An Energy-Efficient, Fast FPGA Hardware Architecture for OpenCV-Compatible Object Detection

Braiden Brousseau, Jonathan Rose

Electrical and Computer Engineering, University of Toronto
10 Kings Collage Road, Toronto, Ontario, Canada, M5S3G4
brousse1@eecg.toronto.edu
jayar@eecg.toronto.edu

***Abstract*—The presence of cameras and powerful computers on modern mobile devices gives rise to the hope that they can perform computer vision tasks as we walk around. However, the computational demand and energy consumption of computer vision tasks such as object detection, recognition and tracking make this challenging. At the same time, a fixed vision hard core on the SoC contained in a mobile chip may not have the flexibility needed to adapt to new situations, or evolve as new algorithms are discovered. This may mean that computer vision on a mobile device is the killer application for FPGAs, and could motivate the inclusion of FPGAs, in some form, within modern smartphones. In this paper we present a novel hardware architecture for object detection, that is bit-for-bit compatible with the object classifiers in the widely-used open source OpenCV computer vision software. The architecture is novel, compared to prior work in this area, in two ways: its memory architecture, and its particular SIMD-type of processing. The implementation, which consists of the full system, not simply the kernel, outperforms a same-generation technology mobile processor by a factor of 59 times, and is 13.5 times more energy-efficient.**

I. Introduction

The modern smartphone is a revolutionary device that brings together sensors, networking and compute power that has been reshaping the way we interact with the world [1]. They enable users to capture, record and process the information previously captured only by their senses. One of the highest bandwidth and most compelling capabilities of these mobile devices is the camera, which together with processing capability brings the promise of computer vision. Tasks such as object detection, face recognition and threat detection require real-time, low latency processing that prevent useful off-device (in the Cloud) processing. In the mobile context, the energy consumption of such tasks is a hard constraint, and so video processing on one of the applications processors is likely slow and energy-inefficient. While an on-SOC hard core [2] will provide a point solution, flexible hardware will permit more domain-specific solutions that can more easily evolve as new algorithms are developed. We believe that this motivates the inclusion of an FPGA within a mobile device that, once there, could be programmed to process other camera and sensor data as needed.

978-1-4673-2845-6/12/$31.00 © 2012 IEEE

In this paper we describe the design of an FPGA-based hardware architecture for object detection in the mobile context. While there has been significant prior work on FPGA implementations of object detection, we make four new contributions: a novel memory architecture, a novel SIMD processing architecture, a full-system implementation and a version that is compatible with the classifiers employed in the widely-used open source Open Computer Vision project, OpenCV [3]. We should note that the mobile context of this paper rests in the motivation, and in the comparisons given at the end. The architecture we describe is also applicable to non-mobile contexts.

We begin by providing the relevant background on computer vison - the object detection algorithm employed, and prior work in Section II. Section III presents the new hardware architecture for this algorithm. Section IV describes the measurement methodology and Section V gives performance, resource utilization, power and energy comparisons with other systems.

II. Object Detection Algorithm and Prior Work

Computer vision applications seek to to extract information about the world and their surroundings from a digital image or sequences of digital images. The output of a vision application will generally become the input for a more complex real world application. The code complexity of those tasks are increasing as researchers find ways to make them more generic and better able to cope with the variability found in the world. This is in part what drove the development of the Open Source Computer Vision project, OpenCV [3], which allows developers to explore the applications that use the results of computer vision while at the same time allowing researchers to continue to improve the underlying implementations.

OpenCV was released in 2006 as collaboration between computer vision researchers and Intel [3]. It currently contains more than 2500 algorithms addressing different aspects of computer vision. They have been submitted by the open-source community and are optimized by Intel to take advantage of the

highest-performance capabilities of CPUs, such as the vector-like instructions. OpenCV is being used all over the world and has been downloaded more than 2.5 million times.

Among the highest level and most directly useful functions provided by OpenCV is the object detection algorithm known as Multi-scale Haar Cascade Detection (MSHCD). A key contribution of the present paper is a hardware implementation that can accept the same inputs, and produce the same output, as the OpenCV implementation of MSHCD. Our implementation preserves the generality and parameterization of the OpenCV implementation. By making our hardware 'compatible' with OpenCV MSHCD in this way, it could potentially be used by developers who are using the OpenCV software. It also permits a very fair comparison to the software version, as it will have identical quality, and the software implementation will be of high quality. The next section describes the MSHCD algorithm.

A. Multi-Scale Haar Cascade Detection Algorithm

The MSHCD algorithm was introduced by Viola-Jones in 2001 [4]. The approach provides both high quality and relatively fast object detection with a fairly simple underlying algorithm. Since its publication, there have been more complex object detection algorithms proposed [5][6], but the method proposed by Viola-Jones still remains an active area of research due to its architectural simplicity and high available parallelism. This parallelism has been exploited on various computation platforms, including CPUs [7] GPUs [8][9], custom hardware [10] and FPGAs [11][12][13]. Vision researchers attempt to make detection algorithms smarter by making more complex and intelligent decisions, while engineering research has sought to evolve the the simpler approaches to achieve the greatest speed on available hardware platforms. The present work is an example of the latter.

We will now describe the structure of the computation for the MSHCD algorithm, and leave the reader to gain the details from [4]. The input to the MSHCD algorithm is the image to be searched (consisting of 8-bit grey scale pixels) and a classifier cascade (which is a characterization of the object being searched for). The output of the algorithm is a list containing the found (x, y) locations of each object, and a scale value which represents how large or small the object was. The MSHCD algorithm works by *sliding a window* across *multiple scales* of an image and at each window evaluating a *classifier cascade*.

In the sliding window approach, calculations are done on a group of pixels in a small bounding box around a central pixel. This bounding box is called a *window*. The center of this window moves to every location across the input as illustrated in Figure 1. At each location a computation is done and the

Fig. 1. Window Sliding Across an Image

algorithm can access any pixel that is contained within the window. Typically the window will step in the horizontal or vertical direction by one pixel. To achieve detection of objects at different sizes in the image, sliding window process is performed on a series of successively scaled images because the object is characterized at only one physical window size. Figure 2 illustrates constant size window relative to successively smaller versions of an input image. Eventually the image is small enough that the object that is being searched for, in this case a face, is roughly the size of window and can be detected.

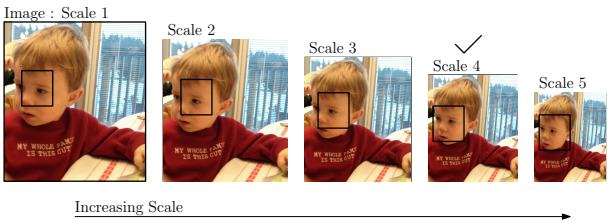

Fig. 2. Multiple Scales of Image

The Core Image Matching Computation

Once a window has been isolated, it is searched to see if the object exists in it at the given scale. The searched-for object is characterized by a hierarchy of structures, called, from simplest to most complex, a *Haar Feature*, a *Haar Classifier*, and a *Haar Cascade*. We describe each of these in turn.

Haar Features. A Haar feature is the fundamental unit of object characterization used by MSHCD. It is a standardized way of describing a few rectangles that should be present in an image window that the searched-for object. OpenCV employes five types of Haar features which are illustrated in Figure 3. Each is composed from either two or three

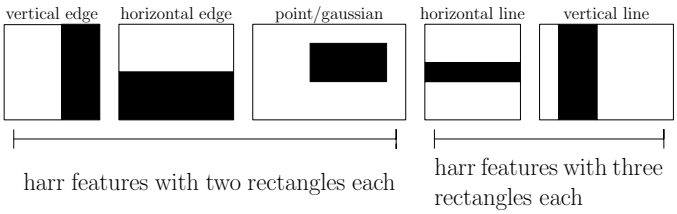

Fig. 3. Five Supported Haar Feature Types

rectangles and describe a point, line or edge. The size and location of these rectangles are specified relative to a window as seen in Figure 4. The characterization of more complex objects is done by grouping multiple features into *classifiers*. See [4] for a precise description of the computation.

Haar Classifiers. A Haar classifier is a collection of Haar features. Each window produced by the sliding window process is tested for *every* feature in the classifier. A sufficient number of features must present in the window for that window to be declared (as part of MSHCD) 'passed' for that classifier. This creates a trade-off between runtime and the quality of results: a classifier with a large number of features

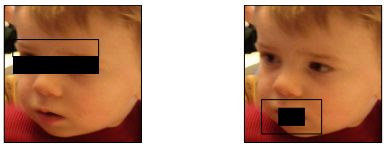

Fig. 4. Application of Haar Features

will likely do a better job of characterizing an object but each window will take longer to evaluate with that classifier, and there are a large number of windows. This trade-off is exploited by employing a *cascade* of successively larger classifiers, described next.

Haar Cascades. A cascade is a grouping of many varying-sized classifiers. The first classifiers contain a small number of features and then gradually increase in size. If a window passes the first classifier, it is sent to the second classifier and so forth. If any classifier in the sequence fails, the object is deemed not to have been detected and the process starts again on the next window. This is illustrated in Figure 5. This mechanism is efficient because it allows the algorithm to reject a window quickly if the earlier/smaller classifiers have failed when the window looks nothing like the object that is being searched for. The quality of results remains high because, for an object to be detected it has to pass all the classifiers which represent a very large number of features. This cascade methodology gives a significant performance advantage since one can assume the vast majority of windows in an image will not contain the object. It also means that work that aims to accelerate this algorithm should optimize for the common case – when windows fail after a small number of classifiers.

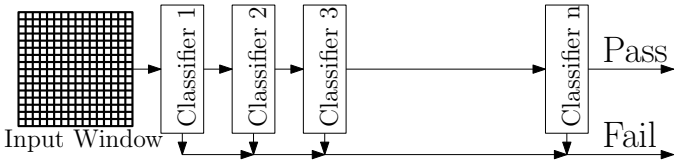

Fig. 5. Computation of a Cascade of Classifier Stages

B. Prior Work on Hardware for MSHCD

Figure 6 illustrates the basic structure that any MSHCD hardware architecture must have. The performance of this structure is limited by the effective memory bandwidth accessing two blocks of data: the classifier cascade and the image. The image size is roughly 1.2-4.8 Mbits, and for the OpenCV face classifier, roughly 2 Mbits. Prior work has evolved this basic structure in ways seeking to exploit the parallelism inherent in this algorithm. Previously the cascade memory has been split into multiple logical memories for independent access of different classifiers of the cascade [11]. In cases of small cascades all of the classifiers can be stored in registers and the amount of computational logic can reflect the size of the cascade [12].

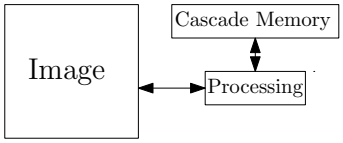

Fig. 6. Basic Compute Architecture

To dramatically increase how quickly the processing hardware could access data but at the same time manage total register usage, [13] stored the window and a *parametrizable* number of the classifiers in *registers* rather than memory, as illustrated in Figure 7. In this method, a parameterized number of classifiers are evaluated on a single window completely in parallel. Since each pixel of the window is in registers, all the values needed by all of the features can be read directly from wires in a single cycle. If the window failed earlier than the number of classifiers computed in parallel it would fail in a fixed number of cycles. If it did not this method would fall back to reading a cascade memory and computing the remaining classifiers sequentially. This method requires the classifier data values to be available at synthesis time, limiting flexibility of the system, but resulting in extremely good overall performance.

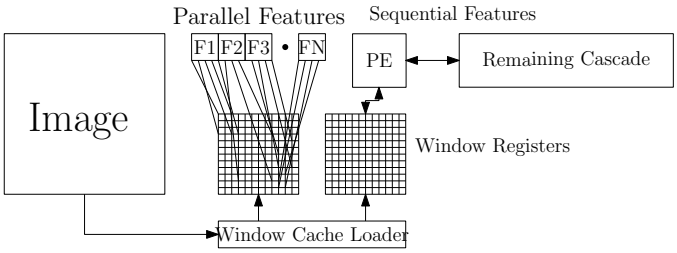

Fig. 7. Register Based Windows

This work, like ours, is a complete system that focused on large image sizes, large classifier cascade sizes, appropriate scaling factors, and high quality of results.

III. New MSHCD Hardware Architecture

A key goal in our work was to build a complete end-to-end OpenCV-compatible hardware system, that performs and accelerates all of the of the Open CV MSHCD software, rather than a more simple kernel as was often done in prior work. This section describes the architecture of the full system, and then focuses on the core computation that performs the MSHCD algorithm. Note that there are many specific components involved in making the system exactly match OpenCV that are neither performance-critical nor particularly novel which are not discussed here. The top level of the system is shown in Figure 8. It consists of storage for input images, cascades and software parameters, the USB communication interface to the host processor, and the MSHCD hardware. The host processor sends an OpenCV cascade of classifiers to

the device and then an 8-bit image. The hardware processes the image and writes the (x, y) location results to a FIFO which is transmitted back upon request to the host processor.

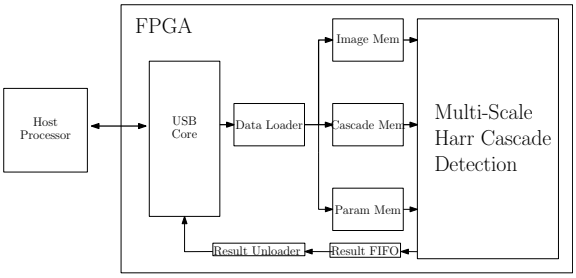

Fig. 8. Top Level Hardware Diagram

A. Main Computation Hardware

The main MSHCD hardware is illustrated in Figure 9. The input image is scaled and then transformed into an integral and square integral image, as described in [3]. The integral images are simple transforms that make it possible to compute Haar features constant time, regardless of the size of the rectangle. An array of processing elements (PEs) have the sliding window data loaded from these image transforms. The PEs also read from the cascade memories to evaluate classifiers on their windows. If the object was found in that window, that result is written to an output FIFO.

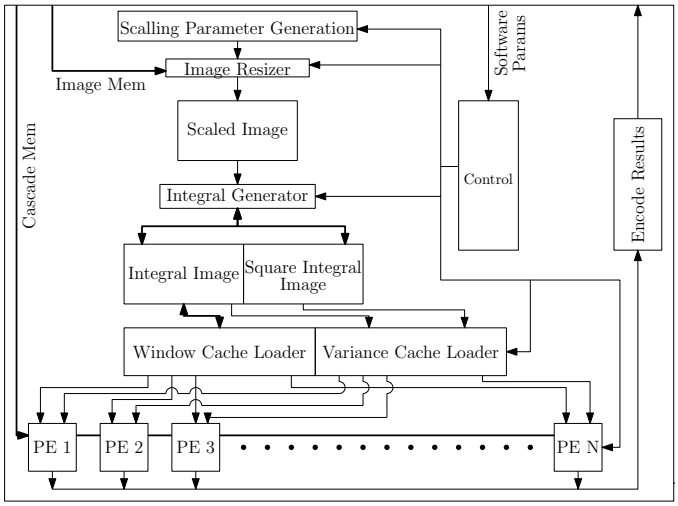

Fig. 9. Top Level of Computation Hardware

Each PE has a double buffered memory that can store the image data for one window. The input window is fed into a block which evaluates the cascade of classifiers while the secondary memory is being loaded with the next input window. The number of PEs in the system is a compile-time parameter called *Cores*.

B. Neighbouring Window Similarity

Architecturally, the system is performance-limited by memory transactions to load window caches in each PE and by memory transactions to read the feature data from the cascade data memory. Our solution to both of these bottleneck stems from the realization that neighbouring image windows look very similar as illustrated in Figure 10.

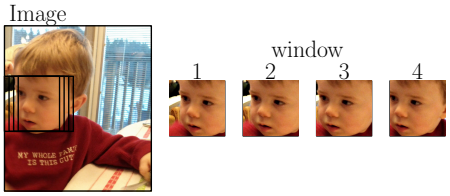

Fig. 10. Commonality Between Neighbouring Windows

From this observation we realize that a single pixel from the image belongs to many windows and we can speculate that neighbouring windows should take approximately the same amount of time to process because they look so similar. This drove the two major architecture decisions: performing parallel loading of the window cache, and employing SIMD-style processing between the PEs. Each of these innovations are discussed in detail in the next sections.

C. Parallel Window Cache Loading

A 20x20 pixel window, loaded one pixel at time, will require 400 cycles to load. For OpenCV cascades the first 4 to 5 classifiers can be read in roughly 400 cycles, and recall by design MSHCD creates these cascade to promote windows failing early. Therefore even if only one PE is used, every window that fails in less than 400 cycles will cause the PE to stall while it is waiting for new classifier data.

In this work we make use of the series of overlapping windows to enable low *average* load times per window. Figure 11 illustrates how we can take a single read from the image memory and put it into multiple window caches. In this example the entire first row could be loaded into all three windows in 8 cycles. If this is done for all 6 rows, it will take 48 cycles. In contrast it would take $6 * 6 * (3\ windows) = 108$ reads if these were loaded independently. The benefit increases with the number of windows loaded.

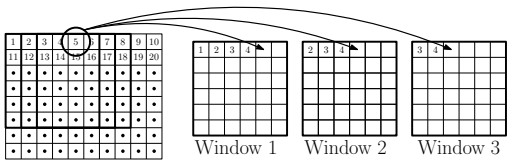

Fig. 11. Loading One Pixel into Multiple Windows

A second way to speed the window cache loading is by structuring on-chip image memory to allow multiple pixels to be read in every cycle. This can be accomplished simply by making the memory wider. The number of pixels that can be read per cycle is called the *PixelRate* and is a compile-time parameter of the system. Loading multiple windows simultaneously with a *PixelRate* of 3 is shown in Figure 12.

Notice that each block of data that is written to a window is a shifted version of the data stream read from the image. We

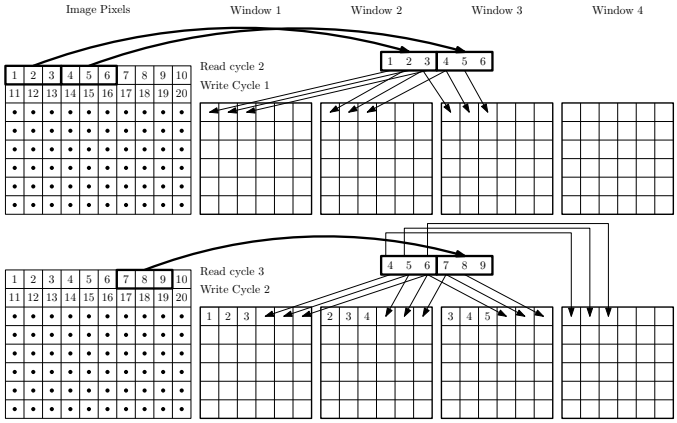

Fig. 12. Loading Multiple Pixels into Multiple Windows

call this process window alignment. The maximum amount of amount of alignment required is equal to the *PixelRate-1*. To avoid using costly mux trees or barrel shifters a series of fixed registers are used. The exact connections between the registers is controlled by a synthesis parameter as illustrated in Figure 13. These two memory arrangements - re-using memory reads and multiple pixel reads result in a much higher effective on-chip memory bandwidth from the image memory to the PEs.

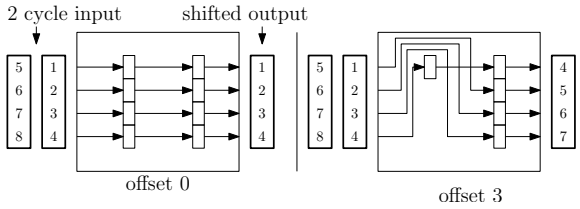

Fig. 13. Simple Alignment Hardware with Various Offsets

D. SIMD Processing

In order to make more efficient use of each read from the cascade of classifiers, we processes input windows using a SIMD-style approach. Here the data from the cascade can be considered the 'instructions' to be executed, as as they tell the PEs what to compute on their window caches. Once every window cache is filled with data, each PE is started at the same time but only one, the master processing element, addresses the cascade as illustrated in Figure 14. The data returned

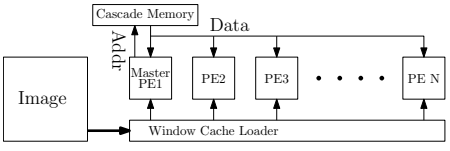

Fig. 14. SIMD Haar Detection Architecture

is broadcast to all PEs which simultaneously process their windows classifier by classifier. Recall from Section II that a window being processed can potentially fail at the end of every classifier. In our work, when an individual PE fails at the end of a classifier, it stops processing data, saving energy. However the PEs which have not failed still need data from deeper in the cascade memory and thus prevent PEs that are finished from starting again until every PE has finished.

IV. Measurement Methodology

Our goal is to build an FPGA-based hardware architecture for object detection that is significantly faster and more energy efficient than than the same algorithm implemented on mobile and other processors. In this section we describe the hardware and software platforms on which the MSHCD algorithm was implemented, and the nature of the input set used in the related measurements. This includes the input image set and the OpenCV classifier cascades that were used, as well as our methodology for measuring performance and power.

A. Platform Descriptions

We have measured the OpenCV software and our hardware on three different platforms: A desktop computer, a mobile processor and an FPGA, described below.

Desktop Computer. This machine uses a 6-core Intel 980x CPU over-clocked to 4.0GHz with 12MB of L3 cache. The system had 6GB of DDR3-2100MHz memory, and was running the Ubuntu Linux 10.04 operating system. The CPU is built on a 32nm CMOS process technology and was first available in the first quarter of 2010. This machine ran the MSHCD algorithm based on OpenCV 2.1, a single threaded configuration with low level SIMD optimizations (optimized by Intel) was used.

Mobile Phone. The mobile phone is an HTC Google Nexus One which employes a single core 1GHz Qualcomm QSD 8250 Snapdragon ARM SoC and has 512MB of RAM. The operating system is Google Android version 2.3.6. This processor was built on a 65nm CMOS process technology, and the first commercial device that has this processor was launched on December 7th, 2009. The mobile phone ran the MSHCD algorithm based on OpenCV 2.1. The library was compiled into native binaries linked to from the Java application environment of Android.

FPGA. The FPGA used was a Stratix IV GX EP4SGX530 which has 531K Altera ALUTs, 27Mbits of on-chip memory and 1024 18x18 multipliers [14]. The FPGA is built on a 40nm CMOS process technology and was available in the second quarter of 2010. It is part of the Terrasic DE4 Development board. This board is connected to a host computer via a USB connection and data is transferred to and from the FPGA though that connection. All designs discussed in this work were running on this FPGA and were clocked at 125MHz.

While each of the key chips in these platforms were built on different IC process nodes, they were all released at roughly

the same time and were considered high-end devices at the time of launch.

B. Input Cascades

OpenCV comes with 19 classifier cascades [3] that were generated with the OpenCV classifier training algorithms. Most of them are large and contain thousands of features. They all recognize some part of a person: the eyes, nose, mouth, face, and body. Of these cascades, three were selected to be used in our measurements. The exact three cascades, as provided by OpenCV, that were used are: haarcascade_frontalface_alt.xml, haarcascade_eye.xml, and haarcascade_upperbody.xml. As the file names suggest these cascades search for an upper body, face, eye or pair of eyes respectively as illustrated in Figure 15. I shows the total number of features and stages for the three cascades that were chosen.

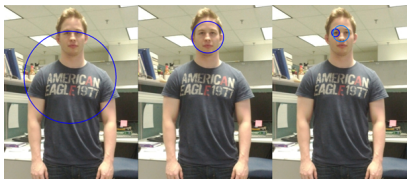

Fig. 15. Object Detection

Haar Features per Classifier Stage			
	Eye	Upper Body	Face
Total # Classifiers	24	30	22
Total Features	1066	2432	2135

TABLE I
HAAR CASCADES USED IN MEASUREMENTS

C. Input Images

We created our own set of images that contained several people facing the camera. For these pictures all of the OpenCV cascades that were selected will detect objects when they are run. The larger number of parts of the image that look like the object sought the more windows will fail later in the cascade. By selecting images that have multiple objects that should be detected, our measurements are attempting to isolate a case where the algorithm will be slow, and so we focus on the speed of the algorithm in unfavourable conditions. The MSHCD algorithm has been used in real applications at image sizes as small as 160x120. As this algorithm, in practice, does not need very large images to produce useful result we choose a modest, but reasonable size for this application: 320x240 pixels.

D. Performance Measurement

The performance of the MSHCD algorithm will be measured as the time it takes to process an input image at all scales and produce a final list of detected objects and their scales. This time measurement is a function of the input image, cascade and also the scale factor. In particular, a lower scale factor results in a greater number of intermediate scaled images that need to be processed. The amount of computation when varying the scale factor will be the same for each platform and thus will not affect the relative performance between them. The scale factor is set for all our experiments to 1.1 (10%).

We envision that a connection from a mobile processor to an embedded mobile FPGA co-processor would be significantly faster than the USB connection we implemented and certainly fast enough to transmit 320x240 images at 30 frames per second. This combined with double buffering of input data means the transfer time of the input data from the host system to the FPGA can be ignored in performance measurements.

E. Power and Energy Measurement

A key goal of this work is to gain an advantage in energy consumption in the mobile context, but this can also be of value in the server/desktop context. Bounds on the power consumed by the desktop CPU were estimated: an upper bound comes from the maximum thermal power dissipation from the processor specification: 130 Watts. We estimated a lower bound on power consumption of the processor as 60 Watts.

To measure the energy used by the mobile phone, we recorded the percentage discharge of the battery, both while it was idle and while performing the MSHCD algorithm. The Nexus One has a 1400mAh battery that operates at 3.7V and we estimated a voltage conversion efficiency of 90%. With this information power can be calculated as below:

$$\begin{aligned} mAh &= 1400mAh * \%\ battery\ used \\ average\ mA &= mAh/(discharge\ time) \\ mW &= (average\ mA * 3.7V * 0.9) \end{aligned}$$

The energy and power consumption of the FPGA system was determined using software tools provided by the vendor, Altera, and the Modelsim simulator. The power estimation software, Altera's PowerPlay Power Estimator, has detailed power characterization models of the FPGA. When these models are combined with with the exact signal switching behaviour of a user's circuit, generated from simulation, the tool can generate an accurate estimation of the overall power consumed.

V. EXPERIMENTAL RESULTS

As implied above, the Verilog hardware design of the MSHCD algorithm implemented on the FPGA is highly parameterized. Using these parameters, a total of 34 hardware systems were synthesised and tested with *PixelRates* of 1,2,4 and 8 and a total number of PE *Cores* ranging from 1 to 36. We begin by presenting the raw performance of the new engine, and how it compares to previous work discussed in Section II-A. First, we can report that the results from our hardware are identical to those produced by the OpenCV version 2.1 software, in line with our goal of compatibility. Although this compatibility is not technically garmented due to potential fixed-point errors it was true for the images tested

here and careful consideration was takin to minimize the impact of these errors. Then we will compare the energy efficiency of one configuration of our engine with a mobile device and desktop CPU.

A. MSHCD Engine Performance

Figure 16 illustrates the impact of the *pixelRate* on the performance of the system for different numbers of PE cores. It gives the execution time speedup for each number of cores, normalized to a *pixelRate* of 1 for each number of cores. Note that the number cores must be an exact multiple of the *pixelRate*, which is why some bars appear to be missing.

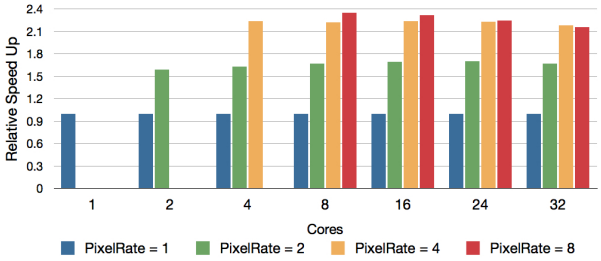

Fig. 16. Effectivness of PixelRate

The figure shows that having the ability to read multiple pixels per cycle increases overall performance by more than 2 times, with diminishing returns when the *pixelRate* reaches 8. This is an indication that the system is consistently loading new data before the SIMD cores have finished computing their current windows. This performance improvement comes with virtually no increased resource utilization as it just takes better advantage of the flexible memory system on the FPGA.

Figure 17 shows how well the system performance scales as the number of *cores* is increased. The figure gives the relative performance increase of the hardware variants normalized to the slowest system - one core and a *pixelRate* of one. Each curve on Figure 17 represents different *pixelRate*. Table II shows the raw performance data for a few of the system configurations.

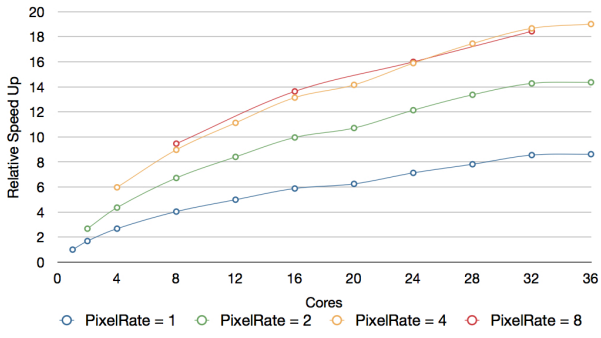

Fig. 17. Performance vs Number of Cores

Figure 17 shows consistent but sub-linear scaling up to approximately 32 cores, at which point the performance improvements become negligible. The largest speed-up occurs

Execution Time of 320x240 Images in (ms)				
Cores\PixelRate	1	2	4	8
1	381	NA	NA	NA
2	226	143	NA	NA
4	142	87	64	NA
8	94	57	42	40
16	65	38	29	28
24	53	31	24	24
32	45	27	20	21

TABLE II
EXECUTION TIME PER FRAME FOR EACH HARDWARE CONFIGURATION

with a *pixelRate* of 4 and 32 *cores* is 18.4 times. We believe the saturation of performance is due to the following effect: as the number of cores increase, the number of windows being processed in parallel increases. As this happens the windows contain image data that is less correlated, as illustrated in Figure 18. When this happens the SIMD-executing PEs become less efficient because it is more likely that the PEs failing at an earlier classifier stages will be stalled waiting for the other PEs to finish.

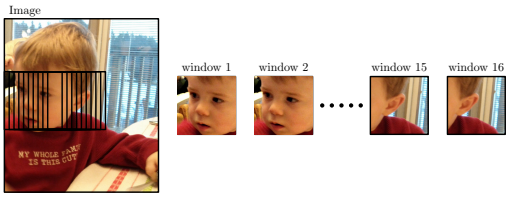

Fig. 18. Distance Windows

B. Performance Comparison with Previous Work

Recall that [13], discussed in Section II, processes one window at a time. However for that window every feature in a fixed number of classifiers is computed simultaneously. The larger number of classifiers processed in parallel, the faster the system will run but the more hardware will be required. In our system performance and resource utilization increases with number of windows processed in parallel. Here we will compare the two systems.

The comparison with [13] must be approximate because the same cascade, images, window sizes and scaling factors were not used. Normalizing where possible and knowing that the cascades used in both works are roughly the same size a performance comparison will be valuable even if it is only a first-order approximation.

Figure 19 shows the number of lookup tables (ALUTs) and registers used for both system as a function of a normalized performance metric - the average number of cycles per window computed. For the fastest configurations both designs perform almost identically, 4.88 cycles per window vs 4.7 cycles per window. This is quite a remarkable result given that the method by which each design exploits parallelism in this algorithm is quite different.

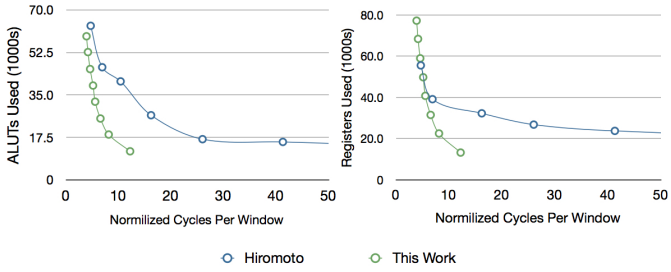

Fig. 19. Resource Utilization

Both works have to add significant resources to improve performance at the high end, indicated by the steep slope of the curves when cycles per window are low. If an end user can accept slower performance the resource utilization savings for our system are far greater than in [13]. This is a result which is beneficial in the context of a mobile FPGA co-processor where resources are likely to be more limited.

The work done by Hiromoto [13] does not measure power, so a direct comparison cannot be done. Qualitatively we might expect the power and energy usage of our engine to be lower for two reasons: First, as was just shown, is that the total resource utilization of our system tends to be better for a given performance level resulting in less static power. The second is that our system does not perform wasted calculations: each SIMD PE will stop after the window it is working on has failed a classifier. In [13] every feature in a fixed number of stages is computed in parallel regardless of whether or not the window would have failed before that stage.

C. Performance and Energy vs. Processors

We now compare the speed and energy usage of a 32 core engine with a PixelRate of 4 to the desktop and mobile processors described in Section IV-A. Table III gives each system, it's power consumption, execution time per image, and speedup relative to the mobile processor. Notice that the hardware is 59 times faster than the mobile processor, and 3.9 times faster than a high-end desktop. More importantly, the FPGA hardware engine is 13.5 times more energy efficient on this problem than the mobile processor, and between 68 and 147 times more energy efficient than the desktop processor. We can speculate this efficiency comes from the application specific nature of the both the core computation and the cache architecture as this is a memory access heavy application.

This efficiency is significant given how conservative the result is. Our 32 core system only uses 30% of the logic and half the DSPs of our FPGA. The Stratix IV FPGA is also designed for performance and high logic capacity. This is compared with a mobile processor designed for power efficiency. One could expect this energy efficiency delta to increase for a FPGA fabric designed for mobile.

Performance Per Computed 320x240 Image		
Device	Time (ms)	Speed Up
Mobile QSD8250	1180	**1**
Intel 980x	77	**15.3**
FPGA EP4SGX530C2	20	**59**
Power Per Computed 320x240 Image		
Device	Power (W)	Relative Power
Mobile QSD8250	0.777	**1**
Intel 980x (130W)	130	**167**
Intel 980x (60W)	60	**77**
FPGA EP4SGX530C2	3.4	**4.38**
Energy Per Computed 320x240 Image		
Device	Energy (J)	Relative Energy
FPGA EP4SGX530C2	0.068	**1**
Mobile QSD8250	0.916	**13.5**
Intel 980x (60W)	4.62	**68**
Intel 980x (130W)	10.01	**147**

TABLE III
PERFORMANCE, POWER AND ENERGY PER OPERATION

VI. CONCLUSION

In this paper we have presented a new FPGA-based hardware architecture for the multi-scale Haar classifier detection algorithm. It employs a novel memory structure to increase the effective bandwidth of the image memory, and a SIMD-style of execution of the features that leverages the similarity between neighboring windows. The implementation is significantly faster than a mobile processor and consumers a factor of 13.5 times less energy. It is also faster than a high-speed desktop by a factor of 4 while consuming at least 70 times less energy. In the future, we will build this into a portable mobile system, and enhance it in several ways, including vertical as well as horizontal window sliding.

REFERENCES

[1] Samsung. (2012) Samsung Galaxy SIII - Designed for Humans. http://www.samsung.com/global/galaxys3/.
[2] J. Oh and G. Kim, "A 320mW 342GOPS Real-Time Moving Object Recognition ADS Processor for HD 720p Video Streams," *IEEE J. Solid-State Circuits*, pp. 220–221, 2012.
[3] W. Garage. (2006) OpenCV Wiki. [Online]. Available: opencv.willowgarage.com/wiki/
[4] Viola and Jones, "Robust real-time object detection," *International Journal of Computer Vision*, vol. 57, no. 2, pp. 137–154, 2001.
[5] Gall and Lempitsky, "Class-specific Hough Forests for Object Detection," *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1022–1029, 2009.
[6] S. Maji and J. Malik, "Object Detection using a Max-Margin Hough Transform," *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1038–1045, 2009.
[7] P. Sudowe, "Efficient Use of Geometric Constraints for Sliding-Window Object Detection in Video," *International Conference on Computer Vision Systems*, pp. 11–20, 2011.
[8] D. Hefenbrock, "Accelerating Viola-Jones Face Detection to FPGA-Level using GPUs," *IEEE Conference on Field-Programmable Custom Computing Machines*, 2010.
[9] M. Ebner, "Evolving Object Detectors with a GPU Accelerated Vision System," *Evolvable systems: from biology to hardware*, pp. 109–120, 2010.
[10] C.-R. Chen and W.-S. Wong, "A 0.64 mm 2 Real-Time Cascade Face Detection Design Based on Reduced Two-Field Extraction," *IEEE Trans. VLSI Syst.*, vol. 19, no. 11, pp. 1937–1948, 2011.
[11] C. Cheng, "An FPGA-based object detector with dynamic workload balancing," *IEEE Conference on Field-Programmable Technology*, 2011.
[12] Y. Wei, "FPGA Implementation of AdaBoost for Detection of Face Biometrics," *IEEE International Workshop on Biomedical Circuits and Systems*, pp. 6–9, 2004.
[13] M. Hiromoto, "Partially Parallel Architecture for AdaBoost-Based Detection With Haar-Like Features," *IEEE Trans. Circuits Syst.*, vol. 19, no. 1, pp. 41–52, 2009.
[14] Terasic. (2011) Altera DE4 Development and Education Board. [Online]. Available: http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=138&No=501&PartNo=2