engineering, The Johns Hopkins University, Baltimore, MD 21218 USA, and also with the Singapore Institute of Neurotechnology, National University of Singapore, 117456 Singapore (e-mail: gorchard@jhu.edu).

J. G. Martin and J. Vogelstein are with the Johns Hopkins University Applied Physics Laboratory, Laurel, MD 20723 USA (e-mail: jacob.martin@jhuapl.edu; jacob.vogelstein@jhuapl.edu).

R. Etienne-Cummings is with the Department of Electrical and Computer Engineering, The Johns Hopkins University, Baltimore, MD 21218 USA (e-mail: retienne@jhu.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNNLS.2013.2253563

stream in the visual cortex. Biological processing systems (networks of neurons) are inherently distributed and massively parallel. If we intend to achieve comparable recognition rates by mimicking biological processing, then we too should use distributed and massively parallel hardware that is suited to the task.

Originally, object recognition models were typically run on sequential processors (CPUs), for which Mutch and Lowe [9] developed the feature hierarchy library (FHLib) tool in 2006 for implementing hierarchical models such as HMAX. CPUs require little effort to program and offer great flexibility, allowing them to be used for a large variety of tasks, but the sequential nature of their processing makes them ill suited to an application such as HMAX. Modern CPUs are capable of impressive performance and allow some parallel processing, but depending on the nature of the algorithm to be implemented, it can be very difficult, if not impossible, to fully utilize the theoretical computational capacity of such devices. In 2008, Chikkerur [10] reported a multithreaded CPU implementation of HMAX, showing that the increased parallelism outperformed previous CPU implementations.

GPUs allow even more parallel processing paths, but writing code for GPUs requires a larger effort than for CPUs. GPUs also offer greater control of data flow and storage during computation, which allows programmers to make greater use of the theoretical computational capacity. In the same paper as his multithreaded CPU implementation [10], Chikkerur presented a GPU implementation of HMAX with even more parallel processing paths, which outperformed the multithreaded CPU implementation by 3–10$\times$ depending on input image size. Soon GPU technologies were being used extensively for HMAX and, in 2010, Mutch and Lowe released the cortical network simulator (CNS) [11] which uses a GPU for processing and can speed up the HMAX model by 97$\times$ compared to the FHLib software it was intended to replace. Later, in 2010, Sedding *et al.* [12] presented another GPU implementation of HMAX which was claimed to outperform the CNS implementation in both accuracy and speed. There are also many other examples in the literature of the application of GPU processing to object recognition [13]–[16].

Application-specific integrated circuits (ASICs) offer an even greater level of control than GPUs through intentional design of the hardware to suit the task at hand, but, once fabricated, an ASIC is typically ill suited to other applications. Furthermore, ASICs require a large design effort, a long time to implement (while waiting for fabrication), and come at high cost, which excludes them from use in many cases. Nevertheless, high performance still makes ASICs an attractive option for some tasks. An example of such work is the object recognition processor developed in [17] which can recognize up to 10 objects at a rate of 60 f/s at an image size of 640 $\times$ 480 pixels.

Field-programmable gate arrays (FPGAs) fall in the space between GPUs and ASICs in terms of time to implementation and level of control. FPGA hardware (fabric) is designed to be highly reconfigurable, thereby giving more control than with GPUs, but the hardware is already fabricated, thereby eliminating the time for fabrication which plagues ASICs. FPGAs also offer an advantage over GPUs in that they can operate in a stand-alone manner and interface directly with external sensors. A disadvantage of FPGAs is that their use often requires knowledge of a hardware descriptor language (such as Verilog or VHDL) which can be difficult to learn.

In an attempt to make FPGAs more accessible and user friendly, Impulse Accelerated Technologies Inc. [18] has developed a C-to-FPGA compiler to make FPGA acceleration more accessible to those not familiar with hardware design languages. A review of this and other C-to-FPGA approaches can be found in [19]. The E-lab at Yale is also working on easing the transition to FPGA with the development of "NeuFlow" [20], which is an FPGA-based system which can be programmed using the easier to learn Lua [21] scripting language. This approach significantly reduces time to implementation, but does not necessarily allow the user to fully exploit the performance capabilities of the FPGA. Despite being a valuable tool, the NeuFlow architecture is not well suited to implementing large filters (the original HMAX model requires filters up to 37×37 pixels in size). Other architectures for implementing HMAX on FPGA, developed in parallel with the work in this paper, have been recently published [22]–[27]. These implementations also show considerable speedup over GPU and CPU implementations. Most interesting of these works is one from Kestur *et al.* [23], which operates on higher resolution images (2352 $\times$ 1724 pixels) but uses a saliency algorithm to identify regions of interest, thereby obtaining further speedup by circumventing the need for an exhaustive search. Further discussion and comparison with these works can be found in Section VIII.

Despite the difficulties of learning hardware design languages, many other vision algorithms have also been implemented in FPGA, including the Lucas–Kanade [28] optical flow algorithm [29], SIFT [30], [31], SURF [32] spatiotemporal energy models for tracking [33] and segmentation [34], as well as bioinspired models of gaze and vergence control [35]. There are also many examples of neural networks (NNs) implemented in FPGA, including multilayer perceptrons [36], Boltzmann machines [37], and spiking NNs [38].

In this paper on multilayer perceptrons, Savich *et al.* [36] compared the use of fixed point and floating point representations for FPGA implementation and found that fixed point representation used less physical resources and fewer clock cycles, and allowed a higher clock speed than floating point representation while achieving similar precision and functionality. In this paper, fixed point representation is used throughout.

Himavathi *et al.* [39] described an NN implementation in FPGA which multiplexed resources for computation in different layers to reduce the total resources required at the expense of computation time. The ultimate aim was to use resources more effectively. In HMAX, cells differ by layer, so instead resources are multiplexed for different cells within the same layer. The ultimate aim is similar, namely to use resources as effectively as possible, thereby achieving maximum throughput with the available resources.

The computation performed by the first four layers of HMAX is task-independent, allowing us to easily estimate the required computation and allocate resources accordingly. The classifier, which follows the fourth HMAX layer, differs depending on the task (binary or multiclass), and in the case of multiclass, the required computation is further dependent on the number of classes (Section IV-A). To simplify implementation and maintain flexibility of the system, we implement the classification stage in the loop on a host PC. We show through testing in Section VII-F that implementing the classifier in the loop on a host PC does not affect the system throughput. Implementing a classifier in FPGA is nevertheless possible, as is evidenced by numerous examples of FPGA classifier implementations in the literature, including Gaussian mixture models [40], NNs [41], [42], naive Bayes [43], K-nearest neighbor [44], support vector machines (SVMs) [45], and even a core generator for generating classifiers in FPGA [46].

To remain consistent with previous work [6] and provide a fair comparison, a boosting classifier is used when performing binary classification, and a linear (SVM) classifier is used when performing multiclass classification. The use of linear SVM is further supported in [47], who did a comparison of multivariate classifiers in a visual object discrimination task using functional magnetic resonance imaging (fMRI) data from early stages of human visual and inferior temporal cortex. Linear classifiers were found to perform better than nonlinear classifiers, which they note is consistent with previous similar investigations [48], [49]. Misaki *et al.* also note that nonlinear classifiers may perform better if larger datasets are used for training, or if fewer features are used. Nonlinear classifiers can better fit the training data, but this comes with the risk of overfitting the classifier to the data, which is particularly problematic when only a few training samples are used.

The rest of this paper describes how the original model [6] was adapted for implementation on an FPGA to increase throughput and how these adaptations affect recognition accuracy. To test the FPGA implementation, we performed a binary classification task on popular categories from the commonly referenced and publicly available Caltech 101 [50] dataset as well as a tougher minaret dataset comprised of images downloaded from Flickr. We also investigated multiclass classification accuracy using Caltech 101. Results are compared to previously published test results on the same dataset using a software implementation of the HMAX model [6]. An analysis of how the image throughput rate and required hardware would change with input image size is also presented. The aim of this paper is not to beat the state of the art in terms of recognition accuracy, but rather to show how a given model can be adapted for implementation on an FPGA to drastically increase throughput while maintaining the same level of recognition accuracy.

II. Original Model Description

The version of the HMAX model used [6] has two main stages, each consisting of a simple and complex substage. We will call these Simple-1 (S1), Complex-1 (C1), Simple-2 (S2), and Complex-2 (C2) as is done in the original paper.

TABLE I
PARAMETERS USED IN FPGA IMPLEMENTATION OF HMAX

Size Band#	Subsampling Period Δ	Filter Sizes $\varnothing \times \varnothing$	σ	λ
Band 1	4	7×7	1.3	3.9
		9×9	1.7	5.0
Band 2	5	11×11	2.1	6.2
		13×13	2.5	7.4
Band 3	6	15×15	2.9	8.7
		17×17	3.3	10.0
Band 4	7	19×19	3.8	11.3
		21×21	4.2	12.7
Band 5	8	23×23	4.7	14.1
		25×25	5.2	15.5
Band 6	9	27×27	5.7	17.0
		29×29	6.2	18.5
Band 7	10	31×31	6.7	20.1
		33×33	7.2	21.7
Band 8	11	35×35	7.8	23.3
		37×37	8.3	25.0

Adapted from the parameter table shown in [6].

A. S1

In S1, the image is filtered at each location with Gabor filters applied at 16 different scales with the side length of a filter ranging from 7 to 37 pixels in increments of 2 pixels as shown in Table I. For each filter size, the filter is applied at four different orientations (0°, 45°, 90°, and 135°). For each filter position, the underlying image region is normalized before filtering to increase illumination invariance. The output of S1 consists of 64 filtered versions of the original image (16 scales $\times$ 4 orientations). The sign of the result is dropped and only the magnitude is passed to C1.

B. C1

Filter responses are grouped by filter sizes into eight size bands, as shown in Table I. Within each size band, the response of a C1 unit is the maximum of the S1 units in that size band over a small local spatial region ($2\Delta \times 2\Delta$ from Table I). The result is then subsampled (every Δ pixels) and output to S2. The output is therefore 32 sets of C1 units (8 size bands $\times$ 4 orientations).

C. S2

S2 units have as their inputs C1 units from all four orientations. They compute the Euclidean distance between a predefined patch and the C1 units at every location. The patch sizes are $4 \times 4 \times 4$, $8 \times 8 \times 4$, $12 \times 12 \times 4$, and $16 \times 16 \times 4$ ($x \times y \times$ orientation). For every S2 unit, the patch distance is computed at every (x, y) location within every size band and passed to C2.

D. C2

The C2 layer computes the minimum of the S2 distance for each patch across all locations in all size bands. The number of C2 outputs is therefore equal to the number of S2 patches used.

E. Classification

Classification is performed directly on the C2 outputs. The choice of classifier can vary on the basis of the required task. Reference [6] presented results using a boosting classifier for binary classification, and a linear SVM one-versus-all classifier for multiclass classification.

III. FPGA Implementation

A. Hardware Description

The large number of multiply accumulate (MAC) operations required to implement the 64 filters in S1 and the 1000 patches in S2 makes the number of multipliers available on an FPGA one of the limiting constraints for throughput. The second limiting constraint is the amount of internal memory available. We need to ensure that we have enough memory to store all intermediate results, S2 patches, and S1 filters since we can save time by not loading S1 filters and S2 patches from external memory, as will be shown in Section III-E. Multiple block RAMs are used in parallel whenever data wider than 16 bits need to be stored. We chose to use the Xilinx XC6VLX240T from the Virtex 6 family for its large number of multipliers (768) combined with its reasonable price of $1800 for a development board (Xilinx EK-V6-ML605-G board). The S1, C1, S2, and C2 stages were each implemented as separate modules in VHDL using a pipelined architecture.

B. Edge Effects

The most obvious way to speed up the model is to not waste resources on unnecessary computation. For this reason, we chose to only compute filter responses and patch distances when the filter (S1) or patch (S2) has full support. We effectively ignored any computation that involved regions beyond the image edges.

C. S1 Filters

The S1 layer consists of directionally selective Gabor receptive fields, similar to the selectivity of simple cells found by Hubel and Weisel [51] in V1. We implement cells at four different orientations (0°, 45°, 90° and 135°) as was done in the original model [6]. Because of symmetry, we need not compute cells at orientations at or above 180°. Each orientation is implemented at 16 different scales and at every location in the image where full support is available. The equations defining the filters used in the original HMAX model [6] are repeated in (1) for convenience. The equations for the filters are the product of a cosine function and a Gaussian weighted envelope

$$F_\theta(x,y) = e^{\left(-\frac{x_0^2+\gamma^2 y_0^2}{2\sigma^2}\right)} \times \cos\left(\frac{2\pi}{\lambda}x_0\right)$$
$$x_0 = x\cos\theta + y\sin\theta$$
$$y_0 = -x\sin\theta + y\cos\theta. \quad (1)$$

Here, λ determines the spatial frequency at the filter's peak response, σ specifies the radius of the Gaussian window, and γ squeezes or stretches the Gaussian window in the y_0-direction to create an elliptical window. For the 0° and 90° cases, we can easily rewrite this equation as product of two separate functions as shown in (2). The 45° and 135° terms are not separable unless we change the Gaussian weighting function to an isotropic function by specifying $\gamma = 1$. By doing this, we arrive at the equations for the 45° and 135° filters shown below

$$\begin{aligned}
F_0(x,y) &= E(x,y) * G(x,y)^T \\
F_{90}(x,y) &= E(x,y)^T * G(x,y) \\
F_{45}(x,y) &= E(x,y) * E(x,y)^T + O(x,y) * O(x,y)^T \\
F_{135}(x,y) &= E(x,y) * E(x,y)^T - O(x,y) * O(x,y)^T \\
E(x,y) &= e^{\left(\frac{-x^2}{2\sigma^2}\right)} \cos\left(\frac{2\pi x}{\lambda}\right) \\
G(x,y) &= e^{\left(\frac{-\gamma^2 x^2}{2\sigma^2}\right)} \\
O(x,y) &= e^{\left(\frac{-x^2}{2\sigma^2}\right)} \sin\left(\frac{2\pi x}{\lambda}\right). \quad (2)
\end{aligned}$$

Here, (x, y) is the location of the kernel value within the filter, $O(x, y)$ is an odd Gabor filter, $E(x, y)$ is an even Gabor filter, and $G(x, y)$ is a pure Gaussian filter. $A*B$ designates the convolution of A and B, while A^T designates the transpose of A. By writing the filters in a separable manner, we can implement them using two passes of a 1-D filter rather than one pass of a 2-D filter [52]. The number of MAC operations required to implement a separable filter grows linearly with the side length of the filter rather than as the square of the side length and therefore results in a significant speedup, or in the case of FPGA implementation, a significant saving of resources. If we consider the specific case of implementing the 64 S1 filters at a single image location, we can compute the number of MAC required using

$$\begin{aligned}
\text{MAC}_{\text{original}} &= 4 \times \sum_{i=1}^{16} [\varnothing(j)^2] = 36\,416 \\
\text{MAC}_{\text{separable}} &= 4 \times \sum_{i=1}^{16} [2 \times \varnothing(j)] = 2816 \quad (3)
\end{aligned}$$

where $\varnothing(j)$ is the side length of filter j as indicated in Table I and in (4).

Using separable filters reduces the number of required MAC from 36 416 down to 2816, a reduction to less than 8% of the originally required computation. Furthermore, each 1-D filter used has either even or odd symmetry about the origin, allowing us to sum values in the filter support either side of the origin before performing multiplication. By exploiting the symmetry of the filter, the required multiplications are reduced by a further 50%, freeing up more dedicated hardware multipliers for use in the more computationally intensive S2 stage of processing. The use of separable instead of nonseparable filters reduces the time taken to compute the S1 filter responses from 2.3 to 0.3 s per 128×128 image in MATLAB.

To increase illumination invariance, the filter response at each location is normalized by the l^2 norm of its support, as is done in the original model. This normalization ensures that filters capture information about the local contrast and

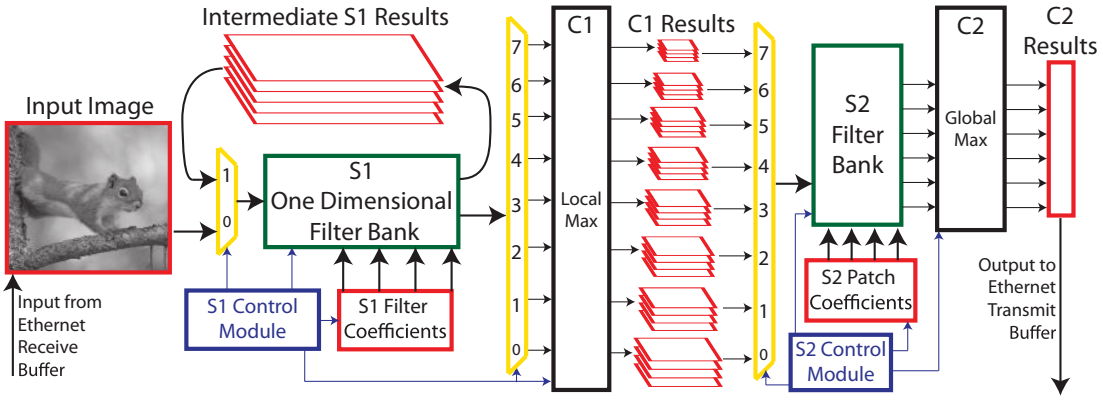


Fig. 1. Block diagram of our hardware implementation of HMAX. Red rectangles indicate the use of block RAM. Input to the 1-D S1 filters can come from either the input image or from RAM holding intermediate results. S1 results are sent to C1 where the maximum is computed over a local region (Table I) and stored. Each size band has its own dedicated RAM. A demultiplixer controls reading of C1 results for the S2 stage. C2 computes the global maximum of S2 outputs and stores the results in RAM before transferring them to the Ethernet transmit buffer.

are unaffected by the absolute brightness of a pixel region. The l^2 norm is computed by first summing the squares in the x-direction, then summing the result in the y-direction, and then taking the square root. We timed this result to be available simultaneously with the filter results so that we could immediately perform division without the need to store intermediate results. Responses for filters at all four orientations are computed in parallel, eliminating the need to recompute or store the l^2 norm of the filter support for each orientation.

The filter kernels are all precomputed and stored in a look-up table (Fig. 1). Each filter is modified to have zero mean and an l^2 norm of $(2^{16}-1)$ to ensure that results are always less than 16 bits wide. The parameters used for these separable filters is shown in Table I. These parameters can be written into equations as shown in (4)

$$\begin{aligned} \varnothing(j) &= 5 + 2 \times j \\ \Delta(b) &= 3 + b \\ \kappa(k) &= (4 \times k)^2 \end{aligned} \tag{4}$$

where j is an index for filter sizes arranged from the smallest to the largest (1–16). The diameter of filter j is $\varnothing(j)$. The filter is actually square with side length $\varnothing(j)$ to avoid the complexity of implementing a round filter. The subsampling period of size band b is written $\Delta(b)$. k is an index for the size of patches (1–4 for the four different patch sizes). At each orientation, a patch of size index k will have size $\kappa(k)$.

D. C1

The C1 layer requires finding the maximum S1 response over a region of $2\Delta \times 2\Delta$ and subsampling every Δ pixels in both x and y (for values of Δ, see Table I). We computed the maximum of a $2\Delta \times 2\Delta$ region by first computing the maximum over adjacent nonoverlapping regions of size $\Delta \times \Delta$. By taking the maximum across every four adjacent $\Delta \times \Delta$ regions, we obtained the maximum over a $2\Delta \times 2\Delta$ region, subsampled every Δ pixels in both x and y.

Computing on data as it streams from S1 eliminates the need to store nonmaximal S1 results (Fig. 1). As with the S1 layer, computation in C1 is performed on all four orientations in parallel. Each time C1 finishes computing the results for a size band, a flag is set that indicates to S2 that it can begin computation on that size band.

E. S2

Even though the data coming into S2 has already been reduced by taking the maximum across a local pool and subsampling in C1, the S2 layer is where most of the computation takes place. The number of MAC operations required to compute all patch responses at a single location in the original model is

$$250 \times 4 \times \sum_{k=1}^{4} \kappa(k) = 480\,000 \tag{5}$$

where there are 250 patches per size and 4 orientations per patch, each of size $\kappa(k)$, which was defined in (4). The computation of these patch responses must be repeated at all locations within all size bands.

We decided to use 1280 patches (320 per size), which was a compromise between the speed of implementation and the number of patches. As in the original model, S2 patches are obtained from previously computed C1 results on images from both the positive and negative classes. Since S2 patches are simply portions of previously computed C1 outputs, the number of bits required to store each patch coefficient is 16. The closeness of a patch to a C1 region is computed as the Euclidean distance between the patch and that region.

We computed patch responses starting with the smallest sized patches ($x \times y \times$ orientation $\rightarrow$ $4 \times 4 \times 4$) and computing their response at a single location. We then repeat this computation for all locations in the current size band, before moving onto the next patch size. Once all patch sizes have been computed for all locations in the current image size band, we move onto the next size band as soon as it is available from C1. All patches of the size currently being considered are computed in parallel. Furthermore, the response at two different orientations is considered in parallel. This results in $320 \times 2 = 640$ parallel MAC operations every clock

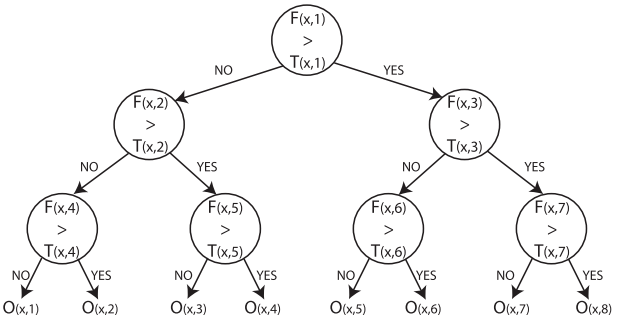


Fig. 2. Weak learner used in the gentle boosting algorithm. Each weak learner is a tree consisting of seven nodes. $F(x, y)$ represents the feature used at node y in weak learner x. $O_{(x,1)}$ through $O_{(x,8)}$ are the binary outputs of classifier x. Each output is a binary value 1 or -1.

cycle. This uses 640 multipliers and requires that 640 patch coefficients be read every clock cycle. Patch coefficients are stored in the FPGA's internal block RAM since the bandwidth to external RAM would not allow such high data rates. Using external RAM would require a data rate of 640×16 bits $\times$ 100 MHz $= 1$ Tb/s for a 10-MHz clock.

F. C2

C2 simply consists of a running minimum for each S2 patch, computed by comparing new S2 results with the previously stored S2 minimum. This is performed for all 320 S2 patches of the current size simultaneously (see Fig. 1).

G. Classifier

Results from [6] suggest that a boosting classifier is better than SVM for the binary classification problem. We used the gentle boosting algorithm [53] with weak learners consisting of tree classifiers, each with a maximum of three decision branches before reaching a result, as shown in Fig. 2. We used 1280 weak learners in the classifier, each computed in series.

For multiclass classification, a linear one-versus-all SVM classifier was chosen [54], [55]. This is a simple linear classifier, but is memory-intensive in its requirement for storing coefficients, as is discussed in Section IV-A.

In order to not restrict the FPGA implementation to only binary problems or only multiclass problems, the classifier was implemented separately on a host PC.

H. Scheduling

The FPGA implementation has an input first in-first out (FIFO) buffer capable of holding up to four complete 28×128 pixel images. As soon as at least one full image has been loaded into the buffer, S1 will read the image. S1 then computes responses at all four orientations for the smallest filter simultaneously and outputs the results in a streaming fashion to C1. After computing the responses from the smallest filter, S1 filters will read in coefficients for the next filter size and compute the new filter responses. S1 will continue in this manner until responses for all filter sizes have been computed. S1 will read a new image from the input buffer as soon as it has completed the first pass with the largest separable filter, or as soon as an image becomes available if none is available at the time.

The C1 and C2 layers operate on the results of S1 and S2 as they are output in a streaming fashion during computation, thereby reducing the internal memory required to store intermediate results. This approach also ensures that C1 and C2 only add a negligible amount of processing time to the algorithm (less than 100 μs for an entire image).

Each stage (S1, C1, S2, C2) uses its own dedicated FPGA resources, thereby allowing all stages to run simultaneously. Sharing of memory occurs between C1 and S2, where access is managed by setting and clearing flags. There is a separate memory unit and flag for each image band. When a flag is low, C1 has exclusive read/write access to the corresponding memory unit. Once C1 has finished storing results in the memory unit, it will set the corresponding flag high. When a flag is high, the S2 stage has exclusive read/write access to the corresponding memory block and will clear the flag once it has finished processing all data from that memory block, thereby transferring control back to C1.

If waiting for access to a particular memory block, a stage (C1 or S2) will begin processing as soon as access is granted (the very next clock cycle). Since results for each image band are stored separately, the S1 and C1 stages can process the next image band (and loop around) without having to wait. This allows S1 and C1 to be almost an entire image ahead in computation than the S2 stage, which is important because, although the S1 and C1 stages take the same length of time to process each image band, the time taken by S2 varies. The S2 stage takes longer to compute on smaller image bands because their higher frequency of subsampling produces more C1 results on which computation must be performed (see Table I). Buffering of C1 outputs in the manner described allows us to focus on matching the throughput of the S1 and C1 stage with the average throughput (across image bands) of the S2 stage, without being troubled by how computation time in S2 varies with each image band.

S1 will not compute new results for an image band if the current results for that image band (from the previous image) have not yet been processed by S2 (i.e., if the relevant memory flag is still high). S1 will, however, still perform the first pass with a separable filter in the meanwhile to ensure that it can start outputting results as soon as the flag is cleared.

Results from S2 stream to C2, which writes the final results to an output buffer for communication back to the host PC.

IV. Scalability of FPGA Implementation

In this section we show how the input image size affects the hardware resources and time required for computation using the FPGA implementation described in Section III. The described FPGA implementation was specifically designed to operate on images of size 128×128 pixels and is therefore not necessarily recommended as the best implementation for larger or smaller images. Nevertheless, if implementing a new design to operate on larger (or smaller) images, extrapolating the current design to different sizes provides a good starting point.

A. Hardware Resources

The number of bits in the counters used to track the progress of computation on the input image and intermediate results in stages S1, C1, and S2 will need to increase to handle larger images. This increase scales as

$$\text{Counter Bits} \propto \log_2 \sqrt{N} \tag{6}$$

where N is the number of pixels in the input image, and the image is assumed to be square, having side length $\sqrt{N}$. This increase in the required hardware is negligible, especially in comparison to the increase in internal RAM required to store the input image and intermediate results in the S1 and C1 stages. The internal RAM requirement scales proportionally to N for large images. Because of the nature of computation in S2 and C2, no additional RAM is required in those stages when the image size increases. The number of elements required to compute multiplication, addition, division, and square roots remains unchanged in all stages. The total required internal RAM is the sum of the RAMs required by all stages.

Internal RAM is required for three purposes in S1: storing the input image, storing intermediate results between the first and second passing of the separable filter, and, finally, storing the S1 filter coefficients. The required RAM can be explicitly calculated using (7)

$$\begin{aligned} S1_{\text{bits}} &= S1_{\text{input}} + S1_{\text{intermediate}} + S1_{\text{filters}} \\ S1_{\text{input}} &= 4 \times N \times 8 \\ S1_{\text{intermediate}} &= 5 \times N \times 23 \\ S1_{\text{filters}} &= \sum_{j=1}^{16} (2 \times (3+j) \times 16). \end{aligned} \tag{7}$$

N represents the number of pixels in the input image. The input buffer has to hold four images (a FIFO buffer) with 8 bits per pixel. The intermediate results require five buffers (one for each orientation and one for calculating the l^2 norm of the filter support). Each result consists of 23 bits. For storage of the filters, the jth filter (ordered smallest to largest) consists of two separable filters, each with $(3 + j)$ coefficients and 16 bits per coefficient.

The output of the S1 stage does not require RAM for storage since each result is processed by C1 as soon as it becomes available, but C1 does require RAM for intermediate and final results. The RAM required by C1 can be explicitly calculated using (8)

$$\begin{aligned} C1_{\text{bits}} &= \sum_{b=1}^{8} C1_{\text{size}}(b) \times 16 \\ C1_{\text{size}}(b) &= \frac{S1_{\text{size}}(b)}{\Delta(b)^2} \\ S1_{\text{size}}(b) &= 4 \times (\sqrt{N} - \varnothing(2b) + 1)^2. \end{aligned} \tag{8}$$

The number of valid S1 results in image band b is then given by $S1_{\text{size}}(b)$, where $\varnothing(2b)$ was previously defined in (4) and there are four orientations. The number of C1 results can then be calculated knowing the number of S1 results and the subsampling period $\Delta(b)$, which was also previously defined in (4). Each C1 result occupies 16 bits.

The RAM required for S2 is constant across image sizes and can be written explicitly as

$$S2_{\text{bits}} = \sum_{k=1}^{4} 320 \times 4 \times \kappa(k) \times 16 \tag{9}$$

where k is an index of patch size. There are 320 patches per size and four orientations per patch, each with $\kappa(k)$ coefficients as previously defined in (4). Each coefficient occupies 16 bits.

C2 requires only enough RAM to hold the final C2 results.

$$C2_{\text{bits}} = 1280 \times 42 \tag{10}$$

where there are 1280 C2 features each consisting of 42 bits.

Although we implement the classifier on the host PC, it is possible to determine the resources required by the classifier. The most memory intensive classifier used in this paper is the 102 class one-versus-all linear SVM classifier, for which the memory requirements are as follows:

$$\begin{aligned} \text{Classifier}_{\text{bits}} &= 102 \times 1280 \times 32 + 84 \\ &= 4\,178\,004 \text{ bits} \end{aligned} \tag{11}$$

where there are 102 possible classes, 1280 C2 features, 32 bits per coefficient, and up to 84 bits required to hold the result. The current FPGA implementation does not have enough remaining internal memory to hold all these coefficients, but the coefficients could easily fit into external RAM, or the classifier could be run on a second FPGA. If running at 190 images per second, an external memory bandwidth of $102 \times 1280 \times 32 \times 190 = 794$ Mb/s would be required, which is only about 6% of the available 12.8 Gb/s bandwidth on the targeted FPGA platform. In our implementation, running the classifier on a host PC did not affect the system throughput.

B. Time

The time taken to process an image is dominated by the S1 and S2 stages. The C1 and C2 stages perform simple maximum operations on each valid data point as it becomes available, and therefore do not contribute significantly to the time taken to process an image. The time computed in the equations below is in units of clock cycles and the actual time taken for computation therefore depends on the FPGA clock frequency.

The time taken to compute S1 can be accurately approximated as the time required to do two passes of the image for each of the 16 separable filter sizes (12). All four orientations are simultaneously computed in parallel and therefore the multiple orientations do not add to the computation time.

$$S1_{\text{time}} = 2 \times N \times 16 \tag{12}$$

where $S1_{\text{time}}$ is in units of clock cycles, N is the number of pixels per image, and 16 filter sizes are implemented.

In S2, all 320 patches of the same size are considered simultaneously and, within each patch, computation is performed at two orientations simultaneously.

$$\begin{aligned} S2_{\text{time}} &= \sum_{b=1}^{8} \sum_{k=1}^{4} S2_{\text{size}}(b,k) \times \kappa(k) \times 2 \\ S2_{\text{size}}(b,k) &= (\sqrt{C1_{\text{size}}(b)} - \sqrt{\kappa(k)} + 1)^2 \end{aligned} \tag{13}$$

where $S2_{\text{size}}(b,k)$ is the number of valid S2 results for size band b and patch size index k. $S2_{\text{size}}(b,k)$ is zero whenever the size of the C1 results is smaller than the patch size, i.e., when $C1_{\text{size}}(b) < \kappa(k)$. $\kappa(k)$ is the patch size and was previously defined in (4). $S2_{\text{time}}$ is the total time (in clock cycles) taken to compute all patch responses of all sizes in every size band.

If the multiclass one-versus-all linear SVM classifier were to be implemented on the FPGA with 102 classes and only a single hardware multiplier, the time taken could be computed as

$$\text{Classifier}_{\text{time}} = 1280 \times 102 \tag{14}$$

for 1280 C2 features and 102 classes. The time taken for classification would not be dependent on the input image size. Using a single multiplier would enable a throughput of up to 765 images per second when using a 100-MHz clock.

V. Simulation

Four different sets of code were used in simulation. The first is a MATLAB implementation of the HMAX model which was retrieved from the HMAX website [56]. This was used as a benchmark against which to compare our modified implementation of HMAX for FPGA to verify that the modifications made did not severely compromise recognition accuracy. We refer to this original HMAX implementation as HMAX CPU.

The second, third, and fourth sets of code are MATLAB, C++, and VHDL implementations, respectively, of our modified version of HMAX for FPGA. These implementations are functionally equivalent and we refer to them as "HMAX FPGA." The MATLAB code was used to make initial changes to the model and test accuracy on small datasets. Once satisfied with the changes made, a faster C++ implementation was written and used to verify the modified model on larger datasets. Finally, the actual VHDL code required to implement the proposed model in FPGA was written. This VHDL code was used to determine possible clock speeds and image throughput as well as to verify that the proposed FPGA model could be implemented using the resources available on the targeted FPGA platform (Xilinx Virtex 6 XC6VLX240T). Both final and intermediate results from the modified MATLAB, C++, and VHDL codes were compared to verify that all three were performing the same computation.

VI. Hardware Validation

The results of simulation were verified through implementation on the Xilinx Virtex 6 ML605 development board. A C++ interface was written for the host PC which handles Ethernet communications with the ML605 board and performs classification. The C++ code transmits four images to the ML605 board to fill the input buffer (described in Section III-H), and then waits for all 1280 C2 values from an image to be returned before transmitting the next image. Reading of images from the hard drive and classification are both performed while waiting for the next set of C2 values from the FPGA, thereby adding negligibly to the overall computation time. Classification results are written to an output file as they are computed. For further verification, C2 results from FPGA could be optionally written to disk for direct comparison against simulated C2 results.

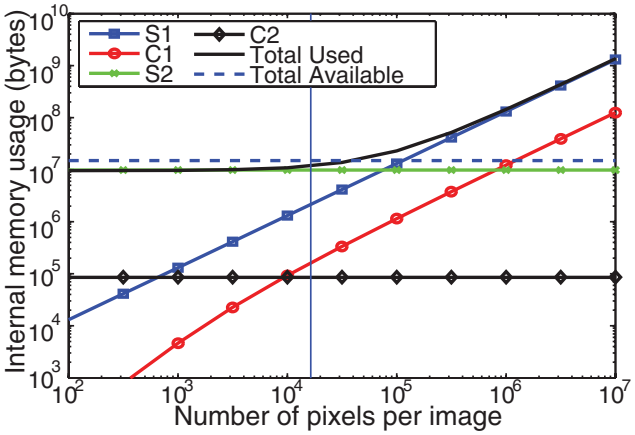


Fig. 3. Theoretical analysis of the internal RAM required by each stage as well as the total RAM required and total RAM available on the selected FPGA. The vertical line shows the number of pixels in a 128 × 128 image, for which this implementation was designed.

VII. Results

A. FPGA Code Analysis

Using the Xilinx ISE, the VHDL code for implementing HMAX on FPGA was analyzed. For simplicity, we use a single clock for all stages within the model. All look-up tables, S1 filters, and S2 patches, as well as all intermediate results, are stored in internal block RAM, as shown in Fig. 1. The system has a latency of 600 000 clock cycles when processing a single image, but can maintain a throughput of an image every 526 000 clock cycles. Implementation of the full model indicates that the design can run at a clock frequency of 100 MHz (10-ns period). A 100-MHz clock results in a latency of 6 ms for processing a single image and a maximum throughput of 190 images per second when processing multiple images. These figures are achieved assuming that the input figure is a 128 × 128 pixel 8-bit per pixel grayscale image. The throughput of the design is determined by the throughput of the slowest stage in the pipeline. Computational resources should therefore be allocated in such a way that all stages have roughly the same throughput. This has been done as is evident in the distribution of multipliers between the S1 and S2 stages. S1 is the slowest stage, limiting the throughput to 190 images per second using 77 multipliers at 100-MHz clock frequency, while S2 is capable of a throughput of 193 images per second, but uses 640 multipliers.

If we were to create an optimal implementation of S1 using nonseparable filters with a 100-MHz clock, then S1 alone would require over 1600 multipliers to achieve the same throughput of 190 images per second (unless a scale space approach was adopted). This is over double the number of hardware multipliers available on the chosen FPGA.

Table II shows the total resources used by the HMAX implementation.

TABLE II
FPGA RESOURCES USED BY HMAX

Resource	Used	Available	% Used
Multipliers (DSP48E1)	717	768	93
Internal RAM (RAM36E1)	373	416	89
Slice registers	66 196	301 440	21
Slice LUTs	60 872	150 720	40

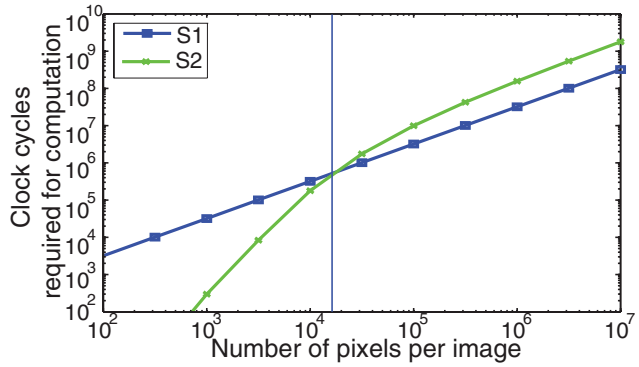


Fig. 4. Theoretical analysis of the time taken to compute each stage of HMAX in the current architecture. Because of the pipelined nature of the computation, the rate at which images can be processed is limited by the stage that takes the longest time. The vertical line shows the number of pixels in a 128 × 128 image, for which this implementation was designed. The time required to compute the two longest stages is equal at this point as a result of the effort to allocate resources in such a way as to maximize throughput.

B. Scalability

Fig. 3 shows the internal RAM requirements computed using the equations presented in Section IV-A, as well as the total block RAM available on the selected Virtex 6 FPGA (14 976 kb, dashed line) and the image size for which the algorithm was designed (128 × 128 pixels, vertical line). Since all S2 patches of the same size are computed in parallel, the number of patches does not affect computation time, but will be limited by the number of available multipliers and the amount of RAM available (see Table II).

The time taken to compute the S1 and S2 stages is shown in Fig. 4 along with the number of pixels for which the current implementation was designed (vertical line). The throughput of the complete system is limited to the throughput of the slowest stage.

The time taken to compute S2 can be seen as the time that would be taken to compute all results (even partial results on edges) minus the time that is saved by not computing edge results. The time saved by not computing at edges is significant at an image size of 128 × 128. The time saved grows proportionally to the side length of the image $\sqrt{N}$, which is much less than the time to compute all results (which grows linearly with N). This is why the time for S2 grows linearly with N only for large N. S1 always grows linearly with N.

The design of the current framework ensures that the time taken for S1 and S2 is roughly equal (within 2%) for images of size 128 × 128, thereby ensuring that computational resources in each stage are not sitting idle waiting for the other stage to finish computing. If working with images of a different size, resources would ideally be reallocated to ensure that S1 and S2 still take equal time.

TABLE III
COMPARISON OF RECOGNITION ACCURACIES OBTAINED FROM ORIGINAL HMAX CODE AND FPGA IMPLEMENTATION ON POPULAR CATEGORIES IN CALTECH 101

Category	HMAX [6]	HMAX CPU	HMAX FPGA
Airplanes	96.7	97.1	98.2
Cars	99.7	99.3	99.2
Faces	98.2	95.8	96.4
Leaves	97.0	94.6	93.7
Motorbikes	98.0	98.3	98.8

C. Caltech 101 Binary Classification

Two datasets were used to test the recognition accuracy of our modified HMAX model. The first is the often referenced Caltech 101 dataset [50]. Recognition accuracy of popular categories in this dataset was presented for the HMAX model in [6]. We ran our own binary classification simulations on these categories using both the downloaded and modified versions of HMAX. The binary task constituted discriminating the class in question (airplanes, cars, faces, leaves, or motorbikes) from the background class. In each case, half the images from the class in question and half images from the background class were used for training. The remaining images from both the class in question and the background class were used for testing. In each case, 10 trials were run. The accuracy reported in Table III is the percentage of correct classifications at the point on the receiver operating characteristic curve (Fig. 5) where the false positive and false negative rates are equal. Looking at the mean accuracy for this metric, the FPGA implementation achieves 0.24% higher accuracy than the original CPU implementation. This shows that the modifications made for the FPGA implementation have not adversely affected recognition accuracy.

D. Binary Classification on Flickr Dataset

The binary Minaret classification task was performed on a dataset containing 662 images of Minarets and 1332 background images. The Minaret (positive) images were obtained from Flicker by searching for "Minaret", while the negative images were obtained by periodically downloading the most recently uploaded Flicker image. Examples of these images are shown in Fig. 6. Ten random splits were used for classification and testing, with the test set consisting of 1000 negative and 500 positive images. The remaining images constitute the training set. This test was performed with both the downloaded HMAX code and the modified HMAX code for FPGA. The results are shown in Table IV. The metric used is the percentage of correct classifications at the point where false positive and false negative rates are equal. As expected, using 2000 features instead of 1280 improves the accuracy for both the

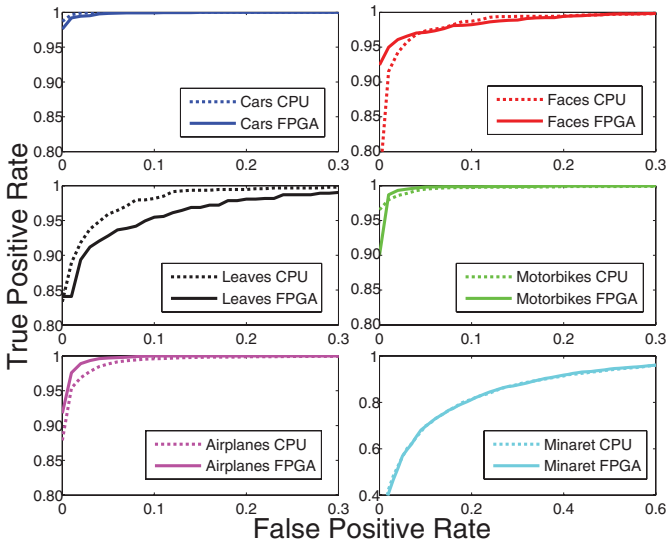
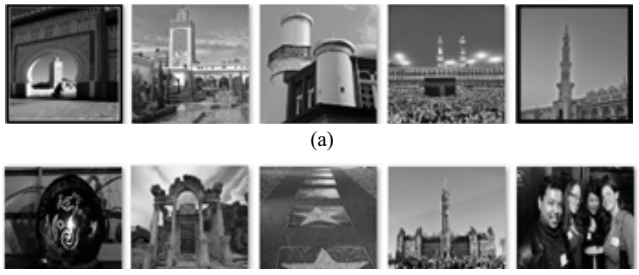


Fig. 5. Receiver operating characteristics for the binary classification task on Caltech 101 popular image categories and Minaret datasets. Each curve is the result of a mean over 10 trials. Note that the true positive rate axis is different for the Minaret classification task.



(a)

(b)

Fig. 6. (a) Sample of images from the Minaret. (b) Background classes used in the Minaret binary classification task.

TABLE IV
COMPARISON OF RESULTS OBTAINED FROM ORIGINAL HMAX CODE AND FPGA IMPLEMENTATION ON MINARET CLASSIFICATION TASK

Model	HMAX CPU	HMAX CPU	HMAX FPGA	HMAX FPGA
Features	2000	1280	2000	1280
Accuracy	82.9	82.2	82.2	81.3

CPU and FPGA implementations. The accuracy of the FPGA implementation is within 1% of that of the original model.

E. Caltech 101 Multiclass One-Versus-All

A second test using the Caltech 101 database is the multiclass one-versus-all test. For this, we used 15 training examples per category, as was done in [6]. Testing was performed using 50 examples per category or as many images as remained if fewer than 50 were available. Each of the categories was weighted such that it contributed equally to the result, as was done in [6]. This is a 102-category problem including the background category. Using the one-versus-all linear SVM multiclass classifier from [55], we achieved a mean accuracy of 47.2% $\pm$ 1.0% over 10 trials, which is in agreement with the result of 44% $\pm$ 1.14% reported in [6] for the same task.

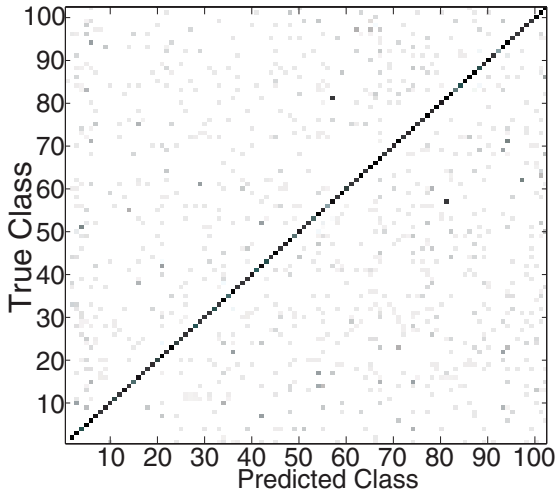


Fig. 7. Confusion matrix averaged over 10 trials for the 102-category multiclass one-versus-all test performed on the Caltech 101 database. The low accuracy on the extreme bottom left of the diagonal is the background category. The largest confusion is between the schooner (81) and ketch (57) categories, which are similar cases of sailboats.

TABLE V
MAXIMUM THROUGHPUT FOR EACH STAGE IN IMAGES (S)

Stage	Input Buffer	S1	C1	S2	C2
Throughput	6100	190	552	193	10000

The slight increase in accuracy can be attributed to the fact that our FPGA implementation uses 1280 features compared to 1000 features used in [6]. The confusion matrix for the 101 multiclass one-versus-all problem is shown in Fig. 7.

F. Hardware Validation

Results from analyzing the VHDL code were verified by implementing the code on the ML605 board and processing the Caltech 101 database. The entire dataset consisting of 9144 images was processed 10 times in different trials. The time taken to complete processing was measured from when the first image is read from disk until the last classification result is written to disk. The time taken to process the entire Caltech 101 database was measured as 48.12 s $\pm$57 μs, which is a throughput of 190 images/s and agrees with VHDL simulation predictions (shown in Table V) to within 0.01%. Accuracy of the VHDL implementation was also verified against simulations. Both classification results and C2 outputs from testing were verified against simulation and found to exactly match.

G. Comparison With Other Approaches

To the best of our knowledge, 190 images/s is the fastest reported implementation of this version HMAX. Direct comparisons with other versions are not always straightforward because both the number of patches and their sizes can vary, as well as the size of the input image or even the model itself.

In 2010, Sedding *et al.* [12] presented a time of 86.4 ms for 4075 patches using custom code on an NVIDIA GeForce285 GTX. They used sparse features as proposed by Mutch and

Lowe [57] and claimed a shorter runtime than both the FHLib [57] and the GPU-based CNS [11]. In our aim to recreate the original model, we chose not to use sparse features; but the use of sparse features would allow us either a 4× speedup or it would allow us to implement 4× as many patches at the same speed (resulting in 5120 patches) on the ML605 board. Their implementation also operates on larger images, with shortest side measuring 140 pixels. If our 1280 dense patch implementation was to run on an image measuring 140×186 pixels (assuming a 3×4 aspect ratio), it would still take under 12 ms to complete.

On Caltech 101 with 15 training and 50 test samples per category, our 1280-patch 128×128 pixel model achieves an accuracy of 47.2% (see Section VII-E) whereas Sedding [12] achieves 37%, most likely a result of using sparse features. In terms of speed, our implementation takes 5.3 ms whereas theirs takes 86.4 ms. They can reduce their processing time to 8.9 ms if they only compute 240 patches, but this will come at the expense of even lower accuracy (less than 30% on the same task).

VIII. Discussion

The previous section showed that a massive increase in throughput can be achieved with almost no change in recognition accuracy. In this paper, the aim has been to achieve a very high throughput as an argument for the use of FPGA in hierarchical models, but one could just as easily trade speed for accuracy. Interestingly, our FPGA implementation of HMAX uses more S2 patches (1280) than the 1000 used in [6]. This increase in the number of patches was implemented simply because the additional resources required for the patches were available, and the parallel processing of patches means that, as long as resources are available, adding more patches does not affect throughput.

The issues of image acquisition, rescaling, and conversion to grayscale are not tackled by the current model since these will be application-specific. The model requires that images are prescaled to 128×128 pixels and converted to an 8-bit grayscale before they are processed. The FPGA model requires an input image in the form of raw pixel values. For 190 images per second, this translates to just over 3 MB of data per second, which is well within the capabilities of the evaluation board's PCI express or gigabit Ethernet interfaces, as has been verified through testing in Section VII-F. If using a laptop, the system can run over gigabit Ethernet, allowing it to be portable as shown in Fig. 8.

The HMAX model used in this paper is one that was freely available in easy-to-follow MATLAB code. It does not represent the least computationally intensive or most accurate version of the HMAX model. The creators of the model are continuously working on improvements, and a number of newer iterations have been presented [57]. One of the most significant changes is the use of a scale-space approach such that the image is rescaled and reprocessed multiple times by filters of a single fixed size rather than keeping the image the same size and using multiple filters of varying size. Many recent implementations [22]–[27] make use of 12 orientations instead of 4, which increases accuracy although it comes at the expense of extra computation time.

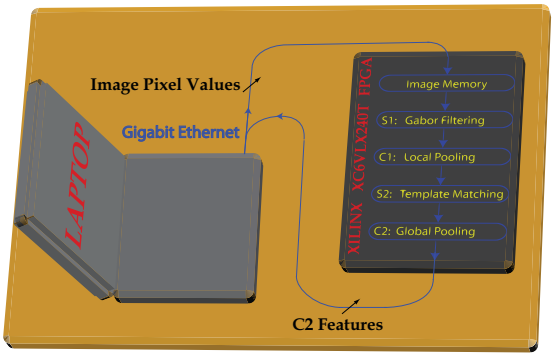


Fig. 8. Portable hardware setup for the binary classification system showing a laptop communicating pixel values over gigabit Ethernet to a Xilinx ML605 evaluation board containing the Xilinx Virtex 6 XC6VLX240T FPGA on which the HMAX model runs. C2 features are returned to the laptop via the same gigabit Ethernet interface.

We achieved a key speedup in the S1 layer by exploiting the known structure of filters, which allowed us to implement the Gabor filters as separable. The unsupervised learning in S2 means that its structure is not known *a priori*. If the model were changed to S2 patches of a known structure which could be similarly exploited, then further significant speedups could be achieved, but the effect on recognition accuracy would have to be further investigated.

Another change that greatly reduces computational complexity is the use of sparse S2 patches as proposed by Mutch and Lowe [57]. In their model, only the S1 orientation with maximal response is considered at each image location, thereby reducing the number of orientations in S2 from 4 to 1, which reduces the number of required MAC to only a quarter of the original. These sparse S2 features are used in most recent works [22]–[27]. The effect on throughput of using sparse versus dense features, and of changing the number of orientations from 4 to 12, can be found in [26]. Despite running on four FPGAs, each of which is more than twice as large as our FPGA (Virtex 6 SX475T versus LX240T), their dense implementation of HMAX using four orientations runs at roughly 45 images per second. However, there are certain differences: they operate on larger images (256×256 versus 128×128), and use more patches (4075 versus 1280). Using four FPGAs, we could run four copies of our model in parallel, each with different patches, thereby giving us $1280 \times 4 = 5120$ patches while maintaining throughput of 190 images per second. We also use an equal number of patches of each size, whereas more recent approaches typically use smaller (4×4) and less large (16×16) patches to reduce computation. To summarize, in comparison with the one in [26], we could implement more patches (5120 versus 4075), with a higher percentage of large patches, and a 4× higher throughput if four FPGAs were used. Their implementation uses significantly larger FPGAs than ours (containing 2016 versus 768 multipliers), but also operates on 4× larger images, making a direct comparison difficult.

A common bottleneck for parallel architectures lies in the

available bandwidth to memory and structuring how memory is accessed. For example, if two cores simultaneously request data from the memory, one will have to wait for the other before it can access the memory. In the presented FPGA implementation, this was overcome by using the internal block RAM of the FPGA, which resulted in a bandwidth of over 1 Tb/s, which could be difficult to maintain on other platforms. Other implementations of HMAX that have recently been published also make use of internal block RAM to overcome this memory access bottleneck [22]–[27].

The size of the current filters and patches are designed to operate on small images. Even if higher resolution images are available, they should be rescaled to 128×128 if they are to be processed with the current filters and patches. Nevertheless, extension to larger images is possible. Scalability of the current implementation has been presented, which shows that larger images can be processed on the current FPGA with minor adjustments but will ultimately be limited by the amount of internal memory available for buffering images and storing intermediate results. To overcome this, one could use a larger FPGA, use multiple FPGAs operating in parallel, reduce the number of S2 patches to free up memory, or change the model to use sparse features.

To provide a fair comparison with the original HMAX model, we used the same classifiers (boosting for binary and linear one-versus-all SVM for multiclass). Linear SVM classifiers remain the top choice for most HMAX implementations because of their computational simplicity and speed. The choice of linear SVM classifiers is also supported by other work on discriminating between visual objects based on fMRI recordings of early stages of the visual cortex [47]–[49]. In our implementation, we were able to run the classifier in the loop on a host PC without affecting the system throughput because classification was performed in parallel with feature extraction for the next image. Nevertheless, various classifiers can and have been implemented in FPGA [40]–[44], including SVM [45], and even a core generator for parameterized generation of your own classifier in FPGA [46].

Comparison with other approaches shows that this is currently the fastest complete HMAX implementation and outperforms reported CNS [11] and custom [12] GPU implementations, as well as many FPGA implementations, although direct comparison with other FPGA implementations is not always possible. As more powerful GPU platforms become available, these GPU implementations will achieve even better results, but the same can be said for FPGAs. The platform we have used (Xilinx Virtex 6 XC6VLX240T) is only in the middle of the range of the Virtex 6 family and is an entire technology generation behind the currently available Virtex 7 family.

IX. Conclusion

We showed how a neuromorphic bio-inspired hierarchical model of object recognition can be adapted for high-speed implementation on a mid-range COTS FPGA platform. This implementation had a throughput of 190 images per second, which is the fastest reported for a complete HMAX model. We performed binary classification tests on popular Caltech 101 categories as well as on a more difficult Flickr dataset to show that adaption for FPGA does not have a significant effect on recognition accuracy. We also showed that accuracy is not compromised on a multiclass classification task using Caltech 101.

References

[1] D. Lowe, "Object recognition from local scale-invariant features," in *Proc. 7th IEEE Int. Conf. Comput. Vision*, vol. 2. 1999, pp. 1150–1157.

[2] H. Bay, T. Tuytelaars, and L. Van Gool, "SURF: Speeded-up robust features," *Comput. Vis. Image Understand.*, vol. 110, no. 3, pp. 346–359, 2008.

[3] N. Logothetis, J. Pauls, and T. Poggio, "Shape representation in the inferior temporal cortex of monkeys," *Current Biol.*, vol. 5, no. 5, pp. 552–563, 1995.

[4] M. Riesenhuber and T. Poggio, "Are cortical models really bound by the 'binding problem'?" *Neuron*, vol. 24, no. 1, pp. 87–93, 1999.

[5] K. Jarrett, K. Kavukcuoglu, M. Ranzato, and Y. LeCun, "What is the best multi-stage architecture for object recognition?" in *Proc. 12th IEEE Int. Conf. Comput. Vis.*, Oct. 2009, pp. 2146–2153.

[6] T. Serre, L. Wolf, S. Bileschi, M. Riesenhuber, and T. Poggio, "Robust object recognition with cortex-like mechanisms," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 29, no. 3, pp. 411–426, Mar. 2007.

[7] T. Serre, A. Oliva, and T. Poggio, "A feedforward architecture accounts for rapid categorization," *Proc. Nat. Acad. Sci.*, vol. 104, no. 15, pp. 6424–6429, 2007.

[8] S. Thorpe, D. Fize, and C. Marlot, "Speed of processing in the human visual system," *Nature*, vol. 381, no. 6582, pp. 520–522, Jun. 1996.

[9] J. Mutch and D. G. Lowe, "Multiclass object recognition with sparse, localized features," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2006, pp. 11–18.

[10] S. Chikkerur. (2008). *CUDA Implementation of a Biologically Inspired Object Recognition System* [Online]. Available: http://code.google.com/p/cbcl-model-cuda/

[11] J. Mutch, U. Knoblich, and T. Poggio, "CNS: A GPU-based framework for simulating cortically-organized networks," Center for Biol. Comput. Learning, Massachusetts Institute of Technology, Cambridge, MA, USA, Tech. Rep. MIT-CSAIL-TR-2010-013, Feb 2010.

[12] H. Sedding, F. Deger, H. Dammertz, J. Bouecke, and H. Lensch, "Massively parallel multiclass object recognition," in *Proc. Vis., Model., Visualizat. Workshop*, 2010, pp. 251–257.

[13] J. Kim, E. Park, X. Cui, H. Kim, and W. Gruver, "A fast feature extraction in object recognition using parallel processing on CPU and GPU," in *Proc. IEEE Int. Conf. Systems, Man Cybern.*, Oct. 2009, pp. 3842–3847.

[14] S. Warn, W. Emeneker, J. Cothren, and A. Apon, "Accelerating SIFT on parallel architectures," in *Proc. IEEE Int. Conf. Cluster Comput.*, Sep. 2009, pp. 1–4.

[15] R. Uetz and S. Behnke, "Large-scale object recognition with CUDA-accelerated hierarchical neural networks," in *Proc. IEEE Int. Conf. Intell. Comput. Intell. Syst.*, vol. 1. Nov. 2009, pp. 536–541.

[16] M. Ebner, "A real-time evolutionary object recognition system," in *Genetic Programming* (Lecture Notes in Computer Science), vol. 5481. Berlin, Germany: Springer-Verlag, 2009, pp. 268–279.

[17] J. Kim, M. Kim, S. Lee, J. Oh, K. Kim, and H. Yoo, "A 201.4 GOPS 496 mW real-time multi-object recognition processor with bio-inspired neural perception engine," *IEEE J. Solid-State Circuits*, vol. 45, no. 1, pp. 32–45, Jan. 2010.

[18] *Impulse Accelerated Technology Website*. (2008). Impulse Accelerated Technology Inc., Bellevue, WA, USA [Online]. Available: http://www.impulseaccelerated.com

[19] B. Holland, M. Vacas, V. Aggarwal, R. DeVille, I. Troxel, and A. D. George, "Survey of C-based application mapping tools for reconfigurable computing," in *Proc. 8th Int. Conf. Military Aerosp. Program. Logic Devices*, Sep. 2005, pp. 1–21.

[20] C. Farabet, B. Martini, P. Akselrod, S. Talay, Y. LeCun, and E. Culurciello, "Hardware accelerated convolutional neural networks for synthetic vision systems," in *Proc. IEEE Int. Symp. Circuits Syst.*, Jun. 2010, pp. 257–260.

[21] R. Ierusalimschy, L. de Figueiredo, and W. Celes, *Lua 5.1 Reference Manual*. Rio de Janeiro, Brazil: Roberto Ierusalimschy, 2006.

[22] A. Maashri, M. DeBole, M. Cotter, N. Chandramoorthy, Y. Xiao, V. Narayanan, and C. Chakrabarti, "Accelerating neuromorphic vision algorithms for recognition," in *Proc. 49th IEEE Design Autom. Conf.*, Jun. 2012, pp. 579–584.

[23] S. Kestur, M. S. Park, J. Sabarad, D. Dantara, V. Narayanan, Y. Chen, and D. Khosla, "Emulating mammalian vision on reconfigurable hardware," in *Proc. 20th IEEE Annu. Int. Symp. Field-Program. Custom Comput. Mach.*, May 2012, pp. 141–148.

[24] M. DeBole, Y. Xiao, C.-L. Yu, A. Maashri, M. Cotter, C. Chakrabarti, and V. Narayanan, "FPGA-accelerator system for computing biologically inspired feature extraction models," in *Proc. 45th Asilomar Conf. Signals, Syst. Comput.*, Nov. 2011, pp. 751–755.

[25] A. Al Maashri, M. DeBole, C.-L. Yu, V. Narayanan, and C. Chakrabarti, "A hardware architecture for accelerating neuromorphic vision algorithms," in *Proc. IEEE Workshop Signal Process. Syst.*, Oct. 2011, pp. 355–360.

[26] M. S. Park, S. Kestur, J. Sabarad, V. Narayanan, and M. Irwin, "An FPGA-based accelerator for cortical object classification," in *Proc. Design, Autom. Test Eur. Conf.*, Mar. 2012, pp. 691–696.

[27] J. Sabarad, S. Kestur, M. S. Park, D. Dantara, V. Narayanan, Y. Chen, and D. Khosla, "A reconfigurable accelerator for neuromorphic object recognition," in *Proc. 17th Asia South Pacific Design Autom. Conf.*, Feb. 2012, pp. 813–818.

[28] B. Lucas and T. Kanade, "An iterative image registration technique with an application to stereo vision," in *Proc. DARPA Image Understand. Workshop*, Apr. 1981, pp. 121–130.

[29] Z. Wei, D. Lee, and B. Nelson, "FPGA-based real-time optical flow algorithm design and implementation," *J. Multimedia*, vol. 2, no. 5, pp. 38–45, 2007.

[30] L. Yao, H. Feng, Y. Zhu, Z. Jiang, D. Zhao, and W. Feng, "An architecture of optimised SIFT feature detection for an FPGA implementation of an image matcher," in *Proc. Int. Conf. Field-Program. Technol.*, Dec. 2009, pp. 30–37.

[31] V. Bonato, E. Marques, and G. Constantinides, "A parallel hardware architecture for scale and rotation invariant feature detection," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 18, no. 12, pp. 1703–1712, Dec. 2008.

[32] J. Svab, T. Krajnik, J. Faigl, and L. Preucil, "FPGA based speeded up robust features," in *Proc. IEEE Int. Conf. Technol. Practical Robot Appl.*, Nov. 2009, pp. 35–41.

[33] K. Cannons and R. Wildes, "Spatiotemporal oriented energy features for visual tracking," in *Computer Vision ACCV 2007* (Lecture Notes in Computer Science), vol. 4843, Y. Yagi, S. Kang, I. Kweon, and H. Zha, Eds. Berlin, Germany: Springer-Verlag, 2007, pp. 532–543.

[34] K. Ratnayake and A. Amer, "An FPGA-based implementation of spatio-temporal object segmentation," in *Proc. IEEE Int. Conf. Image Process.*, Oct. 2006, pp. 3265–3268.

[35] E. Tsang, S. Lam, Y. Meng, and B. Shi, "Neuromorphic implementation of active gaze and vergence control," in *Proc. IEEE Int. Symp. Circuits Syst.*, May 2008, pp. 1076–1079.

[36] A. Savich, M. Moussa, and S. Areibi, "The impact of arithmetic representation on implementing MLP-BP on FPGAs: A study," *IEEE Trans. Neural Netw.*, vol. 18, no. 1, pp. 240–252, Jan. 2007.

[37] D. Le Ly and P. Chow, "High-performance reconfigurable hardware architecture for restricted boltzmann machines," *IEEE Trans. Neural Netw.*, vol. 21, no. 11, pp. 1780–1792, Nov. 2010.

[38] M. Pearson, A. Pipe, B. Mitchinson, K. Gurney, C. Melhuish, I. Gilhespy, and M. Nibouche, "Implementing spiking neural networks for real-time signal-processing and control applications: A model-validated FPGA approach," *IEEE Trans. Neural Netw.*, vol. 18, no. 5, pp. 1472–1487, Sep. 2007.

[39] S. Himavathi, D. Anitha, and A. Muthuramalingam, "Feedforward neural network implementation in FPGA using layer multiplexing for effective resource utilization," *IEEE Trans. Neural Netw.*, vol. 18, no. 3, pp. 880–888, May 2007.

[40] M. Shi, A. Bermak, S. Chandrasekaran, and A. Amira, "An efficient FPGA implementation of Gaussian mixture models-based classifier using distributed arithmetic," in *Proc. 13th IEEE Int. Conf. Electron., Circuits Syst.*, Dec. 2006, pp. 1276–1279.

[41] F. Benrekia, M. Attari, A. Bermak, and K. Belhout, "FPGA implementation of a neural network classifier for gas sensor array applications," in *Proc. 6th Int. Multi-Conf. Syst., Signals Devices*, Mar. 2009, pp. 1–6.

[42] T. Nguyen, K. Chandan, B. Ahmad, and K. Yap, "FPGA implementation of neural network classifier for partial discharge time resolved data from magnetic probe," in *Proc. Int. Conf. Adv. Power Syst. Autom. Protect.*, vol. 1. Oct. 2011, pp. 451–455.

[43] H. Meng, K. Appiah, A. Hunter, and P. Dickinson, "FPGA implementation of naive bayes classifier for visual object recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. Workshops*, Jun. 2011, pp. 123–128.

[44] H. M. Hussain, K. Benkrid, and H. Seker, "An adaptive implementation of a dynamically reconfigurable k-nearest neighbour classifier on FPGA," in *Proc. NASA/ESA Conf. Adapt. Hardw. Syst.*, Jun. 2012, pp. 205–212.

[45] A. Fazakas, M. Neag, and L. Festila, "Block RAM versus distributed RAM implementation of SVM classifier on FPGA," in *Proc. IEEE Conf. Appl. Electron.*, Sep. 2006, pp. 43–46.

[46] D. Anguita, L. Carlino, A. Ghio, and S. Ridella, "A FPGA core generator for embedded classification systems," *J. Circuits, Syst. Comput.*, vol. 20, no. 2, pp. 263–282, Apr. 2011.

[47] M. Misaki, Y. Kim, P. Bandettini, and N. Kriegeskorte, "Comparison of multivariate classifiers and response normalizations for pattern-information fMRI," *NeuroImage*, vol. 53, no. 1, pp. 103–118, Oct. 2010.

[48] D. Cox and R. Savoy, "Functional magnetic resonance imaging (fMRI) 'brain reading': Detecting and classifying distributed patterns of fMRI activity in human visual cortex," *NeuroImage*, vol. 19, no. 2, pp. 261–270, Jun. 2003.

[49] S. LaConte, S. Strother, V. Cherkassky, J. Anderson, and X. Hu, "Support vector machines for temporal classification of block design fMRI data," *NeuroImage*, vol. 26, no. 2, pp. 317–329, 2005.

[50] L. Fei-Fei, R. Fergus, and P. Perona, "Learning generative visual models from few training examples: An incremental Bayesian approach tested on 101 object categories," in *Proc. Comput. Vis. Pattern Recognit. Workshop*, Jun. 2004, p. 178.

[51] D. Hubel and T. Weisel, "Receptive fields, binocular interaction, and function architecture in the cat's visual cortex," *J. Physiol.*, vol. 160, no. 1, pp. 106–154, 1962.

[52] A. Antoniou, *Digital Filters: Analysis, Design, and Applications*. New York, NY, USA: McGraw-Hill, 2000.

[53] J. Friedman, T. Hastie, and R. Tibshirani, "Special invited paper. Additive logistic regression: A statistical view of boosting," *Ann. Stat.*, vol. 28, no. 2, pp. 391–393, Apr. 2000.

[54] T. Joachims, T. Finley, and C.-N. J. Yu, "Cutting-plane training of structural SVMs," *J. Mach. Learn. Res.*, vol. 77, no. 1, pp. 27–59, Oct. 2009.

[55] K. Crammer and Y. Singer, "On the algorithmic implementation of multiclass kernel-based vector machines," *J. Mach. Learn. Res.*, vol. 2, pp. 265–292, Mar. 2002.

[56] *HMAX Website*. (2010) [Online]. Available: http://riesenhuberlab.neuro.georgetown.edu/hmax

[57] J. Mutch and D. Lowe, "Object class recognition and localization using sparse features with limited receptive fields," *Int. J. Comput. Vis.*, vol. 80, pp. 45–57, Oct. 2008.

Garrick Orchard received the B.Sc. degree in electrical engineering from the University of Cape Town, Cape Town, South Africa, in 2006, and the M.S.E. and Ph.D. degrees in electrical and computer engineering from Johns Hopkins University, Baltimore, MD, USA, in 2009 and 2012, respectively.

He is currently a Post-Doctoral Research Fellow with the Singapore Institute for Neurotechnology, National University of Singapore, Singapore, where his research focuses on developing neuromorphic vision sensors and algorithms for real-time sensing on aerial platforms. His other research interests include mixed-signal very large scale integration design, compressive sensing, spiking neural networks, visual motion perception, and legged locomotion.

Dr. Orchard was named a Paul V. Renoff fellow in 2007 and a Virginia and Edward M. Wysocki, Sr. fellow in 2011. He is also a recipient of the JHUAPL Hart Prize for Best Research and Development Project, and the Best Live Demonstration prize at the IEEE Biocas 2012 Conference.

Jacob G. Martin received the Bachelor of Science and Ph.D. degrees in computer science from the University of Georgia, Athens, GA, USA, in 1999 and 2005, respectively.

He was a Post-Doctoral Researcher with Trinity College Dublin, Dublin, Ireland, where he researched multisensory processing in collaboration with experimental neuroscientists, and the Department of Neuroscience, Georgetown University Medical Center, Washington, DC, USA, where he performed research on human psychophysics, EEG, and computational models of vision to explore the dynamics of visual processing in the human brain. He is currently a Senior Staff Scientist in applied neuroscience with The Johns Hopkins University Applied Physics Laboratory, Laurel, MD, USA. His current research interests include cognitive neuroscience, human vision, brain computer interfaces, hybrid brain-machine vision systems, biologically-inspired machine vision, numerical analysis, spectral graph theory, information retrieval, and pattern recognition.

R. Jacob Vogelstein received the Sc.B. degree in neuroengineering from Brown University, Providence, RI, USA, and the Ph.D. degree in biomedical engineering from the Johns Hopkins University School of Medicine, Baltimore, MD, USA.

He is the Assistant Program Manager for applied neuroscience with the Johns Hopkins University (JHU) Applied Physics Laboratory and an Assistant Research Professor in electrical engineering with the JHU's Whiting School. He has worked on neuroscience technology for over a decade, focusing primarily on neuromorphic systems and closed-loop brain-machine interfaces. His research has been featured in a number of prominent scientific and engineering journals including the IEEE TRANSACTIONS ON NEURAL SYSTEMS AND REHABILITATION ENGINEERING, the IEEE TRANSACTIONS ON BIOMEDICAL CIRCUITS AND SYSTEMS, and the IEEE TRANSACTIONS ON NEURAL NETWORKS.

Ralph Etienne-Cummings (F'13) received the B.Sc. degree in physics from Lincoln University, Lincoln, PA, USA, in 1988, and the M.S.E.E. and Ph.D. degrees in electrical engineering from the University of Pennsylvania, Philadelphia, PA, in 1991 and 1994, respectively.

He is currently a Professor of electrical and computer engineering, and computer science with Johns Hopkins University (JHU), Baltimore, MD, USA. He is the former Director of computer engineering at JHU and the Institute of Neuromorphic Engineering. He is also the Associate Director for Education and Outreach of the National Science Foundation (NSF) sponsored Engineering Research Centers on Computer Integrated Surgical Systems and Technology at JHU. His current research interests include mixed signal VLSI systems, computational sensors, computer vision, neuromorphic engineering, smart structures, mobile robotics, legged locomotion and neuroprosthetic devices.

Dr. Etienne-Cummings has served as the Chairman of the IEEE Circuits and Systems (CAS) Technical Committee on Sensory Systems and on Neural Systems and Application. He was also the General Chair of the IEEE BioCAS 2008 Conference. He was a member of Imagers, MEMS, Medical and Displays Technical Committee of the ISSCC Conference from 1999 to 2006. He is a recipient of the NSF's Career and Office of Naval Research Young Investigator Program Awards. In 2006, he was named a Visiting African Fellow and a Fulbright Fellowship Grantee for his sabbatical with the University of Cape Town, Cape Town, South Africa. He was invited to be a Lecturer with the National Academies of Science Kavli Frontiers Program, in 2007. He was a recipient of publication awards including the 2003 Best Paper Award of the *EURASIP Journal of Applied Signal Processing* and the Best Ph.D. in a Nutshell at the IEEE BioCAS 2008 Conference. He has been recognized for his activities in promoting the participation of women and minorities in science, technology, engineering and mathematics.